Strategising RoboCup in Real Time with Uppaal Stratego

Philip Irming Holler, Magnus Kirkegaard Jensen, Hannah Marie Krøldrup Lockey
6. Semester Computer Science
Selma Lagerløfs Vej 300 9220 Aalborg Ø
pholle17@student.aau.dk, magnje17@student.aau.dk, hlocke17@student.aau.dk
26th of May 2020

Abstract

RoboCup 2D soccer simulator was designed to provide a playing ground to further research in Artificial Intelligence. One of the main challenges provided by RoboCup is generating clever strategies for the players given a partial and noisy view of the game state. Additionally, RoboCup is timing sensitive, meaning that all decisions have to be made within each server tick of 100ms. This paper presents a method for generating strategies by modelling players and scenarios as timed automata in Uppaal. The newest version of Uppaal, called Stratego, allows for synthesising strategies optimising some reward value, which is used to guide the decisions process. In order to stay within the time frame of 100ms, two approaches were tested, namely forecasting the current game state and generating a strategy asynchronously for a later point in time, and generating strategies beforehand and saving them in a lookup table. Controllers for the player agents, the online coaches and the trainer agents were developed to apply the strategies and evaluate their effectiveness. Four different timed automata were included in testing; one timed automaton for positioning the goalie, one for conserving stamina for an entire game, one for determining a pass-chain and finally one for deciding whether to pass or dribble when in possession of the ball. We found that the strategies could be successfully generated and used within the time constraints of RoboCup, using our proposed methods. The pass-chain strategy is, however, only applicable, if the rule of message delay from online coach to the players is removed from the server. The strategies generated by the goalie positioning and stamina timed automata improved performance measured in terms of goalie interventions and running distance respectively. The pass-chain model appeared to perform slightly better than default behaviour, but we were unable to identify a statistically significant performance difference. The possession model was functional, but results indicated that it performed poorly compared to the default behaviour implemented in our player agents.

1 Introduction

Robotics involves many complex decision problems with tight time constraints that are often difficult to solve using traditional rule based programming. One example is control of physical robot components, where complex movement must be coordinated to respond to sensory data in real time. This category of problems also extends to higher levels of abstraction, such as strategising to solve problems in a dynamic environment.

The RoboCup federation aims to push the limits of Artificial Intelligence in robotics by providing platforms for soccer competitions between autonomous robots (Robocup Federation, 2020a). In this paper we focus on the strategisation part of the problem, as represented in the in the RoboCup 2D multi-agent soccer simulation (Robocup Federation, 2020b). This simulation includes a dynamic environment with a large amount of possible configurations that makes linear search for optimal solutions infeasible. The RoboCup players are autonomous agents reacting to stimuli from the environment. In order to make the players perform well in the game, some way of quickly creating effective strategies is required.

Uppaal is a modelling and verification tool, that allows for modelling the behaviour of timed systems in terms of states and transitions between states. Additionally, Uppaal is able to analyse and verify properties of the models using efficient algorithms (Uppaal, 2019). The latest edition of Uppaal, Uppaal Stratego, enables strategy generations based on the models using different machine learning algorithms, like Q-learning (David et al., 2015). Stratego has already been applied to solve problems of real time systems such as a floor heating control (Larsen et al., 2016) and traffic light control (Eriksen et al., 2017). Both approaches use Stratego to generate strategies in real time, but neither with the tight time constraints required by RoboCup.

The aim of this paper is to introduce methods for applying the Uppaal Stratego tool to solve problems that require swift reactions to a dynamic environment. We create a RoboCup team named RoboPaal, which utilises the presented methods for generating strategies in real time with Uppaal Stratego.

2 Background Information

This section will provide some background information needed to understand the context of the solution. Additionally, some frequently used terms are defined.

Definition 1. An **objective** is defined as a subgoal of a strategy, which is set for a single player agent. An example of this could be; move to position *x,y* on the field. An objective can require several player actions, i.e. dash, kick and turn, to complete, or none if the objective is already fulfilled.

Definition 2. A **strategy** is a set of objectives assigned to one or more player agents. An example of a strategy for one player can be seen in figure 1 where *run_to_ball*, *face_target* and *kick_to_target* are objectives. A strategy is carried out by the player agents by fulfilling their objectives.

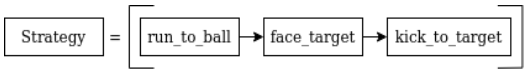


Figure 1: Strategy example for one player

Definition 3. The definition of a **model** in this report, is a timed automata modelled within the scope of Uppaal Stratego (David et al., 2015).

2.1 RoboCup

The idea of RoboCup was conceived by a group of Japanese researchers in 1992. The intent was to promote science and technology, especially with regards to artificial intelligence, through the game of soccer. An official league for both real robots and a 2D and 3D simulation was established in 1997. The overarching goal of the project is to have a team of humanoid robots beat the human world champions at soccer by 2050. (Robocup Federation, 2020a)

For this paper we will use the 2D simulator. Some big challenges when working with the RoboCup 2D soccer simulator include interpreting noisy sensor data, strategising multiple agents independently and dealing with time sensitivity since the server enforces a 100 ms tick rate.

In order to achieve a level playing field for all participants an official simulator was developed as well as some rules. The rules are defined on the RoboCup website and include for example a limitation, that all communication between agents has to go through the server (Robocup Federation, 2019). The 2D simulator is, at the time of writing, still actively being developed and maintained. The simulator software consists of 3 components: A soccer server, a soccer monitor and a soccer log player. The soccer log player is for debugging the players. It will not be discussed further, since it was not used for this paper. All three components are open-source and publicly available through GitHub (Rodrigues et al., 2020).

The **Soccer Server** is the main component which stores and updates the current state of a game. A team can connect three different types of agents to the server on three different ports. An online coach, i.e. a live coach for official matches, a coach/trainer for training models/algorithms and lastly a player agent. The three different agents communicate with the server using different protocols and with access to different functionality. These three entities have to be developed by the user of the simulator. For this paper version 16.0 of the server was used. (Rodrigues et al., 2020)

The **Soccer Monitor** can be used to give a visual representation of the game state. Version 16.0 of the monitor was used for this paper. Additionally, the soccer monitor allows for replaying log files of previously played games. (Rodrigues et al., 2020)

Player Agents can only communicate with each other through the server. All actions taken by a player are communicated to the server using UDP, after which they will be executed to influence the game state. The interface of the server includes actions like *dash*, *kick*, *turn*, *turn_neck* and *say*. The kick and dash actions are accompanied by a power parameter indicating how hard to kick and how fast to dash. The power parameter is a value between 0 and 100. The player state has a stamina level that dictates the effectiveness of the dash and kick actions, which by default starts starts at the upper limit of 8000 stamina. Dash actions will consume stamina equal to the power of the action. Stamina regenerates at a rate of at most 30 per tick throughout the game. (The RoboCup Soccer Simulator Maintenance Committee, 2020)

Player agents periodically receive sensory data of three different types: visual data, body sensor information and auditory data. The visual data includes everything within field of view, but distorted with noise proportional to how far away the objects are. Visual messages from the server to the player are in RoboCup called *see* messages. The visual data includes, but is not limited to, distances and relative directions to flags, other players and the ball. Since the flags' positions are static and known beforehand, they can be used as positioning beacons. Furthermore, the player receives so called *Body_Sense* messages, which include current values of stamina of the player, head angle and much more. *Hear* messages include messages from the referee, other players and the coach. (The RoboCup Soccer Simulator Maintenance Committee, 2020)

The **Online Coach** is meant for communicating and generating overall strategies for the team, as well as substituting players during a match. Unlike the player agent, the online coach receives perfect information about the entire game state. The online coach communicates via short messages broadcasted to the players through the server. The online coach also has restrictions on when and how often, it can communicate. (The RoboCup Soccer Simulator Maintenance Committee, 2020)

The **Trainer**, also called the **Coach**, is meant for developing and testing strategies. The trainer has the same perfect view of the game state as the online coach. There are two important distinctions between the online coach and the trainer. The trainer can directly impact the game by moving objects and changing the game state while the online coach cannot. The trainer can, however, not be used in official matches while the online coach can. (The RoboCup Soccer Simulator Maintenance Committee, 2020)

There are many **rules** for the RoboCup tournaments. Most of them are enforced by the server controlled referee, which can for example give out yellow or red cards if a player is tackled. Some rules, however, are what the RoboCup federation calls "Code of Honor" or "Fair Play" rules. An example of a fair play rule is, that a team is not allowed to form a wall of players in front of their own goal. (Robocup Federation, 2019).

