Pontifícia Universidade Católica
do Rio de Janeiro

Waldecir Vicente Faria

D-Engine: a framework for the random execution of plans in agent-based models

DISSERTAÇÃO DE MESTRADO

Dissertation presented to the Programa de Pós-Graduação em Informática of the Departamento de Informática, PUC–Rio as partial fulfillment of the requirements for the degree of Mestre em Informática

Advisor : Prof. Hélio Côrtes Vieira Lopes
Co–Advisor: Prof. Bruno Feijó

Rio de Janeiro
July 2015

Pontifícia Universidade Católica
do Rio de Janeiro

Waldecir Vicente Faria

D-Engine: a framework for the random execution of plans in agent-based models

Dissertation presented to the Programa de Pós-Graduação em Informática of the Departamento de Informática do Centro Técnico Científico da PUC-Rio, as partial fulfillment of the requirements for the degree of Mestre.

Prof. Hélio Côrtes Vieira Lopes
Advisor
Departamento de Informática — PUC–Rio

Prof. Bruno Feijó
Co–Advisor
Departamento de Informática — PUC–Rio

Prof. Marcus Vinicius Soledade Poggi de Aragão
Departamento de Informática — PUC-Rio

Prof. Simone Diniz Junqueira Barbosa
Departamento de Informática — PUC-Rio

Prof. José Eugenio Leal
Coordinator of the Centro Técnico Científico — PUC–Rio

Rio de Janeiro, July 10th, 2015

Waldecir Vicente Faria

Waldecir Vicente Faria obtained a bachelor's degree in Computer Science from the Pontifical Catholic University of Rio de Janeiro (PUC-Rio, Rio de Janeiro, Brazil). During his undergraduate years he held a scholarship from PROUNI and he worked with projects related to databases and human computer interaction. He also earned a scholarship from CNPq to do his masters at PUC-Rio.

Bibliographic data

Faria, Waldecir Vicente

D-Engine: a framework for the random execution of plans in agent-based models / Waldecir Vicente Faria ; advisor: Hélio Côrtes Vieira Lopes; co–advisor: Bruno Feijó. — 2015.
72 f. : il. ; 30 cm

Dissertação (Mestrado em Informática)-Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2015.
Inclui bibliografia

1. Informática – Teses. 2. Simulação Estocástica;. 3. Modelos Baseados em Agentes;. 4. Animação Procedural;. 5. Leilões.. I. Lopes, Hélio Côrtes Vieira Lopes. II. Feijó, Bruno. III. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. IV. Título.

CDD: 004

Acknowledgments

To my late uncle Benedito da Silva Polck for all the encouragement and patience while introducing me to the computers since when I was a kid.

To all teachers from Resende's social preparatory course for all the knowledge and support given.

To my family and friends for supporting me in all decisions that I took and helping me to solve all problems that happened until now.

To my cousin Ana Angelica and her husband Paulo Roberto for receiving me in their home and giving me the opportunity to receive higher education.

To my advisors Hélio and Bruno for the guidance, time and trust invested on me while creating this thesis.

To my friend André Mac Dowell for the support given in the initial design of the main algorithm from this thesis.

To PUC-Rio and CNPq for the financial support given.

Abstract

Faria, Waldecir Vicente; Lopes, Hélio Côrtes Vieira Lopes(Advisor); Feijó, Bruno(Co-Advisor). **D-Engine: a framework for the random execution of plans in agent-based models** . Rio de Janeiro, 2015. 72p. MSc Dissertation — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

An important question in agent-based systems is how to execute some planned action in a random way. The answer for this question is fundamental to keep the user's interest in some product, not just because it makes the experience less repetitive but also because it makes the product more realistic. This kind of action execution can be mainly applied on simulators, serious and entertainment games based on agent models. Sometimes the randomness can be reached by just generating random numbers. However, when creating a more complex product, it is recommended to use some statistical or stochastic knowledge to not ruin the product's consumption experience. In this work we try to give support to the creation of dynamic and interactive animation and story using an arbitrary model based on agents. Inspired on stochastic methods, we propose a new framework called D-Engine, which is able to create a random, but with a well-known expected behavior, set of timestamps describing the execution of an action in a discrete way following some specific rate. While these timestamps allow us to animate a story, an action or a scene, the mathematical results generated with our framework can be used to aid other applications such as result forecasting, nondeterministic planning, interactive media and storytelling. In this work we also present how to implement two different applications using our framework: a duel scenario and an interactive online auction website.

Keywords

Stochastic Simulation; Agent-Based Models; Procedural Animation; Auctions.

Resumo

Faria, Waldecir Vicente; Lopes, Hélio Côrtes Vieira Lopes(Orientador); Feijó, Bruno(Coorientador). **D-Engine: um framework para a execução aleatória de planos em modelos baseados em agentes**. Rio de Janeiro, 2015. 72p. Dissertação de Mestrado — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Uma questão importante em sistemas baseados em agentes é como executar uma ação planejada de uma maneira aleatória. Saber responder esta questão é fundamental para manter o interesse do usuário em um determinado produto, não apenas porque torna a experiência menos repetitiva, mas também porque a torna mais realista. Este tipo de execução de ações pode ser aplicado principalmente em simuladores, jogos sérios ou de entretenimento que se baseiam em modelos de agentes. Algumas vezes, a aleatoriedade pode ser obtida pela simples geração de números aleatórios. Porém, quando estamos criando um produto mais complexo, é recomendável usar algum conhecimento estatístico ou estocástico para não arruinar a experiência de consumo deste produto. Neste trabalho, nós damos suporte à criação de animações e histórias dinâmicas e interativas usando um modelo arbitrário baseado em agentes. Para isto, inspirado em métodos estocásticos, nós propomos um novo framework, chamado D-Engine, que é capaz de criar um conjunto de timestamps aleatórios, mas com um comportamento esperado bem conhecido, que descrevem a execução de ações em regime de tempo discreto e a uma determinada taxa. Ao mesmo tempo em que estes timestamps nos permitem animar uma história, uma ação ou uma cena, os resultados gerados com o nosso framework podem ser usados para auxiliar outras aplicações, tais como previsões de resultado, planejamento não determinístico, mídia interativa e criação de estórias. Nesta dissertação também mostramos como criar dois aplicativos diferentes usando o framework proposto: um cenário de duelo em um jogo e um site de leilões interativo.

Palavras–chave

Simulação Estocástica; Modelos Baseados em Agentes; Animação Procedural; Leilões.

Contents

1
Introduction

With the advance of the technology, interactive media is becoming each day more popular. In the present, we are able to produce interactive stories that does not follow a single storyline, having multiple endings and multiple paths. With that, a single product can be enjoyed several times and still keeping the consumer entertained. We say that products with this property have a high "replay factor" (23), a factor that increases the users' desire to "replay" the same content various times. Addicting or flexible gameplay, unlockable or secret contents, good soundtrack and plot variation itself are ways to increase the product's replay factor.

It is usual to see stories which can be presented in multiple ways or with distinct possible endings in digital games. The player can influence how the story will proceed in different ways such as: triggering events with key actions or activating events based on a player's profile, for example, the story would proceed to a bad ending if the player was behaving crudely in the game world context.

Moreover, there is a set of algorithms that focus on the automatic generation of story plots. These can be related to digital games, but they are not limited to only that kind of content. For example, there is also interest in applications that present dynamic stories like a common movie (29). These algorithms, commonly called artificial intelligence planners, are able to choose a sequence of actions based on a set of parameters and the desired goals to be achieved. Generally they focus on selecting which actions will be performed, but they do not detail how each action will be executed. Excluding sections from digital games, where the player can interact with the same scenario in different ways to achieve some objective, when a specific sequence of actions is given, usually it is presented in the same way, having pre-made animations representing each action.