2.2 Uppaal Stratego

Uppaal is a modelling and verification tool developed by the Department of Information Technology at Uppsala University and the Department of Computer Science at Aalborg University. The latest edition of Uppaal is called Stratego and includes tools for creating strategoies in addition to the regular Uppaal functionality like model checking and verification (Uppaal, 2019). A strategy in Uppaal Stratego consists of a number of transitions in a timed automata depending on the values of variables and clocks.

One of the main challenges of modelling in Uppaal Stratego is making a model, that correctly represents the scenario, but still does not cause a state explosion. A state explosion would be a timed automata with a state space too big to analyse. The strategies are generated according to a query formulated in a query language developed for Uppaal. Each query contains the variable or clock, that should be optimised (David et al., 2015). Strategies are generated using different machine learning methods. Uppaal Stratego currently supports co-variance, Splitting, Regression, Naive, M-Learning and finally Q-learning, which is the default (David et al., 2014). In this paper, Stratego will only be used in its default setting using Q-learning. Additionally, Uppaal Stratego has a graphical user-interface for modelling of the timed automata. Models in Uppaal are saved as XML files, which enables easier direct manipulation of the model without having to open the file in the Uppaal interface. The Uppaal verifier, called verifyta, is a binary used to run strategy generation queries on the models.

2.3 Related Work

To our knowledge, strategising using Uppaal or other modelling of timed automata has not been done before for RoboCup. This approach has, however, been used for other real time systems, such as for a floor heating controller (Larsen et al., 2016) and for a traffic controller (Eriksen et al., 2017).

For the floor heating problem, they propose an online synthesis method for continuously creating periodic strategies on the fly instead of learning an entire strategy at once. They reason that this is necessary is because the computations needed to learn an effective strategy in Stratego grows exponentially with the amount of states, and so generation of effective strategies becomes limited in the time horizon. Using this online approach, a heating control strategy is created for the near future, and then recalculated when the period ends. They conclude that their new floor heating controller made with Uppaal Stratego was better than the existing controller in the test house.

The traffic controller uses Uppaal Stratego to create an online controller for intelligent traffic lights. They reduced waiting times at traffic lights by continuously creating new strategies using Uppaal Stratego.

3 Design

This section describes the architecture of the system that allows us to generate strategies using Uppaal and apply them to the player agents. This section also covers the two different strategy types that we use to achieve fast application of strategies.

3.1 Architecture

Our implementation uses Uppaal in two different ways: through the player agents themselves and through the coach. The architecture can be seen in figure 2.

In the first approach, a player agent can represent its view of the world as a Uppaal model, which can then be used to generate some strategy with Stratego. The player agent then interprets this strategy into objectives and acts upon them by sending actions to the soccer server. This approach allows creating individual strategies for each player, based on that player's view of the world. However, as described in section 2.1 the player's view of the world is distorted and incomplete. For this reason generating an overall strategy for the entire team would be very difficult given only the view of the game state from a single player agent.

The second way of using the Uppaal verifier for generating a strategy is using the online coach. As described in section 2.1 the online coach has a perfect view of the game state, which enables it to generate strategies based on the entire game state. In order to execute the strategy, however, an additional communication step is required. First the online coach receives the game state from the server, this happens every tick (100ms). The game state is then forwarded into an Uppaal model, and a strategy is generated and returned to the online coach. The online coach can, as mentioned in section 2.1 not directly execute the strategy, since it does not have control over the players. Instead the online coach can communicate the strategy with a *Say* action through the server and to the player agents. Here the strategy can be used for actions impacting the game state.

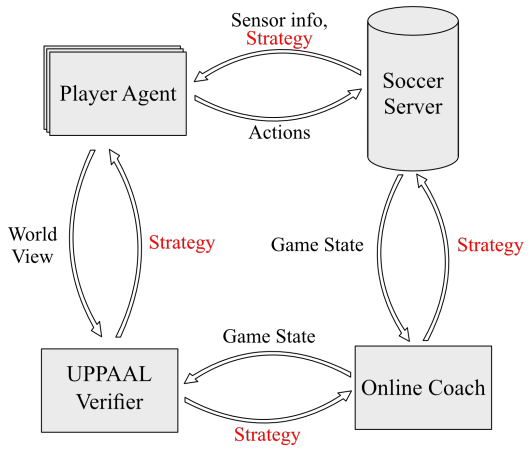


Figure 2: Program Architecture

3.2 Strategising RoboCup with Uppaal

When strategising a game of RoboCup with Uppaal, one can either model an entire game state or subparts of a game and try to strategise using this model. A game of RoboCup can, however, have up to 11 player agents for each team. Each player has more than 40 variables describing their state. Many of these variables have values in the domain of real numbers. Creating a general model describing an entire game state would lead to a state explosion too big to handle for Uppaal. For this reason, we determined it infeasible and instead worked with subparts of a game. A subpart could be a given scenario, for example a free kick. This would limit the amount of states, that the model has to represent.

This approach, however, might still not be enough, due to the sheer amount of variables allowing for many game states. For this reason we introduce additional abstractions. For a strategy for passing the ball, we might limit the model information to which player is in possession of the ball and the location of the 5 nearest players. Additionally, the actions represented in the model, and therefore also the strategy output, are further abstractions over series of single actions executed by a player agent.

Another obstacle when working with strategies is that of noisy data. As described in section 2.1 the player agents receive inaccurate data from the server, meaning that the data might be imprecise or sometimes not even present. This can lead to strategies generated on wrong assumptions of the game state. This problem is mitigated by saving old game states and matching it with the new one to make educated guesses about the environment. Since the player agents continuously receive sensory data, the missing data is likely to be present eventually.

3.2.1 Strategy Types

While strategising RoboCup with Uppaal, we identified two distinct methods for generating strategies. The methods differ in when the strategies are generated.

An **Online Strategy** is generated on the fly when it is needed. This approach is very similar to the online methods used in the floor heating problem (Larsen et al., 2016) and the traffic controller (Eriksen et al., 2017). However, our approach must adhere to much tighter time constraints, since ideal control of the player agents require response times below 100 milliseconds. The models for generating online strategies are altered at generation time to represent the current state of the game, and the output strategies provide some immediately applicable objectives. This could for instance be finding the optimal target for passing the ball based on the current positions of teammates and opponents.

If generation takes too long, the strategy might be outdated by the time it is generated. We reduce the risk of outdated strategies by using a forecasted game state when generating an online strategy. This does, however, introduce the risk of generating a strategy for a game state, that never materialises. Additionally, if the strategy generation takes longer than one server tick, i.e. 100ms, it has to be asynchronous to avoid halting player agent.

An **Offline Strategy** is generated before a game is run. An offline strategy has to be stored in order for the players to access them when needed. We propose lookup tables where the key corresponds to some deciding element of the game state. Offline strategies cannot feasibly be generated for all scenarios of the game, due to the large amount of possible states. Instead, this kind of strategy can be developed for scenarios that are predictable and repeated often. At run time the agent can then search through the offline strategies, and apply them if they match the current game state. Offline strategies have an advantage over online strategies, in that they can use more computation generating the strategy, since they do not have to be generated in real time.

4 Implementation

This section describes how the architecture and all of its subparts were implemented. The implementation of the architecture is done using version 3.6 of the Python programming language.

4.1 Player agents

The biggest part of the implementation is the **Player Agents**. The rules of RoboCup, see section 2.1, disallow communication between player agents that does not go through the official protocol of the soccer server. For this reason all player agents were developed to run on independent threads. To facilitate concurrent server communication and player thinking, each player consists of two subthreads The first sub-thread is the **Communication thread**, which is responsible for receiving and sending messages from and to the soccer server. Every received message from the server is simply put into a thread safe queue. This thread safe queue is then accessed by the other sub-thread, namely the **Thinker thread**. This thread is responsible for parsing the received messages and generating actions given a game state. The actions are put into a thread safe queue accessed by the communication thread. This architecture of the player agents allows for both thinking, i.e. generating actions that could be based on strategies, and communicating at the same time.