We find a more simple example in Real-Time Strategy (RTS) Games (like Age of Empires[1]), where there are many actions and tools that influence the game progress throughout the war in which we are participating. However,

[1] http://www.ageofempires.com/

when we focus on the atomic actions that compose this complex flow of events, we notice that some of them are executed in cycles. A fight between two units is just each warrior repeating its static attack animation until being interrupted by something else.

1.1 Objective

Our objective is to increase the replay factor by adding variation and complexity to those atomic actions by introducing randomness on this kind of cyclic events. We break this cycle in multiple well-defined parts, transforming a **single action or state** into a sequence of **stages** that are executed over time. This way, we can establish how the action is executed in a discrete sequence of events. Each stage starts in a randomly generated instant and has a random time of execution before going to the next stage. While this approach may generate some limitations, it allows us to use decision theory techniques based on discrete event simulation (section 3.1).

Since we cannot represent many stories using just a single action as base, we also give support to an arbitrary planner that chooses what each agent should do to achieve some goal (section 2.2). Also, we create a simple system of message exchange between agents to add more power to the simulated model. One extra feature is the capacity of mixing this algorithm execution with the processing of user input to let the user interact with the simulated universe, and not just observe its execution (section 4.4).

1.2 Contribution

Our contribution consists of a new framework, named **D-Engine** (chapter 5), for agent-based models with randomized actions execution without demanding deep knowledge about stochastic simulation. With that, artists and game designers would take advantage of stochastic methods that can be applied to this kind of discrete simulation without having much background in mathematical areas.

This framework was devised to simulate environments where the actions can be described as a sequence of discrete events through time without losing relevant information. Also it supports multiple agents being able to exchange messages and to execute different actions simultaneously; however, a single agent can execute only one action per time.

As a proof of concept we present two applications. Firstly, following the inspiration from Real-Time Strategy Games, we present a simple agent-based

model that represents a battle between two warriors as a base to let new users know how it works and what they need to implement in order to create a new custom model.

The second application is the simulation of an e-commerce application based on English Auctions. The proposed model can have hundreds of agents with different states and it is used to show how to apply the framework in a more complex context, including user interaction with the simulated universe to let it compete with the agents through multiple auctions as a serious game.

1.3 Outline

This document is organized as follows. Chapter 2 discusses the related work. Chapter 3 describes an overview of the proposed framework and some basic concepts. Chapter 4 presents the details of the **D-Engine** framework. Chapter 5 shows the **D-Engine** software architecture. Chapter 6 and chapter 7 show an application of this framework to Real-Time Strategy Games and to English Auctions, respectively. Chapter 8 compares this proposed work with some solutions for similar problems or approaches. Finally, chapter 9 concludes this thesis and suggests ideas for future works that can be done in the proposed framework or using it as base for custom projects using agent-based models.

2
Related Work

This chapter describes some related work in the fields of procedural animation, artificial planning, stochastic processes and agent-based modeling.

2.1
Procedural Animation

If we know the exact stage in which an agent is, while it is executing some action, and we know its starting and finishing instants, then we can animate how this action is executed through time using an animation clip for each stage. This is a very lightweight and simple strategy for creating procedural animations.

There are other complex strategies, such as the crowd simulation used in the Lord of Rings movies (8), the animation based on data blending to simulate realistic character motion through a complex scenario (12), and the work from (18) based on game theory and machine learning to teach two agents how to act in a fighting game or in a tag game like a competitive game for example.

These projects produce smooth and realistic animations, some of them including physical interaction with the scenario. They are mainly focused on 3D models that manipulates the skeleton of the human characters to create real life-like motions. Our approach focuses in describing how an action is executed in an abstract way; we want to specify just what is happening in each point of time. Doing this, we can use our approach in any kind of story presentation: 2D animation, 3D animation, text description or audio description. Our approach may not offer all the realism details from these other complex techniques, but because of that, it can be used by simpler projects using less computational power and demanding less effort to implement.

2.2
Artificial Planning

The proposed framework's main objective is to describe how an arbitrary action should be executed; however, it is not focused on choosing which action should be executed in a specific context. To do that, our framework needs to

be compatible with some custom program able to plan which actions should be executed.

According to (19, Page 1), "planning is the subarea of Artificial Intelligence that studies the abstract and explicit deliberation process of choosing and organizing actions, by anticipating their expected outcomes, in order to achieve objectives".

Using classical planning with a well-defined set of states, enough knowledge about the simulated environment and a list of actions to change the system's state, it is possible to define a linear sequence of actions to satisfy a goal with a deterministic process.

However, this linearity may not look natural. In the real world, actions can have different results and stories can occur in different ways. In these cases, nondeterministic planning can be a good tool to improve the experience (19). There are planners that use the expected outcome from each task to improve the plan generation, like Markov Decision Processes (9) or Markov Games (18).

Also there are planners prepared to work with Knightian Uncertainty (24), when the probability of something happening is unknown or immeasurable. Examples of these planners are Hierarchical Task Networks (HTN) (27, 28), Model Checking (9) and Nondeterministic Finite State Machines (2).

Note that for any planning algorithm, choosing the parameters needed to do that efficiently can be a very hard task. While the system does not need to know everything about the simulated world, it must have enough information to take satisfying decisions (27). Even knowing what a good parameter can be, choosing how it should be presented to the planning system can be a challenge, for example we have simulated emotions (20), physics-based actions or long-term strategies and tactics (30).

2.3 Stochastic Processes

One important source of inspiration for this project was discrete event simulation, mainly the algorithms used for queue simulation. Queues are a common example in introductory courses about this specific kind of simulation, because they are simple to implement and it is easy to see how one can simulate their expected behavior by just knowing how to model the time when each event will occur and how much time it takes to accomplish some task.(25)

Instead of generating random times of client arrival on a line, for each agent doing a specific task, our approach generates the initial time of any stage from the basic animation cycle and its duration. In this way, we have a tool

to discretize the atomic tasks and we can also profit from all decision theory algorithms that exist to support this kind of simulation.

Stochastic processes are a vast knowledge area, and they have some methods that can be used in tasks like animation or planning. (18) is based on Markov Games to tell how each agent should react to achieve its best performance. Markov Decision Processes can be used to aid agents to plan their movements in fighting games, giving support to nondeterministic moves (10). However, if the represented model is complex, those methods can become very expensive in computational terms, requiring the use of some heuristics to make them usable (21).

2.4 Agent-based modelling

Agent-based models can be used to get insights from complex environments based on interaction between a massive number of agents, being widely used to observe phenomena from biology, economy, business and other areas where it seems to have some emergent behaviour, in other words, if there is some bigger pattern or result that is created by multiple interactions of smaller patterns or events (15).

Generally these agents are modeled in the simplest way possible to just represent their effects on the simulated universe, ignoring everything else. On the other hand there are models used in artificial intelligence projects focused on solving hard problems using multiple agents rather than just observing their behaviour. These models are popularly known as "multiagent models" and their agents can be much more complex and require synchronization or group work between multiple agents and machine learning to let the agents discover new ways of doing something (6).

Both models can give support to visual representation of their simulation; however, in this project we are more interested in visualizing agents performing actions than in the result of their actions. This does not mean that we do not care for the model's result, but we prefer to give more emphasis to the observation than to the problem solving feature of a model, so we use the word "agent-based model" for any model that will be simulated with our framework. Note that both areas share common techniques, terminology and methods and their names are used with the same meaning in some works.