4.1.1 Interpreting Noisy Data

Since the sensory data a player receives from the server contains large amounts of information, we first perform some parsing and analysis to achieve an abstraction level suitable for the Uppaal models. However, as mentioned in section 2.1, the data received by the players from the server is noisy and sometimes even missing. This means, that the players have to interpret the data to get an idea of their actual value, since the face value of the data cannot be trusted. An example of the noisy data is that the distance to the ball, and every other object, is being judged incorrectly by up to 10 percent (Akiyama, 2010).

To mitigate the problem of noisy data, several measures were taken. Our player agents approximate their positions

and directions by using trilateration on all visible flags on the field, and then averaging the results. Once the global direction and position of the player is approximated, positional information of the rest of the objects can be filled in using the information about their relative distance and direction.

Another way we mitigate the noise problem is by keeping a history of how the player recently perceived the game state. A comparison of the current game state with the recent history can expose inconsistencies in the data, which can then be handled in different ways, such as using a weighted average. This approach is used when approximating the speed of the ball. Additionally, we associate a time stamp with every piece of information which indicates when it was received from the server. This allows us to approximate positional data in case an object goes missing for a few ticks. For instance if the ball is travelling at some speed in some direction, the position can be forecasted, given that no player interacts with the ball. This is also used when forecasting ball and player positions for use in online models, as we do in the possession model, seen in section 4.4.2.

4.1.2 Strategising Player Agents

Every time some message from the server is parsed by a player agent in the thinker thread, that agent checks to see if some strategy could be used or generated given the current game state. If the game state matches the requirement for an offline strategy, objectives are immediately applied for the player. If the game state matches an online strategy configuration, then the agent initiates strategy generation using the Uppaal module of the program, see section 4.4. Since generating an online strategy may take longer than 100ms, the player may be stalled until it is finished. To avoid this, strategy generation is done asynchronously using a separate thread. Upon completion the Uppaal module returns a series of objectives to the player agent.
Player agents can also receive objectives for a strategy generated by the online coach. These are received as *say* messages from the Online Coach, through the server and to the agent, see more in section 4.2. In the case that no suitable strategies are found, objectives are determined by default hard coded behaviour.

4.1.3 Objectives

In order to act on the objectives of a strategy, the *Objective* class was introduced. An instance of the *Objective* class takes as parameters the perceived game state of a given player, an action generator function, a completion criteria function and a maximum duration of the objective. This functional approach was used, since it is not known exactly how many actions are needed to achieve a given objective, and because these actions depend on the ever changing game state. For example, if the objective is to move to the position of the ball, then this can require an arbitrary amount of actions, since the ball itself can be impacted by other players. The **action generator function** is used to generate actions for fulfilling the completion criteria using the updated view of the game state at every tick. Since the completion criteria may never be met, the maximum duration parameter was introduced, which provides some upper bound to the duration of an objective. The **completion criteria function** can be run on each tick to check whether the objective has been fulfilled, in which case it returns true. An objective will therefore be executed until some completion criteria is met or the **maximum duration of the objective** has elapsed.

4.2 Online Coach

The **Online Coach** was implemented using a pattern similar to the player agent's. The online coach is also an independent thread with two sub-threads, namely a communication thread identical to the one used in the player agent and a thinker thread. The difference lies in the parser for the messages from the soccer server. The parser has to be different, due to the communication protocol for online coaches to the soccer server being different as well as the data from the server being complete and not distorted, which eliminates the need for a lot of interpretation. At every tick the online coach receives a game state from the server. It then compares the game state to the requirement for the models, in our case simply the pass-chain model, see section 4.4.1. For example it is necessary for the team to be in possession of the ball in order to optimise a pass-chain. If the conditions for the model is met, the online coach will generate a strategy using Stratego and communicate it to the player agents through the server. However, this approach has to account for the coach message delay, normally set to 50 ticks.

4.3 Trainer

The implementation of the **Trainer** is very similar to the implementation of the online coach, see section 4.2. The pattern with a thread and two sub-threads for thinking and communicating, respectively, was reused. This is due to the trainer using the exact same protocol as the online coach. The trainer does, however, have a different use, which requires different functionality. The trainer can change the current game mode, for example to free kick or corner kick, and even overrule the referee. The trainer is in our implementation able to move objects on the field. This was an essential feature for testing the performance of the strategies, that we generate, since it allows us to test the same situations repeatedly.

4.4 Implementing Models and Strategy Generation

We introduce one offline and three online Uppaal strategies. The **Goalie Defence Model** is used for an offline strategy while the **Pass-Chain Model**, the **Stamina Model** and the **Possession Model** are used for generation of online strategies. In order for the online strategy models to be relevant for a given game state, some variables of the model, like specific player positions, are updated in real time. For this purpose the **UppaalModel** class is developed. This class provides an interface for reading and updating the XML files of the Uppaal models.

In our models, the possible objectives that the player can pursue are represented as controllable transitions. For example a transition might represent passing the ball to

a specific player. When Stratego synthesises a strategy, it provides an estimated value of taking each possible transition of the model, i.e. the value of pursuing each objective. Uncontrollable factors such as the movement of opponents, is represented as uncontrollable transitions (dotted lines). These transitions can influence the state of the model and the estimated reward of the controllable actions, but will not appear directly in the output strategy file.

In order to represent strategies generated by Uppaal Stratego inside the program, the **UppaalStrategy** module was introduced. This module provides an interface for interpreting the strategy file generated by Stratego. By reading the transition rewards from the file we are able to translate the output file into a series of objectives usable by the player agent.

Now that the abstractions for both model and strategies are in place, we can start generating strategies for use in RoboCup. The function for this, used by both the player agents and the online coach, can be seen in code 1.

```
1  Generate_Strategy(game_state, model_name)
2    model ← UppaalModel(model_name)
3
4    model_data ← model_modifier(game_state,
       model)
5
6    Execute_verifyta(model)
7
8    strategy ← UppaalStrategy(model_name)
9
10   return parse_strategy(strategy,
       model_data)
```

Code 1: Generating a strategy from a game state

The *Generate_Strategy* function takes as argument the game state as well as which model to be used for the strategy generation. The model is, as mentioned previously, decided based on the game state. First an object of the UppaalModel class is generated with default values using the model name. In order to make the model fit the current game state exactly, the *model_modifier* function is called with the game state and default model as arguments. This function returns some model data needed later when parsing the strategy. This could for example include which players are represented in the modified model.

Now the Uppaal verifier, called verifyta, can be executed using the model and the accompanying queries for the model. When the verifier is done, it has created a new file with the strategy. This strategy is then read and abstracted upon using the UppaalStrategy module. The parse strategy returns the result in the form of a series of objectives. An important note about this function, is that the *model_modifier* as well as the *parse_strategy* functions are specific to the model being used.

4.4.1 Pass-Chain Model

A player is unable to kick the ball hard enough, that the ball crosses the entire field. To get a goal they must either dribble or pass the ball to each other multiple times, while avoiding interception by the opposing team. In order to decide a sequence of players to pass through, the pass-chain model was introduced. This model is used by the online coach, which observes the game state, and communicates the strategy via a *Say* message to the players. The model is used to generate a list of players, and the coach then tells these players one by one who the next player to pass to is. The last player in the list is instructed to dribble the ball forward.