Also to keep the research focus on the presented algorithm for action discretization and to keep the model design simple for artists, designers or any other kind of programmer that have interest in creating some project using our framework, we do not follow any formal specification or standards related to

agent systems, such as FIPA (Foundation for Intelligent Physical Agents)(14) or KQML (Knowledge Query and Manipulation Language)(3).

3
Framework and Basic Concepts Overview

In this work, we propose a new framework, named **D-Engine**, to randomize the execution of tasks within an agent-based model. Since each event has a random duration, it may generate random results. The main feature of this framework is that it is mathematically well grounded and it is able to support custom extensions to be used for different purposes. Also it should provide methods of stochastic simulation for inexperienced users, letting them estimate how the model will act and what kind of outcome it could generate.

Planners generate a sequence of compose actions and atomic actions. Composite actions are made of the union of multiple composite or atomic actions, and an atomic action is the indivisible unit for planner (see figure 3.1 as an example). Our algorithm's idea is to add variation to those atomic actions, while detailing how this sequence will be executed.

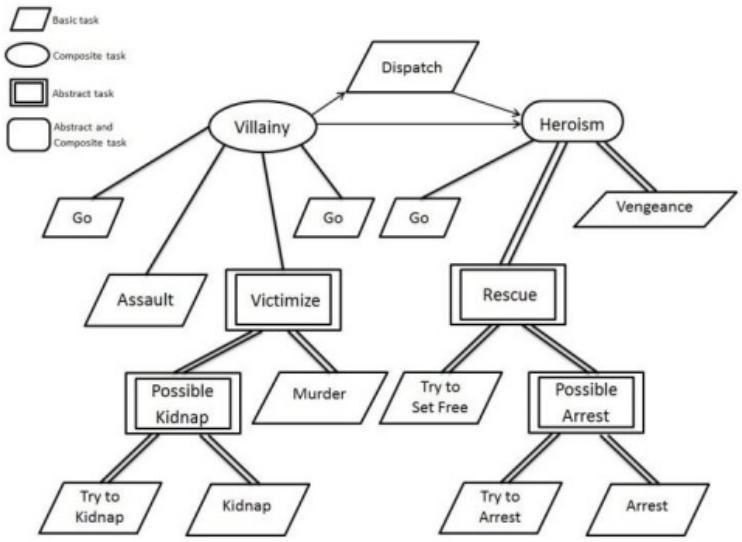

Figure 3.1: The set of possible actions that can be generated by a specific Hierarchical Task Network planner. Composite actions are made from a set of atomic or basic actions. Adapted from (28).

It is important to notice that our method alone cannot handle multiple actions. It needs the support of a planner compatible with nondeterministic actions, since our method cannot decide which action each agent will execute

in some specific context. Nondeterministic planners are also able to create alternative plans when a planned action does not generate the expected result. These planners are able to work with policies, which are total functions (π) that map model states (S) into actions (A) to achieve some goal (9):

$$\pi : S \rightarrow A.$$

An action A may not have a deterministic result but policies are made to handle this kind of problems. If possible in the presented context, the policy will move to an alternative state to recover itself from the failure and keep moving towards to the final goal. Policies may also select the best set of actions for each state, a set that increases the expected outcome of the plan execution over an arbitrary context.

Our method details the execution of each atomic action by dividing it in a group of stages executed in a loop. This cycle of stages is repeated until the action achieves the desired subgoal or it fails trying to achieve it. Each stage execution is defined by a simple timestamp (figure 3.2) with information such as which stage is being executed, which action is being performed, the initial time and its duration.

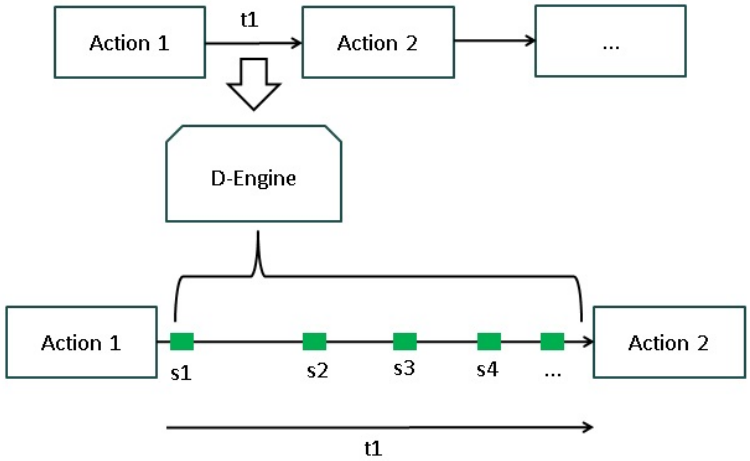

Figure 3.2: With our method we are able to describe an atomic action execution using a set of discrete stages indicated by multiple timestamps $s1, s2, s3, ...$ through time. Then action's total time $t1$ is the sum of the duration of every timestamp used to describe it.

In order to give a more concrete meaning to this division, we chose three common stages that we observed in simple animations like the attack one from Real-Time Strategy Games. Following this example, first we need to prepare to attack, then we execute the sword movement to attack, and we finish the movement going back to the basic fighting stance. Therefore we have three default stages: **Prepare**, **Do** and **Finish**. However, in real applications, a stage

could be interrupted by some reason. For example, consider an archer shooting an arrow in our agent. So we also create an extra stage that is executed when our agent is interrupted to let it recover itself from this halting. Properly, we call this stage **Recover**.

We use **Discrete Event Simulation** (section 3.1) concepts to create the events that represent these stages at a specified timestamp. Queue simulation (section 3.2) is, in fact, the main inspiration for the timestamp creation system. Consequently, we can use some decision theory processes in any model generated using this model. In special, this project provides **Statistical Analysis** (section 3.3) to let the user estimate the given model's outcome. Running this analysis multiple times, we are able to refine an arbitrary model to satisfy some custom constraints or say that they are probabilistically impossible to be reached.

Since the idea of this work is to create a suitable and flexible framework for this task, we provide hotspots to the user in order to implement the functions responsible to generate the plans, check goals, define the agents parameters and their behavior (chapter 5). We let agents exchange messages between them to create more complex models, and to not let the user modify the default discretization cycle (Prepare, Do, Finish). For example, using messages, the agent can interrupt other agents; this interruption is captured by the framework and forces the interrupted agent to go to the recovering stage.

Using this message system we provide the programmer with a method to intercept final user's inputs and let him interact with the system as if the user were an ordinary agent. Moreover, we abstract the task of processing the timestamps to present the agents behavior through time.

3.1 Discrete Event Simulation

While deep knowledge about this area is not necessary to use this framework, it is necessary to understand how this framework is implemented. Also, it can aid the user to create models faster and with more precision. This section briefly describes how this approach works, and it is based on the explanations of (25), where the reader can find more details about it.

Simulation is the act of trying to mimic some existing process or event through time in a device like a computer, a mechanic machine or a sheet of paper. Here, we are interested in stochastic processes simulation, which deals with models based on random variables indexed by time.

When a simulation tracks the behavior of a model continuously through time, we say that we are using a continuous simulation. This simulation is

needed in models in which we are interested in how their objects act through time based on some continuous function, such as a differential equation. Examples of its application are models based on physical processes, like rocket trajectories or rigid body simulations (11).

On the other hand, there are some real-life processes for which we do not identify a continuous function that describes how their objects works, or we just need to know some key points in time when important events happen. In these cases we can use discrete event simulation, which is an approach to track the behavior of a model based on a sequence of discrete events through time. This technique may not have the same precision of the continuous simulation, but in many occasions it is sufficient to study some processes. Also it is generally faster and lighter to be used when compared with the continuous option.

In this kind of simulation we are more interested to observe how some variables are affected when some events are triggered. The most basic variables needed are:

- The time variable t, which refers to the amount of simulated time;
- The system state variable SS, which refers to the state of the simulated process at the time t;
- The counter variable C, which refers to the amount of times that a specific event happened in the simulated process until the current time t.

To start a discrete simulation process, we initialize a list of events that will happen after the beginning. The simulation system will choose the event with the smallest starting time (the first event to happen) and the time variable t will be updated with this value, so our simulated time is the time when this event will happen. The simulation proceeds to create this event's duration and the effects generated by it. We assume that this event happened, and if it will chain the creation of some new event in the future, we insert this new event in the event list and restart this simulation cycle until some condition is satisfied to finish the simulation.

3.2 Queue Simulation

Single line queues are a good example to understand better how this kind of discrete event simulation works. Consider a single server where the time tA of

customers arriving to be served follows a nonhomogeneous Poisson Process [1] with rate $\Lambda(t), t \geq 0$. If this server is free, the customer proceeds to it and expends a random time t_D, following an arbitrary probability distribution G before its departure. Otherwise it goes to the end of a waiting line until the server becomes free again, giving us a classic first-in first-out system (figure 3.3).

If we are interested in knowing the average time a customer spends in this system, or when the server can go home after serving all customers, we can use discrete event simulation to estimate these values. To simulate this queue system, we need the following variables:

- The time variable t that represents to the amount of simulated time and increases each time we detected the ending of an event;
- The system state variable SS that is used to count the number of client arrivals and departures until the simulated time t;
- The counter variables C that are used to count the number of clients at the simulated time t in the system that are waiting in the line or being served.

The variables that we are interested to track are mainly affected by two events: customer arrivals and customer departures. So our event list can be composed of the combination of these two kinds of events, the arrival occurs on t_A and the departure happens on t_D.

The simulation proceeds with the following idea: The simulated system is initialized with the event of the first customer arrival on the event list. Then it chooses this event as the current event, increasing the number of customers on the line and changing the simulating time to $t = t_A$. If $t \leq T$ (T indicates the maximum time when customers can enter into the line) we also calculate the time t_A of the next customer arrival. If the server is idle, we generate a random number t_D indicating the time when the service will be finished and insert a new departure event from this customer on the event list. Otherwise, we just need to let the customer wait in the line until their turn arrives. The simulation keeps doing this cycle of choosing the next event and updating its variables according to the current event. At one moment t will be greater than T, then we block new customer arrivals and generate t_D, updating t with $t = t + t_D$ for each remaining client on the line until the number of customers in the system becomes zero. Finally the simulation is over.

[1] Basically a Poisson Process is a stochastic process used to count the number of events with independent inter-arrival times that would occur until some specific amount of time. When the event does not happen at a constant rate, we say that the process is nonhomogeneous. For more details see (25).

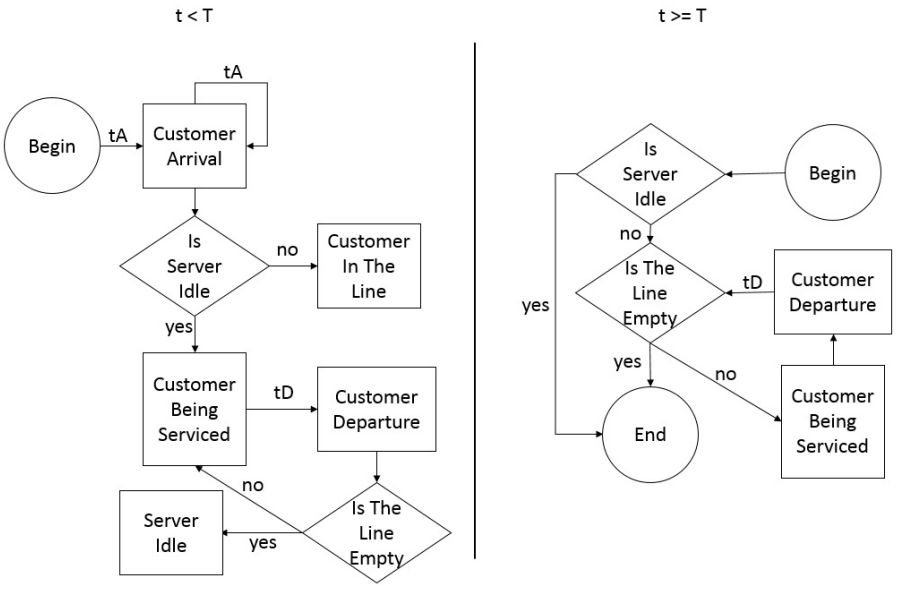

Figure 3.3: The sequence of actions used in a queue system. In the left we have the queue system working normally but when they go beyond some specific time T it will stop receiving new customers and start to service the present ones until the line becomes empty to let it close the establishment for the next day.

Using this simulation we can count how many clients were served and when the server would halt its operations by observing the variable T. Then we can play with the functions related to the generation of t_A and t_D and optimize the simulated model to see what kind of changes we need to do to make the real-life line work better.

3.3 Statistical Analysis of Simulation Results

When running a single simulation we have access to the result of a single run of an arbitrary model. To let us make decisions with confidence on the simulated result, we can analyze the average value θ generated from multiple simulations of the same model.

If we run k simulations and the simulation i generates the output variable X_i with expected value θ, we can consider that

$$\sum_{i \leq k} X_i / k = \overline{X}$$

is an estimator for our simulation output with an unknown average value θ, known as the **sample mean**. Since the mean calculation E is a linear operation

$$E[\overline{X}] = E[\sum_{i \leq k} X_i/k]$$

$$= \sum_{i \leq k} E[X_i]/k$$

$$= k\theta/k = \theta$$

we can conclude that the average value from the estimator is the same as the real value θ. Then we can consider it as an unbiased estimator.

The precision of the simulation statistical analysis increases when we increase the number of simulations performed. To measure this precision we can use the **sample variance** of X_i given in a similar way of the mean operation by

$$S_k^2 = \sum_{i \leq k} (X_i - \overline{X})^2/(k-1).$$

Given some degree of tolerance l, we can use this sample variance to know when the execution of more simulations does not change the sample mean $\overline{X}$ in a significant way.

With those formulas in hand we can use the following technique to analyze an arbitrary simulation model based on its statistics:

1. Run the simulation at least 100 times;
2. X $\leftarrow$ the sample mean of these simulated values;
3. S $\leftarrow$ the sample variance of these simulated values;
4. if a custom tolerance function using these values accepts the calculated sample mean, we stop and return $\overline{X}$;
5. Run the simulation again and go to (2).

For example, in this project our tolerance function is based on the normal distribution to approximate with $100(1-\alpha)$ percent that the distance between $\overline{X}$ and the unknown θ is less than an arbitrary l, the logical expression used is

$$2Z_{a/2}S_k/\sqrt{k} \leq l$$

where Z is the standard normal random variable (for example $Z_{0.25} = 1.96$). S_k is the standard deviation calculated from the sample variance S^2 and k is the number of simulations performed until the calculation moment.

4
Discretization Engine

Our focus is on discretizing atomic action in multiple internal stages to detail how an action is executed. These actions are executed by agents inside the simulated universe. We consider the **Simulation Universe** as the set of the following concepts:

- A set of agents;
- A set of states defining the actions that each agent can execute;
- A planner.