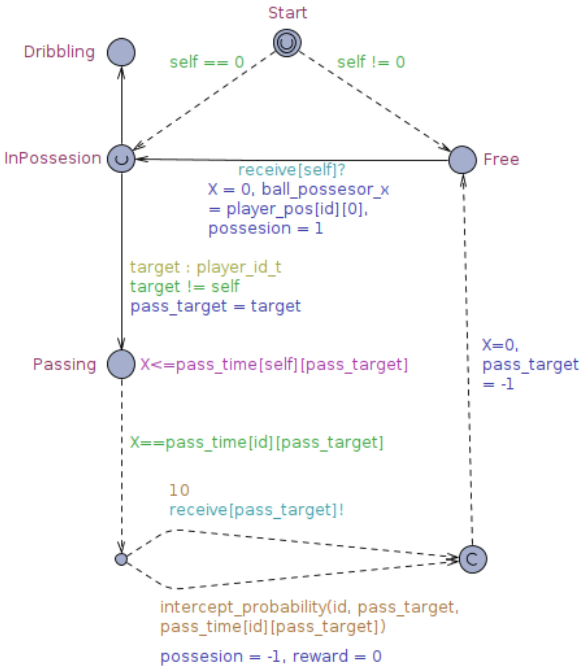


Figure 3: Player template of the pass-chain model

The model consists of a series of team players, instantiated from the template seen in figure 3, and a ball for measuring reward shown in figure 4. Opponents are represented as a list of coordinates, corresponding to their positions. Upon instantiation each player is assigned a unique id (named 'self' in the model). A player may either be in possession of the ball or free. From the possession state, the player may either choose to dribble or pass the ball to a teammate. When passing the ball to a teammate there is a probability that the ball is lost to the opponent. This probability value is estimated based on the position of the pass target and the opponents.

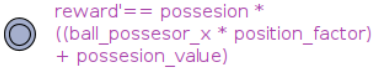


Figure 4: Ball/Reward template of the pass-chain model

The ball models the reward of the system, measured as the value of a clock variable. The rate of this clock is is determined by how far the ball is currently positioned from the opponents goal, and by which team is in possession of the ball.

The query for generating this strategy can be seen in code 2. It aims to maximise the reward within a time limit of 10 time units, corresponding to 100 ticks or 10 simulated seconds.

```
1 strategy BestPasses ← maxE(reward) [<=10]:
      <> T = 10
```

Code 2: Pass-chain model query

4.4.2 Possession Model

When a player is in possession of the ball, it must decide whether to pass the ball to a teammate or to dribble forward using a series of low power kicks. To estimate the effectiveness of these choices, we created the possession model. This model is highly dependant on a large number of uncontrollable and unpredictable variables such as the current positions of the opponents. For this reason it is infeasible to precompute strategies for all possible scenarios, which makes online strategy generation an ideal match for this model.

Simply generating the strategy when it is needed causes problems, since it often involves computation time spanning one or more server ticks. This is sometimes enough for the opponents to take control of the ball. To combat this timing issue, player agents continuously estimate the time that they will come into contact with the ball. This is done in several scenarios, for example when chasing the ball, when attempting an interception or simply when the ball is moving towards the player. Strategy generation is then initiated a few ticks before the estimated contact time, making the strategy ready close to when it is needed.

Another timing issue arises from the use of online strategy generation, as the computation time can cause the strategy to be outdated by the time it is available. To mitigate this issue, we feed a forecasted version of the game state to the model. This forecasted game state is created by performing simple extrapolation on the position of all visible moving objects.

The possessor template of the model can be seen in figure 5. The teammate template of the model and the queries can be seen in appendix A. The model is relatively simple and serves primarily as an illustration of forecasted online strategy generation. The strategy file generated using this model contains estimated reward of performing passes to each of the visible teammates and the reward of performing a forward dribble. These rewards are based on the estimated probability that the action will succeed, as well as how far the ball will advance towards the opponents goal.

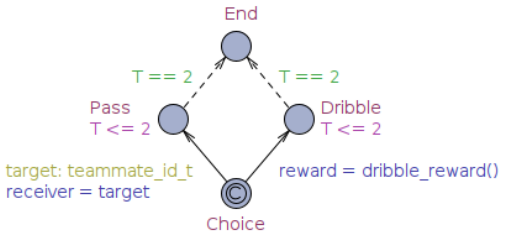


Figure 5: Possessor template from the possession model

4.4.3 Goalie Defence Model

The goalie is the last resort to prevent the opposing team from achieving a goal. To maximise the chance of a save, the goalie must stand in the right spot if an opposing striker is approaching while in possession of the ball. To strategise this, we made a model that is capable of creating a strategy for goalie positioning.

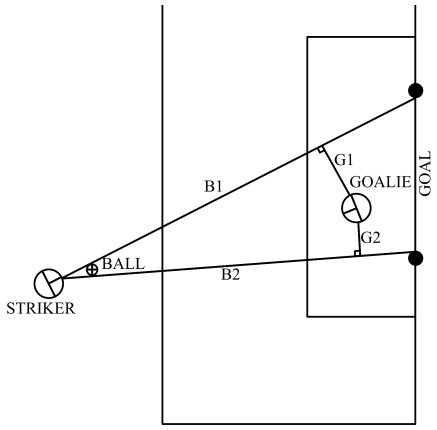


Figure 6: Goalie defender idea illustrated

The idea can be seen in figure 6. When the striker approaches the goal, we do not know in which direction he might move or shoot. We assume, though, that if the striker shoots, it will be within the cone to the goal. This can be seen in figure 6 as the cone between *B1* and *B2*. If the goalie was to intercept the ball in this cone, the shortest distance to the trajectory would be the perpendicular line from the goalie to the edge of the cone. This lines are illustrated as *G1* and *G1*. The objective of the goalie is to keep the longest of *G1* and *G1* as short as possible, since if one of these were long, the goalie would take longer to intercept the ball if kicked with this trajectory. The goal chance is then calculated as a function from the goalie position and striker position to the ability of the goalie to intercept the ball on its trajectory.

To model and strategise this idea in Uppaal, two timed automata running in parallel were created, one for the goalie and one for the striker. They are very similar, except that all transitions in the striker are uncontrollable. This models the fact, that the goalie does not know the intent

of the striker. Additionally, the striker and goalie have different terminal states. The striker has the *ball_kicked* state and the goalie the *end* state. The model for the striker can be seen in figure 7. The goalie model can be seen in appendix B together with the code from the model.

At each step the goalie and striker can choose to accelerate or decelerate in some direction modelled as a change in x and y velocity. After this the striker non-deterministically decides to either kick the ball or move further. If the striker moves, the goalie can then respond by either moving or standing still. When the striker eventually kicks, the goal chance is calculated as described previously. The best strategy is measured in terms of minimising the goal chance.

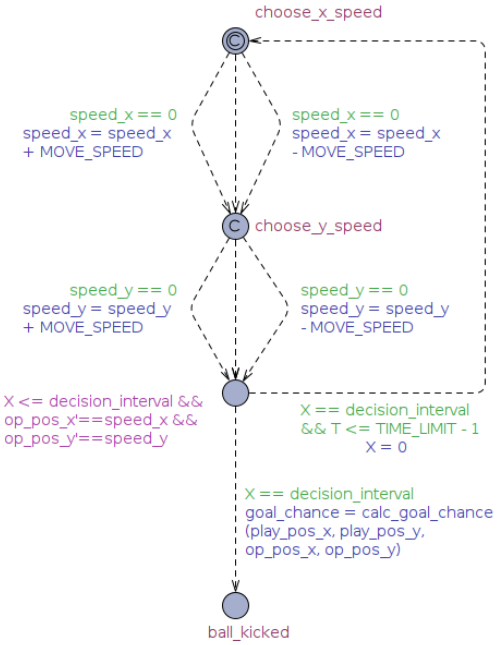


Figure 7: Striker part of static goalie model

The query used to synthesise the strategy can be seen in code 3. The query can be read as generating a strategy minimizing the goal chance within some time limit, set to 3 as default for this model, when the striker has kicked the ball.

```
1 strategy GoaliePositioning ← minE(
      goal_chance) [T<=TIME_LIMIT]: <>
      striker.ball_kicked
```

Code 3: Uppaal query for generating a strategy from the defence goalie model

Since the decisions made by the goalie requires an instant reaction and the possibles states are limited, this strategy was made as an offline strategy. Implementing and using the defence model as an offline strategy was done using a lookup table. The penalty area was divided into a grid of 1x1 meter squares. We then generated a strategy for all combinations of goalie positions and striker positions within this grid. All the combinations were then saved in a dictionary, with the key being the positions of the striker and goalie and the value being movement instructions for the goalie. The Python dictionary class is a hash map, which enables constant time lookup, allowing the goalie to apply the strategy the same tick that it receives visual information about the striker.

4.4.4 Stamina Model

If players dash as fast as they can throughout a game, they will run out of stamina quickly. As previously mentioned the player's stamina is a value between 0 and 8000. Since a player can dash with 100 power, meaning spending 100 stamina, every tick, 8000 stamina can be spent within 115 server ticks, i.e. about 12 seconds, even when subtracting the recovery of 30 stamina per tick from the stamina spent. If the stamina drops below 1000, the effect of all actions are reduced, making a stamina conservation strategy desirable. We introduce a model for online stamina strategy generation. The strategy is generated every 11 seconds, or 110 ticks. The model, as viewed in Uppaal Stratego, can be seen in figure 8.

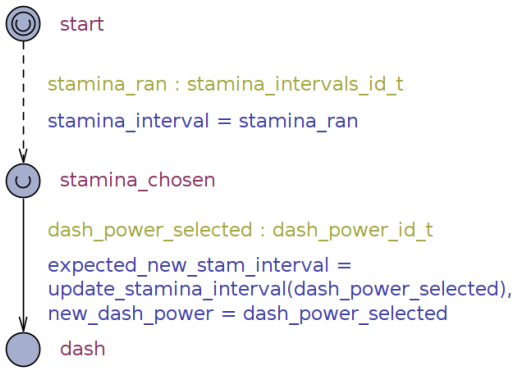


Figure 8: Stamina model

The model uses stamina intervals reduce the number of possible states. Since this model is meant as a online controller model used during games, the complexity and run time of generating strategies must be low to secure up to date strategies. The intervals chosen for this model is 0 to 8, where the interval is calculated by dividing the current stamina by 1000 and flooring the result, i.e. 1500 would be the interval 1. The model starts by non-deterministically choosing a random stamina interval. This leads to the stamina-chosen state. Here a dash power can be chosen as an integer value from 0 to 100. Depending on the dash power chosen, the *expected_new_stam_interval* variable is assigned a value corresponding to dashing with said dash power for the next 10 seconds. The code for calculating the expected new stamina can be seen in code 4. Whenever the player agent uses this stamina model the variable *dashes_last_strat* is assigned the number of dashes performed by that specific player agent since the last time, it generated a stamina strategy. This way, the model is updated to represent the state of the game before being utilised. The updating of the model is done using the interface described in section 4.4.

```
typedef const int[0, 8]
    stamina_intervals_id_t
typedef const int[0, 100] dash_power_id_t

stamina_interval
expected_new_stam_interval ← 9

const seconds_per_strategy ← 10
const recovery_per_sec ← 300

dashes_last_strat ← 42
new_dash_power ← 0

update_stamina_interval(dash_power){
    recovery ← recovery_per_sec *
    seconds_per_strategy
    consumption ← dashes_last_strat *
    dash_power
    delta ← recovery - consumption
    delta_interval ← fint(delta / 1000)
    if (delta_interval > 8){
        return 8
    }
    if (stamina_interval + delta_interval
     < 0){
        return 0
    }
    return stamina_interval +
    delta_interval
}
```

Code 4: Stamina model code

With the model updated to reflect the current state, a strategy can be generated using some predefined queries. These can be seen in code 5. Since the goal of the strategy is to conserve stamina, a **safe query** is introduced. This query says, that for all states it should be impossible, that the *expected_new_stam_interval* would be below 2, i.e. regardless of the strategy generated, it can never recommend a strategy which leads to stamina below 2000. Now a strategy can be generated for optimal dash power. Here we maximise the *new_dash_power* variable, i.e. the dash power selected, but doing it under the safe property defined previously. This provides a strategy maximising the dash power recommendation, while still staying above 2000 stamina.

```
strategy safe ← control: A[] not (
    expected_new_stam_interval < 2)
strategy opt_power ← maxE(new_dash_power)
     [<=1000]: <> player.dash under safe
```

Code 5: Stamina model queries

5 Data Collection

In order to test the performance of the strategies generated by our models, a module for data collection was developed. Every time the soccer server is closed, two log files are automatically generated by the RoboCup simulation. The first contains all actions received and executed by the server. The second log file contains a detailed view of the game state for every tick, the server ran for a particular game. Using these log files an entire game can be replicated. These provide a great foundation for analysing and gathering the data, that we wish to collect.

The data collection module is able to analyse the log files and generate the results for some metrics, that are not directly visible in the log files. This could for example be possession, see definition 4, which is not directly documented in the log files.

In order to test the performance of the strategies generated by Uppaal Stratego a script was developed, see code 6. This script takes an amount of ticks for the run as argument. It then generates some random scenario fitting the model in question, with some predefined constraints. This scenario is translated to some move commands, that the trainer can use to set up the scenario. Next the verifyta part of Uppaal Stratego is run with the scenario to generate the strategy, see section 2.2 for more information about verifyta. Now the server along with the clients, i.e. players, coach and trainer, are started. The trainer issues commands to set up the game according to the generated scenario. The coach communicates the strategy to the players. The game is now setup according to the generated scenario with the players aware of the strategy to follow. The game can now be started and run for the specified amount of ticks. Finally, the data collector module analyses the log and produces a file with the results.

```
Collect_data(ticks)
    setup_commands ← generate_scenario()

    strategy ← strategise_scenario(
    setup_commands)

    soccer_sim ← start_soccer_sim()

    trainer.setup_scenario(setup_commands
    )

    coach.send_strategy(strategy)

    soccer_sim.start_game()

    while soccer_sim.ticks < ticks
        continue

    soccer_sim.end_game()

    data_collector.analyse_log()
```

Code 6: Script for gathering data

We generate the scenarios with the same random seed to enable better performance comparisons of the different strategies using the exact same starting configurations. However, the built in nondeterminism of RoboCup affects the game in the form of the simulated physics and the noise in the sensory data received by the players. Furthermore the game is affected by the timing of the agents, which can vary from game to game based on multiple factors, such as how the scheduler prioritises agent threads.

5.1 Performance Metrics

As mentioned in section 5 the log includes data sufficient enough to completely replicate a game. In order to test our strategies, however, we need some metrics, that are not directly visible in the log. This section defines the extra metrics and explains how they are gathered.

Possession is an important metric for measuring the performance of the pass-chain model, see section 4.4.1. If the opposing team intercepts the ball shortly after starting the game, the model is considered to perform badly. Intercepting the ball would mean taking possession from the opposing team.

Definition 4. Possession: The first team to kick or collide with the ball is considered to be in possession. This team is now in possession until a player from the opposing team either collides with the ball, or kicks the ball.

Since a player can attempt to kick the ball, but miss, additional measures had to be made. In order to not count misses as kicks, we analyse both the actions log and the game state log to see, if the kick impacted the velocity or direction of the ball. If this was in fact the case, the kick must have affected the ball, hence the kick was successful.

In order to judge the performance of the pass-chain model, the field progress metric is introduced.

Definition 5. Field Progress is measured as meters progressed toward the center of the goal of the opposing team without losing possession of the ball. If possession is lost, the field progress is measured as the difference in distance to the opposing teams goal from the balls starting point to the position where a teammate last kicked or collided with the ball.

For performance evaluation of the stamina models two new metrics are introduced. One metric for gathering data about the minimum stamina and one for gathering averages. These can be seen in definition 6 and definition 7 respectively. This will be crucial for evaluating the stamina strategies, see section 4.4.4.

Definition 6. Stamina Minimum refers to the stamina of the team player with the lowest current stamina level.

Definition 7. Stamina Average refers to the average stamina of all players in a team.

In order to test how well the goalie is positioning, the goalie intervention metric is introduced, see definition 8. Initially, we thought that the amount of collisions, i.e. blocking the ball on its path to the goal, would suffice for measuring the performance of the goalie positioning. This, however, could lead to some imprecision, since the ball can, if kicked hard enough, pass through the goalie. Since this metric is made to test the positioning, but it is possible to be positioned perfectly and still not succeed in blocking, the collision approach cannot be used. As an alternative, we will test to see, if the goalie is within 1.2 meters of the ball, i.e close enough to catch the ball, at any point during the attack. The attack ends, when the ball passes behind the goal or after some predefined amount of server ticks.