4.1
Agents and the Universe Updating Cycle

Unlike basic discrete simulation where we usually focus on performance, sacrificing modularity when creating a solution for a specific problem, in our framework we wanted to give support to an arbitrary discrete simulation system with programming flexibility but keeping the support to formal methods of statistical analysis. Instead of having few variables to modify, like in a single line queue simulation, we can have multiple agents with multiple variables. Also, different agents can generate a different set of states, and agents can execute multiple actions concurrently (but a single agent can execute only a single action at a time).

Instead of having a global event list to choose the next event to be executed like the list used in queue simulation, each agent has its own event list. In this way, at the end of the simulation, each agent will know which actions it has executed throughout a simulation. Since the agent action can generate multiple events before changing the current action, we consider that an agent is in a **state** of doing an action, rather than just saying that it did the action in our system.

This state tells the agent how it should proceed to fulfill some arbitrary action, so it should be able to tell how each stage should be executed, and how much time each stage could use. **Stages** are the smallest units which let the agents affect the simulated universe, and there are four types of stages based on the following ideas:

- "Prepare" : The agent prepares itself to do something;
- "Do" : The agent actually executes the action;
- "Finish" : The agent does anything that happens after executing the action here;
- "Recover" : If something interrupts the other stages, the agent may pass through this stage to try to recover itself.

Atomic actions may describe something more abstract than an individual movement. For example, the action of "fight" has a clear meaning, but its execution may be a repetition of moves like punches and kicks. The action of talking would be represented as a cycle of producing specific words with the mouth. So, our purpose is to break any atomic action according to these stages in a cyclic way.

Since we are discretizing the execution using these stages as keyframes or timestamps, we need a method to determine their duration and when each stage starts. We also need to know which stage should be executed to keep the logical flow of execution. Algorithm 1 can be used to do these tasks. Figure 4.1 shows a graphical representation of the update algorithm.

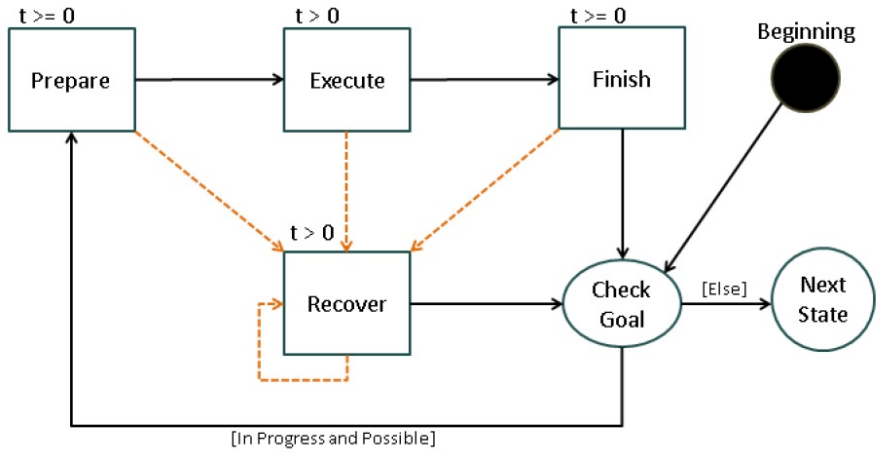

Figure 4.1: A graphical representation of the update algorithm integrated with the planner that it is responsible to check the goals and choose the next state.

It begins checking if someone has interrupted the execution flow in the line 3 from the *update* algorithm. If this is true, it proceeds to execute the recovering routine that involves calculating the execution time of the recovering stage with the *doRecover* function, disabling the interruption flag and turning on the check necessity flag. This flag is used to let the agent know when it needs to evaluate its current status and decide if it makes sense to stay in the current stage. Using the usual "fight" state as example, there is no sense to keep attacking when our opponent is knocked-out.

Algorithm 1 Calculate the duration of an arbitrary stage execution and apply its side effects to the simulated universe.

```
1:  function UPDATE()
2:      tRet ← −1
3:      if state.interrupted then
4:          tRet ← state.doRecover()
5:          state.interrupted ← false
6:          state.needCheck ← true
7:          if tRet < MIN_TIME then
8:              error('Invalid stage duration')
9:          end if
10:     end if
11:     while tRet < MIN_TIME do
12:         if state.stage = FINISH then
13:             tRet ← state.doFinish()
14:             state.stage ← PREPARE
15:             state.needCheck ← true
16:         else if state.stage = PREPARE then
17:             tRet ← state.doPrepare()
18:             state.stage ← DO
19:         else
20:             tRet ← state.doDo()
21:             state.stage ← FINISH
22:             if tRet < MIN_TIME then
23:                 error('Invalid stage duration')
24:             end if
25:         end if
26:     end while
27:     returntRet
28: end function
```

Otherwise it proceeds to the normal discretization flow in the line 11. This loop is used to mimic the cyclic sequence *Prepare* - *Do* - *Finish* - *Prepare* - *Do* .... For example, if the state is on the stage of preparing the action (line 16), it will calculate the necessary time to execute the preparation with the *doPrepare* function and set the next stage to be executed as "Do".

In our definition, we allow the state implementation to ignore the execution of the preparing and finishing stages; this happens when their calculated time is less than some specified time $MIN\_TIME > 0$. The loop condition will continue being valid and the next stage will be executed until any of them returns a value greater than MIN_TIME. The "do" and "recover" stages must return a positive time greater than this limit to avoid infinite loops, while simplifying this discretization loop.

We can only consider one stage execution finalized, when we start the next one. When we calculate the necessary time to execute the "Finish" stage,

we are doing that because the previous stage has been finished. This happens because any stage may be interrupted in the middle of its execution.

Finally, it is also relevant to know that all functions which calculate the estimated execution time from a stage (*doPrepare, doDo, doFinish* and *doRecover*) may modify the simulated universe variables. They can send messages to other agents and interrupt what they are doing. All these functions are planned to be abstract functions and they can be implemented in different ways, they just need to obey the time value restrictions described before.

4.2 D-Engine Algorithm

Section 4.1 describes how to discretize a single action using a single agent in a static state. This may be useful to animate single actions, but its use is very restricted. Algorithm 2 uses that idea to simulate an agent-based model with the help of an external planner responsible for the logic decisions:

The first thing that the algorithm does is to initialize all agents with their initial state to be executed and their other parameters. The "external planner" is some arbitrary planner compatible with nondeterministic actions adapted to do the following functions:

- Initialize the agents parameters and other values used to define the simulated universe;
- Decide if a simulation has reached its goal;
- Decide if some set of predicates is valid in the simulated universe;
- Decide which action each agent will execute.

We can consider that the planner controls the global logic, while each agent controls how the world is modified. Depending on these changes, the planner needs to change the global strategy to achieve the final goal.

After that, the D-Engine algorithm proceeds to use this planner and check whether the goal has been accomplished or it is possible to be accomplished using the *mayProceed(universe)* function in line 5 of algorithm 2. We say that the "universe" is the parameter, because it may need to verify any variable in the simulated universe to verify this affirmation. If it is not possible, we finish the algorithm returning a logical value showing if some extra constraint has been satisfied while it was trying to achieve the final goal.

Otherwise we need to choose which agent will start to execute its state. In line 10, we get the agent whose stage execution starting time is the smallest one. This means that this agent is the next agent to act.

Algorithm 2 Simulate an agent-based model with actions detailed in discrete points of the time.

```
1: function dEngine(t_0, delta_T, universe, externalPlanner)
2:     externalPlanner.initializeAllAgents(agentsArray)
3:     t ← t_0
4:     nIter ← 0
5:     while externalPlanner.mayProceed(universe) do
6:         nIter ← nIter + 1
7:         if MaxIterNum < nIter then
8:             error('Stage maximum number exceeded')
9:         end if
10:        currAgent ← agent with smallest tProx value
11:        t ← currAgent.getState().tProx
12:        if t < t_0 then
13:            error('Stage with an invalid initial time')
14:        end if
15:        if t ≥ t_0 + delta_T then
16:            return externalPlanner.checkConstraints(universe)
17:        end if
18:        if currAgent.getState().needCheck then
19:            currAgent.getState().needCheck ← false
20:            if currAgent.getState().needChange() then
21:                nextState ← externalPlanner.nextState(currAgent)
22:                nextState.tProx ← currAgent.getState().tProx
23:                currAgent.setState(nextState)
24:                currAgent.getState().needCheck ← false
25:                if currAgent.getState().needChange() then
26:                    error('A new stage should't be finished without executing any stage.')
27:                end if
28:            end if
29:        end if
30:        tNextEvent ← currAgent.update()
31:        currAgent.getState().tProx ← t + tNextEvent
32:    end while
33:    return externalPlanner.checkConstraints(universe)
34: end function
```

We check whether the selected agent's event initial time is valid in the lines 12 and 15 to secure that $t_0 \leq t < t_0 + delta_T$. t_0 is the simulation's initial time and $delta_T$ indicates the time span that will be simulated. An error is only raised if we try to run a stage that was supposed to start before the initial time. When this initial time is greater than $t_0 + delta_T$, we consider that the simulation was halted before executing the current stage. In section 4.4, we will detail how we can work with these values to get the real-time version of this algorithm. Consider $t_0 = 0$ and $delta_T$ as $+\infty$ to simulate the universe as long as it needs.

Line 18 is used to verify whether the agent needs to check if something relevant happened which raises the need to change its current state. Generally this happens when it finishes the local goal for the current state or it becomes impossible to do that.

If it is really necessary to change its state to achieve some goal, it will ask to the planner to give a suitable new state on line 21. Then it will update the new state's internal time, change the agent's current state and do an extra test on line 25 since every state must be tested on its first execution. An error is raised if the new state begins being considered as finished of failed, since it means that the planner chose the next state in a wrong way.

The algorithm proceeds to line 30 and runs a single stage of the discretization cycle. This line may affect the universe variables and it will return the needed time to finish this stage, using this value we calculate the next time the agent will act again.

Each interaction of this loop results in the creation of a new event timestamp. To avoid infinite loops, we also limit the number of timestamps that this algorithm can create. In the end, Algorithm 2 returns a logical value indicating if some custom predicate was satisfied in the simulated universe using the *checkConstraints* function.

4.3 D-Engine Result Statistical Analysis

Algorithm 2 returns the evaluation of some logic predicate about the universe state after the simulation of an arbitrary multiagent model on it. If we run this algorithm multiple times, we can calculate the expected probability of satisfying the given logic predicate. This value can be useful since we are dealing with a nondeterministic system.

To support this kind of analysis, we developed some functions compatible with our D-Engine algorithm to support discrete event simulation and statistical analysis, as described on section 3.3. Statistical Analysis can be

used directly in our framework by running the D-Engine multiple times. For each run, we get the returned value, calculate its sample mean and variance, and check if we can estimate the average value of return with some level of confidence.

Also, we have a function that runs the Statistical Analysis multiple times to refine the simulated model. "Refining" means that we tweak some universe variables to increase the probability of satisfying some arbitrary constraints defined by the planner. Algorithm 3 describes the refiner.

Algorithm 3 Checks if there is some degree of cheating where we can satisfy some constraint with some tolerance

```
1:  function SIMULATIONREFINER(desiredTolerance)
2:      cheat ← 0.0
3:      while cheat ≤ 1.0 do
4:          externalPlanner.modelRefine(cheat)
5:          simulationAvg ← DEngineStatAnalysis()
6:          if simulationAvg ≥ desiredTolerance then
7:              return cheat
8:          end if
9:          cheat ← cheat + refiningStep
10:     end while
11:     return −1.0
12: end function
```

We define the "cheat" parameter as a value from 0 to 100 percent (or 0.0 to 1.0). The planner must have some function which tweaks the simulated model taking into account this cheat value. After applying this cheat value in the line 4, it uses Statistical Analysis with D-Engine as the analyzed discrete system. It runs D-Engine at least 100 times to calculate its estimated average result. If the average value is greater than the desired tolerance, we finish the algorithm returning to the current cheat level. Otherwise, we increase this cheat value by some value smaller than 1.0. If the cheat level becomes greater than 1.0, we consider that it is impossible to achieve the desired tolerance, and it returns -1.0.

4.4 Presenting and Interacting with the Simulated Model

At the end of a simulation, each agent will have a set of timestamps indicating its behavior through the simulated time. Using these objects as key frames, we can present this simulation run to the user in any convenient way. However, if we run the simulation from its beginning until the end without stopping, we cannot consider external inputs such as the user interaction.

To solve this problem we can break the simulation time interval in several time fractions, and before simulating each fraction, we check whether some user has given some input to the system, in this case we intercept this input and convert it into some relevant change for the simulated universe. For example, after intercepting it we would do things such as:

- Convert the input into an internal message that can be sent to one or more agents;
- Modify the simulated universe directly based on the given input.

For that, we execute the following steps, per frame, until the simulation (starting with $t = 0$) is finished:

- Intercept and process any possible user or external input;
- Call the D-Engine algorithm with $t0 = t$ and $deltaT = frameLength$;
- $t \leftarrow t + frameLength$;
- Start to animate every timestamp with initial time smaller than t that has not been animated yet.

In this way, we simulate a very small interval of time per frame (generally 1/60 seconds) and the simulation universe generation progress as the presentation time increases. With this, we have the real time D-Engine algorithm where users can interact with agents and the simulated universe.

5
Framework Architecture Overview

Algorithms 1 and 2 describe the general idea behind the discretization engine but it does not detail how each part should be implemented. In this chapter we give an overview about one way to use those algorithms inside of a flexible framework compatible with a general game engine. In this specific implementation we focused on the following points:

- Facilitate model presentation: The framework should expose all data related with the model that represents how any action is executed;
- Facilitate model analysis: The framework should provide and abstract statistical analysis methods for any model created with it;
- Facilitate model design and creation: The framework should offer classes and methods that aid the user to think how to create and simulate an agent-based model;

Because of those points, the framework implementation sacrifices performance thinking in the ease and use and creation flexibility. It uses generic functions to be compatible with different applications, it is based on a generic Mediator design pattern (4) for the exchange of message between agents and its main core classes use the Dependency Injection design pattern (16) to let the user modify how these classes works without modifying their most internal and complex functions, just changing some external components.

Since we started to create this project thinking about applications related with storytelling, we refer to agents as "actors" that are part of some arbitrary story and the story creators need a tool to animate the story plot generated by some external planner.

5.1
Story Actors

A single actor represents any kind of agent inside our model (figure 5.1). It has an object that contains every parameter that describes the actor's information at some specific point of the time. So to create a new personalized actor, the user should not extend the base actor class, he should extend the

actor parameters object and attach it to a new instance of a base actor. This parameter object also describes how an actor receives any message from other agents or for external sources. If the message processing implementation was made on the state object, we would need to repeat some code sections relative to receiving similar messages between different states.

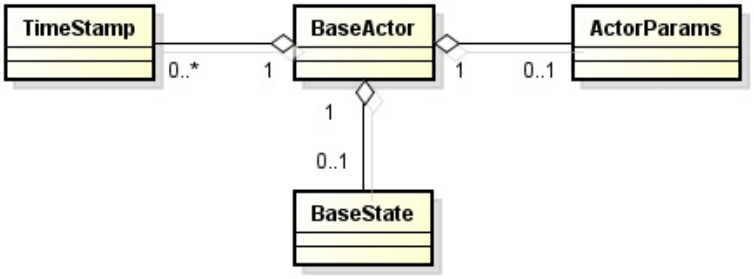

Figure 5.1: Simplified class diagram of the classes that represents a generic actor.