The data used for testing if the goalie intervened is taken from the log, where the precision of the positions of objects on the field is perfect. While this metric can be used as an indication of the position performance of the goalie, it may not be precise. Depending on the implementation of the striker, one could imagine the striker missing the goal entirely while trying to shoot past the goalie, if the goalie is positioned perfectly, since the striker might be forced to kick at a steep angle. This would not be captured by the metric, since the goalie would then never be within catch distance of the ball. This can, however, be avoided by having the strikers shooting not depend on the goalies position. This will, however, compromise the realism of the test.

Definition 8. Goalie Intervention means, that the goalie was within 1.2 meters of the ball during a striker attack of some duration.

6 Experimental Results

In this section we try to analyse and interpret the data collected by the data collection module.

6.1 Goalie Defender Strategy

This section describes and analyses the results for experiments conducted using the strategies generated using the goalie defender model, see section 4.4.3. For this test the performance will be measured in terms of goalie interventions, see definition 8. To test how often the goalie manages to intervene in attacks, we generate many random striker attacks. The attacks start with the striker being at a random position on the y-axis, but always right at the edge of the penalty area on the x-axis. The ball is spawned right in front of the striker. This should represent a striker approaching the goal. After this the goalie is given 75 ticks to position according to the offline strategy for goalie positioning. This is allowed, since in a real game, the goalie would position gradually as the striker approaches the penalty area. After 75 ticks, both the goalie and the striker are allowed to proceed normally. The striker is designed to randomly either do one dribble to a random direction or shoot directly. The goalie is identical to our default goalie implementation, except in that its optimal position is pulled from the offline goalie defence strategy. This means, that it will try to intercept the ball, if it is inside the penalty area. Additionally, it will try to catch the ball, if the ball is in movement and within the goalies reach of 1.2 meters.

In order to compare the strategy to a baseline, an idle strategy is introduced. This strategy positions the goalie in the middle between the two posts of the goal regardless of the strikers position. Additionally, a manual approach

where the goalie follows the balls y-coordinate, bounded by the goal posts. If the ball is higher or lower than the goal posts, the goalie will position himself at the posts. For both comparison strategies, the behaviour of the goalie is identical to the goalie defender strategy with the exception of positioning.

In table 1 the results can be seen. It appears that the goalie defender strategy does help the goalie perform better in terms of interventions.

	Succesful interventions
Idle in front of goal	48
Goalie Defender Strategy	58
Manual Y-axis adjustment	60

Table 1: Succesful goalie interventions out of 100 random attacks

In order to test, if the results are significant we create the following hypothesis and test it with a significance level of 0.05. The binomial distribution is run with n=100, expected probability=0.48, and actual number successes 58.

Null Hypothesis 1. The goalie defender strategy does not perform better, than the idle strategy.

Using a binomial distribution, we get a p-value 0.029, which is significant under 0.05. We reject hypothesis 1, meaning the deviation in performance is very unlikely to be a product of chance. This leads us to the conclusion that the defender strategy performs statistically better than the idle strategy. Additionally, we want to know if the defender strategy performs differently to the manual y-axis adjustment strategy. For this, we propose the null hypothesis 2. We also test this using a binomial distribution with a significance level of 0.05. The test is run with n=100, expected probability of success 0.60, and the actual number of successes set to 58.

Null Hypothesis 2. The manual y-axis adjustment and goalie defender strategy perform identically given the same random attacks.

Expecting the outcome to be equal in a binomial distribution test yields a p-value of 0.38 for getting a result of 58 or lower. We cannot reject hypothesis 2. In this test, there is no significant difference between manually adjusting the goalie position according to the balls y-coordinate and using the goalie defence strategy generated by Uppaal.

6.2 Stamina Strategy

In order to test the performance of the strategies generated using the stamina model, see section 4.4.4, two tests are developed.

6.2.1 Beep test

As a test for running capacity, a beep test is introduced. The beep test is a real world fitness test for athletes, where the players have to run continuously back and forth between the two posts with less and less time to reach the next target (Wood, 2008). In our RoboCup version of the beep test the robots run from one side of the field to the opposite on the shortest side of the field, but without a time constraint. Our beep test performance is measured in laps, where a lap is 74 meters, or running from one side of the field to the other. We measure the performance against a base line, which includes a player simply dashing at full power all the time. This base line will obviously quickly run out of stamina, but will still be able to dash, but with decreased effect. The test is run for an entire game 4 times, meaning about 40 minutes in real time. The results are the averages of the 4 runs.

As seen in figure 9 the stamina strategy is at first a little behind, but around tick 2000 it catches up by running slightly faster. The fluctuation around tick 3000 are caused by the server replenishing the stamina of the players at half time. In the second half of the games, the stamina strategy slowly increases its lead. The stamina strategy achieved a higher number of laps in all 4 runs.

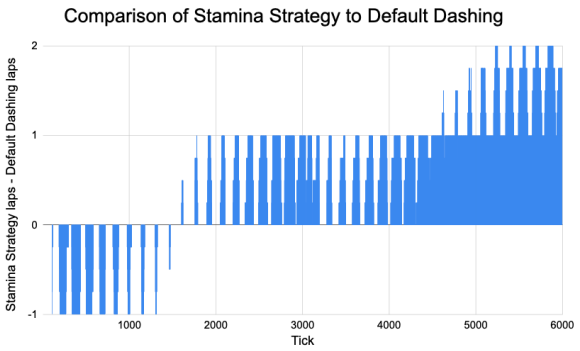


Figure 9: Lap comparison of stamina strategy against dashing at full speed

The stamina strategy increases the performance in terms of running capabilities measured in distance travelled given a time limit. Short term, however, it limits the capabilities slightly in trying to preserve stamina.

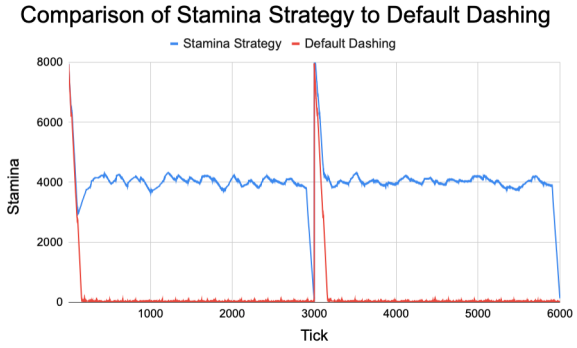


Figure 10: Stamina comparison of stamina strategy against dashing at full speed

In figure 10 the average stamina at each tick can be seen. The stamina strategy clearly keeps the player at a higher

level of stamina. The stamina stabilises at about 4000 stamina. The decrease in stamina of the stamina strategy just before half time and end game is due to the player being allowed to use the rest of his stamina, if the game is about to end.

6.2.2 Games

In addition to the beep tests, which tests the running capabilities of the players in an isolated fashion, we also want to test the stamina strategy in a full game. To test this two teams of identical implementations, except one team is using the stamina strategy, play against each other in 40 games. This test was included, since players realistically do not dash all the time, and this provides a more realistic test case.

The results of the games can be seen in table 2. In order to verify whether or not the difference is significant, we propose the following null hypothesis, see hypothesis 3, and test it using a significance level of 0.05.

Null Hypothesis 3. The stamina strategy does not increase the chance of winning.

In order to accept or dismiss the hypothesis we use a binomial distribution test. We use n=40, expect success probability of 0.5 wins and actual number of successes=28. This yields a p-value of 0.008853, which is clearly enough to dismiss the hypothesis. It appears that the stamina strategy does indeed increase the chance of winning when used, and with everything else being equal.

Team	Goals Scored	Wins
Stamina Strategy	247	28
Default Dashing	160	11

Table 2: Game results of 40 games with one team using the stamina strategy

In figure 11 the average stamina of the players on each team as an average of the 40 games can be seen. The metric is defined in definition 7. The results clearly show, that the stamina model increases the average stamina of the players. The players are able to preserve more stamina while still winning more games.