Its state object indicates how it should act to achieve some global or local objective, so they contain the functions used in the update function (algorithm 1). Another important method contained in the state object is the one responsible for checking whether the current state's goal has been achieved or it became impossible to do that using just that state.

While trying to achieve a goal it will generate a timestamp and save it for each stage that it executes, letting the actor with an internal history of the stages that it executed until the current simulated moment.

5.2 Simulation Core

The simulation core (figure 5.2) is formed by the components listed in section 4 plus a class that manages all these components based on the algorithms 1 and 2 and also on the statistical analysis methods from section 3.3. Each actor represents an agent and the set of actors with any relevant variable that characterize the current simulation forms the universe parameters. Using this universe object the user can access any variable or send individual or global messages to the simulated universe.

An external planner is used to control the global logic, implementing the necessary functionalities cited on section 4.2. It also includes the model refiner method used in the algorithm 3. Note that, in this framework, an actor can be its own planner if a model needs agents with more autonomy. The global planner just needs to ask the agent itself to choose which state it wants to execute. All these objects combined form the Simulated Universe.

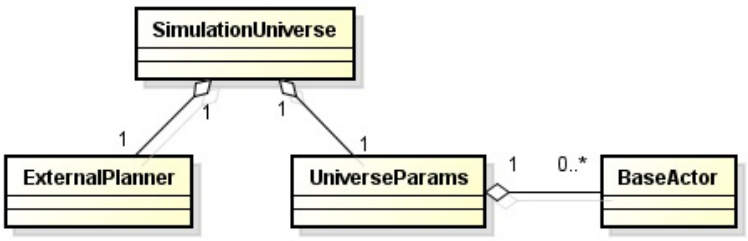

Figure 5.2: Simplified class diagram of the classes that constructs the simulated universe. The *BaseActor* class's components were hidden in this picture.

5.3 Implementation Tips

This framework works well with the bottom-up application design, where the user first models the most basic functions and the combination of those tools creates something more elaborated. In our case, the most basic elements are the actors themselves and the combination of their interactions says how the simulated universe will behave through time.

After defining which kind of application the user wants and which features he wants to provide, the user should choose the suitable actors that will be part of this universe. Each actor will have a set of actions that needs to be executed in a certain order to achieve some objective, so it will also need some adequate planner to control the action flow.

The user also needs to define how each action will be broken into different stages (Figure 5.3). The category name of each stage may help the user to divide it into several logical sections, but the user needs to be careful to choose when each stage's side effects should be applied to the simulated universe. A good idea is if stage **A** happens before stage **B** and stage **A** is responsible for changing something, you will only know if it was really executed correctly when it reaches stage **B** without being interrupted, so you should only apply stage **A**'s changes when stage **B** begins. See section 6.1 for a practical example of how to solve this kind of design problem.

To represent the effect from the stages executed by the actors, the user may need extra variables that define the global state of the simulated universe. A good strategy to create a new model in this framework can follow these steps:

- For each actor:
  - Implement and test its parameters;
  - Implement and test its states and their stages;
  - Implement and test the timestamps created by each stages;
- Add any necessary variable to the universe parameters and test them;

- Add the necessary features to let the planner control the actors and test them;
- Run the D-Engine using these objects as parameters;
- Analyze the results and execute this development cycle again if needed;

Following this order, it is also easier to test each module from the application. Since we use dependency injection, it is possible to create mock components to test other components if they need some specific element that is not ready or it needs to have a very specific behaviour for some test.

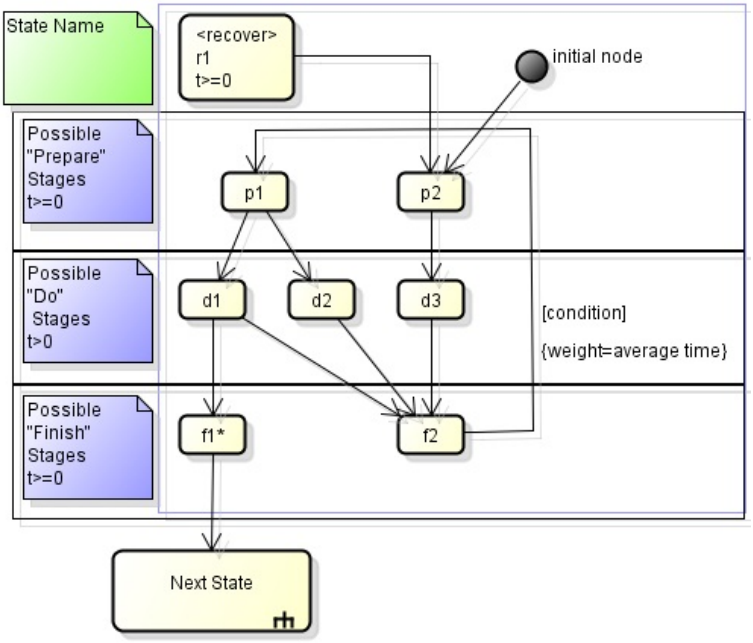

Figure 5.3: Some states can be broken into a complex combination of stages. Instead of going directly to code generation, it may be useful to start modeling them as a finite state machine with some extra informations like the "height" of the diagram indicating which stage is being represented, an asterisk indicating which stage changes some global variable or the conditions and the weight of each transition in seconds for each edge. See figure 7.3 for a practical example of this kind of diagram.

Unit tests are specially useful to experiment each state execution in the new model and how the planner should control the state transitions. While the combination of these events can generate a big number of possible ways of executing a state, if you guarantee the state transitions and the side effects generated by each one of them by using the unit tests, the number of tests needed are much inferior.

Finally the statistical analysis can simulate the same model thousands of times in seconds, being able to trigger some obscure errors that the user would not be able to find in some cases. Therefore it is also good to insert

some assertions in the execution of key stages to ensure some predicates that enforce the model correctness.

After developing the custom model, testing it and achieving a desired number of features, the user can generate the code responsible for the simulation graphic visualization and external input interaction. The framework has methods that expose the information needed to present the simulated model as explained in the section 4.4.

The external input can be done in two ways: Modifying the universe parameters directly or sending messages to the agents and letting them modify the simulated universe for you. The former is simpler to implement and useful when the user has the role of giving absolute commands into the agent-based model and the latter can be applied when the user is acting as an extra agent in the simulated universe.

6
Proof of Concept Application: Simple Duel

Since we are proposing a framework, we need to provide documentation and some application implemented on our framework to show that it is usable for agent-based model simulation compatible with the features described in the sections 4.3 and 4.4. This application can also serve as an example project to let interested users know how the framework works and how it should be used.

Consider the following problem: There is a digital scenario (figure 6.1) where two warriors, warrior 1 and warrior 2, need to fight. They cannot move, just keep attacking until one of them faints. We want to make this battle look more dynamic and realistic, so we have decided to insert more variation in it.

To solve this problem, we can use our framework to describe how this fight happens through time. Also, to show that the user can interact with the simulated system, when it presses a specific key, warrior 1 will try to dodge an attack.