In figure 12 the stamina of the most tired player at every tick as an average of all 40 games can be seen. The metric is defined in definition 6. This graph shows an even bigger difference between the stamina strategy and dashing normally, than in figure 11. This indicates, that players run out of stamina more often without the stamina strategy, which would prevent them from playing well.

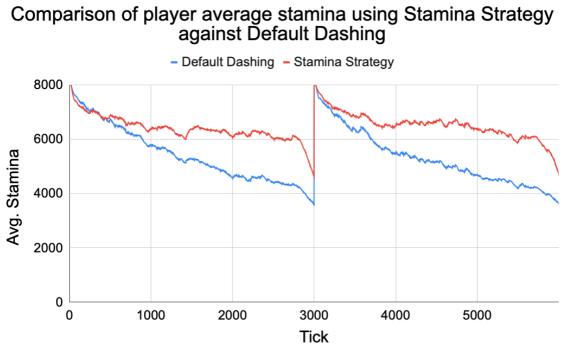


Figure 11: Player avg. stamina comparison for 40 games

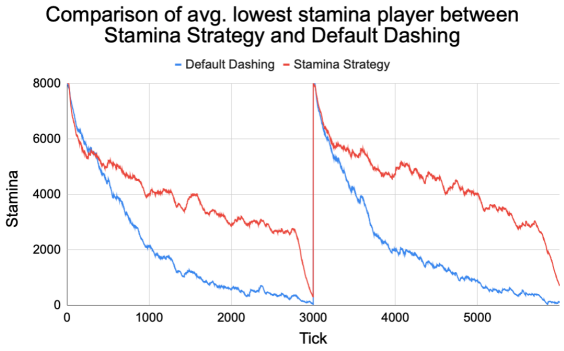


Figure 12: Average stamina of lowest stamina player in the 40 games

Another indicator that the stamina strategy does indeed make a difference becomes clear, when we look at the time of the goals. The goals of the team using the stamina strategy mostly fall in the latter half of each half time. As seen in figure 12 this is the time where some players without the stamina strategy are out of stamina. An extreme example of this effect can be seen in one of the 40 games, where the stamina strategy team won 1-11. Only 1 of the 11 goals scored by the winning team fell within the first third of the half times, i.e. when the opposing team had enough stamina to play well.

6.3 Possession Strategy

To test the possession strategy 50 full games were played with one team using the possession strategy, but otherwise identical implementations. We want to see, if the players strategising their possession perform better in games. The results can be seen in table 3.

Team	Goals Scored	Wins
Possession Strategy	167	28
Default	233	11

Table 3: Results of 50 games with possession model against default

The team with the possession model performs significantly worse than the default implementation. The poor performance could be a result of considering backwards passes in the strategy, which in certain situations appear to be a good idea, but often results in losing the ball closer towards your own the goal. Another explanation could be that the possession model team passes more and dribbles less, specifically the possession model team performed around 4 passes per dribble, where the default team performed around 3.6 passes per dribble. Since passes often fail and miss the target due to mechanical deficiencies in the player agent, it might be advantageous to prioritise dribbling more often. Lastly, it is possible that the forecasted world view is too inaccurate, and that this causes the player to generate strategies based on the wrong view of the world.

6.4 Pass-Chain Strategy

Two teams, one using the pass-chain strategy, and one using the default behaviour, were tested using random scenarios. Both teams were tested with the same random seed, and against opponents using default behaviour, which allows the opponents to try and intercept the ball. Each scenario has 5 players on each team, and the players get placed according to the generated random scenario. Each scenario is run for 100 ticks. The metric used is field progress, see definition 5. The field progress of each scenario is calculated as an average of 200 runs per strategy. As seen in table 4 the pass-chain strategy has a better average field progress.

Teams	Average Field Progress
Pass-Chain	17.81
Default	16.58

Table 4: Results of 200 games in average field progress

To test the results statistically, a Wilcoxon signed-rank test is used. We test the data against the null hypothesis 4.

Null Hypothesis 4. The pass-chain performance is identical to the default behaviour in terms of field progress.

The significance level of this test will be 0.05. The outcome of the test was a p-value of 0.52218, which is not $p < 0.05$, which means that we cannot reject our null hypothesis.

6.5 Computation time of online strategies

For the two online strategies used by the player agents, the possession model and the stamina model, we measured the amount of game ticks passed from starting strategy generation to having an applicable strategy. The results are measured during live games where all 22 players on the field use the strategy, so this can be considered a worst case scenario for performance. Around 1000 data points were gathered for each online strategy. The specifications of the test system can be seen in table 5.

CPU	Core i5-4460 @3.20Ghz 4c/4t
Operating System	Ubuntu 18.04 - 64 bit
Memory	16GB DDR3 1600mhz

Table 5: Test System Specifications

For the stamina model the median amount of ticks used was 7. The slowest computation took 20 ticks, and the fastest completed within 5 ticks. The range of 5-8 ticks constituted 95.5% of the cases. This relatively stable result is likely due to the fact that we offset each consecutive generation of the players stamina strategy by 10 ticks. Meaning player 1, 2 and 3 generate their strategies at ticks 10, 20 and 30 respectively, wrapping back to player 1 after player 11. As long as the computations takes at most 10 ticks, there will be no overlap between the players' calculations.

The possession model had a median computation span of 3 ticks. Around 73% took less than 4 ticks, while 97% took less than 5 ticks. This information is used when determining how early to start generation of the strategy, relative to when it is expected to be needed. The current implementation generates the strategy at least 4 ticks in advance, meaning we have to forecast the game state at least 4 ticks. Had we used an offset of 3 ticks instead of 4, we would have slightly more precision in the forecasted game state, but would also mean that the strategy would not be generated fast enough approximately 27% of the time.

6.6 Games Against Other Implementations

In order to test the performance of our implementation in a realistic scenario, we have tested it against other implementations. Since the stamina strategy and goalie defender strategies appear to improve performance, see section 6.1 and 6.2, we utilise these in the test against other implementation. These games were played like official games, meaning all rules were adhered to. For this reason the pass-chain strategy was not used, due to it requiring the coach message delay to be removed.

6.6.1 Keng

We found the Keng implementation on GitHub. It was developed as part of a course in machine learning at the Lafayette College in the United States. According to the authors, this implementation won the class tournament. Also according to the Keng authors, they employ a hybrid of reflex, utility and goal-based agents using swarm intelligence. The implementation is interesting, since Keng's player agents are developed manually and in the Python language just like ours. The implementation has, according to the author, some issues with position. These game were very one sided and all ended with a win for our implementation as seen in table 6. (Keng, 2015)

6.6.2 HfutEngine

To test our implementation against a team from the World Championship, we found the HfutEngine team. This team made it to the elimination round of the 2019 RoboCup 2D tournament (Lu et al., 2019). The player agents were developed using neural networks combined with Helios Base player agent code (Zhiwei-Le et al., 2015). Helios Base was developed by one of the creators of RoboCup as a free to use base agent, which allows teams to skip straight to the AI part of developing player agents. The Helios Base resources consist of about 250,000 lines of heavily optimised

C++ code (Akiyama, 2020). As seen in table 6 HfutEngine wins with a big lead against our RoboPaal implementation.

6.6.3 Validity of the results

Both of these comparisons are, however, not that meaningful in terms of comparing strategies. This is due to to the large differences in the mechanical abilities of the players, i.e. the ability to analyse sensory input and perform manuevers such as dribbles, intercepts and passes. The Keng implementation is created from scratch and the players turn and move significantly slower than the RoboPaal agents. Inversely, the HfutEngine players are much more adept than the RoboPaal agents. A more interesting comparison for another time would be to apply our generated strategies to the Helios Base player agents and then play a game against teams using the same library.

RoboPaal vs Keng	RoboPaal vs HfutEngine
19 - 0	0 - 34
21 - 0	0 - 34
21 - 0	0 - 36
16 - 0	0 - 36
15 - 0	0 - 35

Table 6: Results of games against other implementations

7 Conclusion

Generating strategies in real time for RoboCup players using timed automata modelled in Uppaal Stratego is possible. This can be done using both the game state of the online coach for generating a strategy involving more players and the noisy game state of the player agents themselves. Challenges like time-sensitivity can be mitigated using game state forecasting combined with asynchronous computing or offline strategy generation. In this paper we show, that strategies for goalie positioning and stamina conservation using Uppaal models can improve player performance, see section 6.1 and 6.2. The pass-chain model appears to perform better for moving the ball forward on the field, but we were unable to identify a statistically significant difference to our default implementation, see section 6.4. The pass-chain model does, however, only work if the message delay applied to the online coach is removed. The possession model was functional in producing strategies, but it was found to have decreased performance relative to the default implementation, see section 6.3.

8 Bibliographical References

Akiyama, H. (2010). Robo cup soccer simulation 2d league winning guide. ftp://ftp.iij.ad.jp/pub/sourceforge.jp/rctools/46021/RoboCup2DGuideBook-1.0.pdf.

Akiyama, H. (2020). Resourcese releases. https://osdn.net/projects/rctools/releases/.

David, A., Jensen, P. G., Larsen, K. G., Legay, A., Lime, D., Sørensen, M. G., and Taankvist, J. H. (2014). On time with minimal expected cost! In Franck Cassez et al., editors, *Automated Technology for Verification and Analysis*, volume 8837 of *Lecture Notes in Computer Science*, pages 129–145. Springer International Publishing.

David, A., Jensen, P. G., Larsen, K. G., Mikučionis, M., and Taankvist, J. H. (2015). Uppaal stratego. In Christel Baier et al., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 9035 of *Lecture Notes in Computer Science*, pages 206–211. Springer Berlin Heidelberg.

Eriksen, A. B., Huang, C., Kildebogaard, J., Lahrmann, H., Larsen, K. G., Muñiz, M., and Taankvist, J. H. (2017). Uppaal stratego for intelligent traffic lights. http://people.cs.aau.dk/~muniz/stratego Traffic.pdf.

Keng, W. L. (2015). Cs 420, artificial intelligence, robot soccer project, lafayette college department of computer science. https://github.com/kengz/robocup-soccer/blob/master/report/paper.pdf.

Larsen, K. G., Mikučionis, M., Muñiz, M., Srba, J., and Taankvist, J. H. (2016). Online and compositional learning of controllers with application to floor heating. http://people.cs.aau.dk/~muniz/LMMST-TACAS-16.pdf.

Lu, K., Ma, J., Cai, Z., Wang, H., and Fang, B. (2019). Hfutengine simulation 2d team. http://archive.robocup.info/Soccer/Simulation/2D/binaries/RoboCup/2019/PreliminaryRound/HfutEngine_SS2D_RC2019_R1_BIN.tar.gz.

Robocup Federation. (2019). Rules of soccer simulation league 2d. https://archive.robocup.info/Soccer/Simulation/2D/rules/RC2019_SS2D_Rules.pdf.

Robocup Federation. (2020a). A brief history of robocup. https://www.robocup.org/a_brief_history_of_robocup.

Robocup Federation. (2020b). Robocup 2d simulation league. https://www.robocup.org/leagues/24.

Rodrigues, H., Akiyama, H., Zare, N., and Obst, O. (2020). The robocup soccer simulator. https://github.com/rcsoccersim.

The RoboCup Soccer Simulator Maintenance Committee. (2020). The robocup soccer simulator documentation. https://rcsoccersim.github.io/manual/.

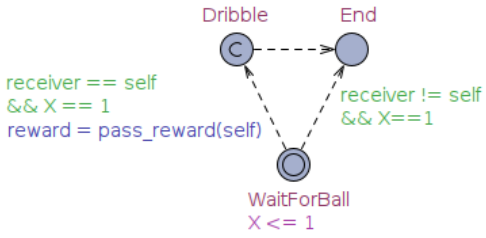
Uppaal. (2019). About uppaal. http://www.uppaal.org/about.

Wood, R. (2008). Beep test variations and modifications. https://www.topendsports.com/testing/beep-variations.htm.

Zhiwei-Le, Keting-LU, Wang, G., Hao-Wang, and Baofu-Fang. (2015). Hfutengine2015 simulation 2d team description paper. http://robocup2015.oss-cn-shenzhen.aliyuncs.com/TeamDescriptionPapers/SoccerSimulation/Soccer2D/RoboCup_Symposium_2015_submission_83.pdf.

Appendix A

Teammate template from the Possession Model

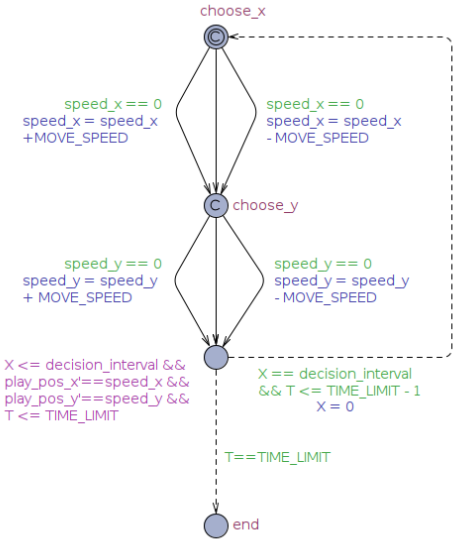


Possession Model Query

```
1 strategy DribbleOrPass ← maxE(reward) [T⇐2]: <> Possessor.End
```

Appendix B

Goalie template from the Goalie Defence Model



Goalie Defence Model Code

```
1  const int decision_interval ← 1;
2  const int TIME_LIMIT ← 3;
3  const int MOVE_SPEED ← 1;
4  const double BALL_SPEED ← 3.0;
5  const double GOAL_X ← 52.5;
6  clock op_pos_x ← 49.5;
7  clock op_pos_y ← 19.5;
8  clock play_pos_x ← 52.5;
9  clock play_pos_y ← 8.5;
10 clock T ← 0.0;
11 double goal_chance ← 0.0;
12 double goal_chance_targeted(double goalie_x, double goalie_y, double ox, double oy,
       double target_x, double target_y){
13     double b ← -(target_x - ox);
14     double a ← target_y - oy;
15     double c ← target_x*oy - target_y*ox;
16     double distance ← fabs((target_y - oy) * goalie_x - (target_x - ox) * goalie_y +
       target_x * oy - target_y * ox) / sqrt(pow(target_y - oy, 2)+pow((target_x-ox),2));
17     double ball_travel_dist;
18     double ball_travel_time;
19     double goalie_travel_dist;
20     double goalie_travel_time;
21     double col_x ← (b * (b * goalie_x - a * goalie_y) - a*c) / (pow(a, 2) + pow(b, 2));
22     double col_y ← (a * (-b * goalie_x + a * goalie_y) - b*c) / (pow(a, 2) + pow(b, 2));
23     if (col_x > 52.5) {
24         col_x ← 52.5;
25         col_y ← (a*col_x + c) / (-b);
26     }
27     ball_travel_dist ← sqrt(pow(col_x - ox, 2) + pow(col_y - oy, 2));
28     ball_travel_time ← ball_travel_dist / BALL_SPEED;
29     goalie_travel_dist ← sqrt(pow(col_x - goalie_x, 2) + pow(goalie_y - col_y, 2));
30     goalie_travel_time ← goalie_travel_dist / MOVE_SPEED;
```

```
31   if(ball_travel_time = 0){
32     ball_travel_time ← 0.1;
33   }
34     return goalie_travel_time / ball_travel_time;
35 }
36 double calc_goal_chance(double goalie_x, double goalie_y, double ox, double oy){
37     const double target_1_x ← GOAL_X;
38     const double target_1_y ← 6.5;
39     const double target_2_x ← GOAL_X;
40     const double target_2_y ← -6.5;
41     double chance_1 ← goal_chance_targeted(goalie_x, goalie_y, ox, oy, target_1_x,
       target_1_y);
42     double chance_2 ← goal_chance_targeted(goalie_x, goalie_y, ox, oy, target_2_x,
       target_2_y);
43     if (ox > 52.5) {
44         return 0;
45     }
46     if (chance_1 ≻← chance_2){
47         return chance_1 * chance_1;
48     }
49     return chance_2 * chance_2;
50 }
```