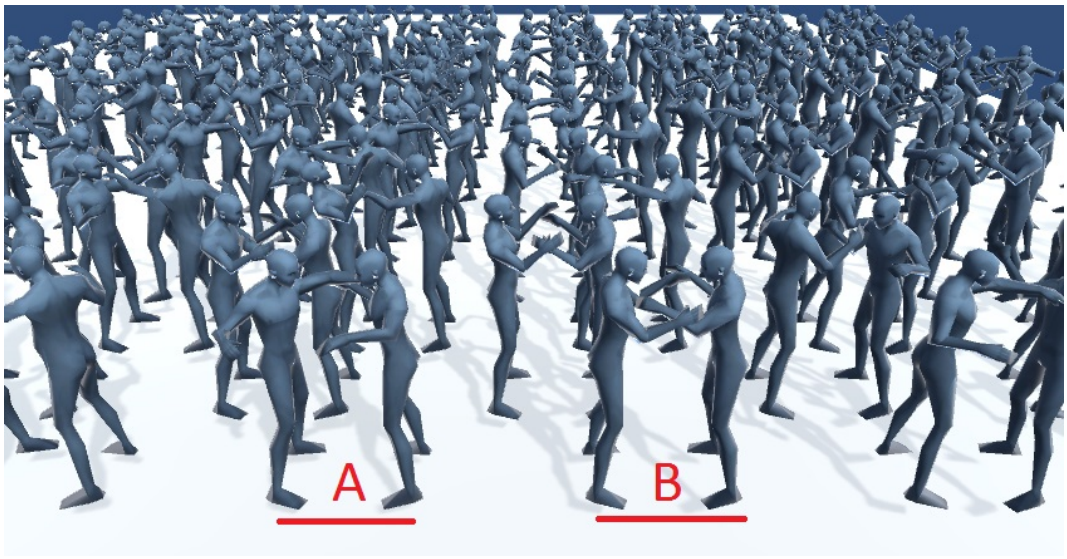

Figure 6.1: 120 duels presented using our framework with the same configurations of the machine cited in section 6.3. Each duel is an isolated duel from the others but they are presented simultaneously to give the impression of a war happening. A and B indicates two examples of different duels. Note how a different action is happening at each duel even though all of them were using the same initial parameters.

6.1 The Agents

In this model, we have only one kind of agent: Warriors. We can represent their life as hitpoints (HP) and add other variables like speed or dexterity to be the parameters that dictate how they should act as a warrior.

Since they just need to do one thing, they can have only a single state: Fight (Figure 6.2). This state must contain the functions that define the duration of each of the four stages that comprises our framework:

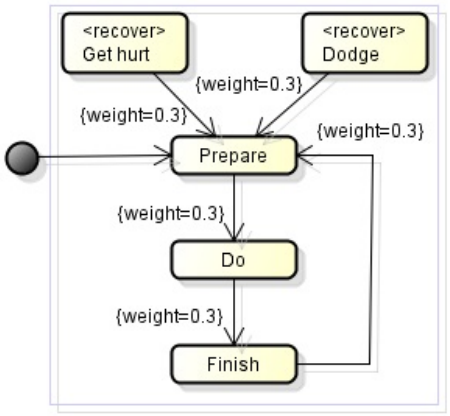

Figure 6.2: A diagram representing the possible stages from the fight state. It follows the basic update cycle from algorithm 1 plus two cases of recovering stages. See figure 5.3 for more details about the diagram.

- *doPrepare()* : Simulates the agent contracting its arm to throw a punch. It does not affect the simulated parameters. Returns $0.2+rexp(w.speed)$;
- *doDo()* : Simulates the agent executing the punch. It does not affect the simulated parameters. Returns $0.2 + rexp(w.speed)$;
- *doFinish()* : Simulates the agent moving its arm back after throwing a punch. A successful attack interrupts and damage the opponent with damage defined by $0.9 + rexp(w.speed)$. The attack fails if the opponent is executing the dodge state, and its progress is between 25% and 75%. Returns $0.2 + rexp(w.speed)$;
- *doRecover()* : called when w is damaged or when w starts to try to dodge an attack. Returns $0.2 + rexp(w.dexterity)$ in both cases.

Where w is the warrior object executing these functions and $rexp(\lambda)$ creates a random value following the exponential distribution, whose mean value is $1/\lambda$. If $w.speed = w.dexterity = 10$, the average duration of each stage will be 0.3, giving us attacks (figure 6.3) with an average duration of 0.9

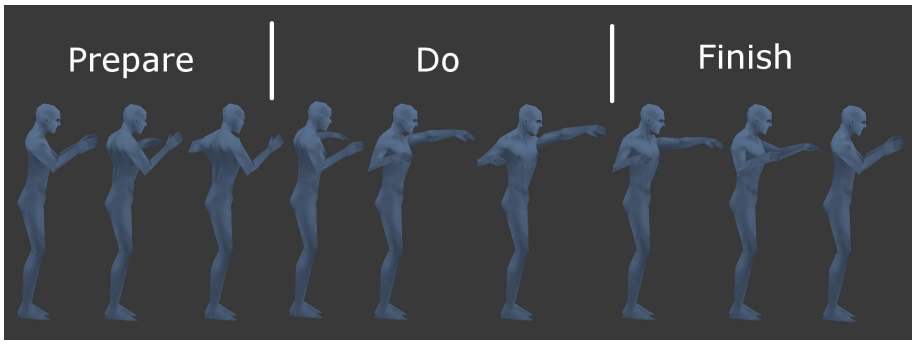

Figure 6.3: A sequence of frames showing how the attack main animation is broken into multiple stages. Each stage has a random time and this attack animation can be executed in loop until it is interrupted by some external agent. The interaction between the agents that makes the scene looks more dynamic instead of just an animation being executed repeatedly.

seconds. If $w.hp$ is 5 and we keep $w.speed = 10$, the attack damage will have an average value of 1.0, so a warrior would generally be defeated after 5 hits.

Warrior 1 inflicts the damage by sending a message to the opponent warrior. The opponent receives the message, reduces its own HP to represent this attack and marks itself as interrupted. Note that we only apply the damage on the *Finish* stage because only then that we are sure that the punch landed correctly.

To show a simple example of external input affecting the simulated universe, we let the user send an input to the warrior 1 ordering it to try to dodge some attack, so when warrior 1 receives a message with that command it will halt any stage and start to execute the dodge stage. This action is also interesting to see how a single event can impact the simulated model when involving two agents that were in a well-balanced conflict (Figure 6.4).

6.2 The Planner

For this purpose, we do not need any complex planner, our planner always return a plan with only one state to be executed. When it initializes the two warriors, it just sets their initial states to "Fight".

The planner is responsible to tell to D-Engine that it needs to run until one of the warrior's hitpoints becomes less than or equal to zero. It also defines the constraint predicate used when the D-Engine finishes the simulation. For example, we can ask if warrior 1 defeated warrior 2 when the simulation ends. So when we use this model as input, D-Engine will return *true* if warrior 1 defeated the warrior 2 and $false$ otherwise.

Considering this return predicate, we can use a custom refining function

Figure 6.4: Two pictures of the ending of a war scenario with 120 duels happening at the same time. If a pair of warriors is light green, then warrior 1 won, otherwise it is dark red and warrior 1 lost. In both pictures all warriors had the same parameters, but in the bottom one all warriors 1 received exact one interruption to try to dodge an attack, interrupting their current action. Note how the picture including a single user input changed the final result, making the warrior 1 lose in more duels.

in our planner to tweak the model and let warrior 1 be the winner more often. This little function is defined by Algorithm 4. With the cheat at the maximum value (1.0), these parameters from warrior 1 can be 50 percent greater than the untouched parameters from Warrior 2.

6.3 Results

The implementation of this example application was very useful to test the proposed framework and check whether it had enough expressive power to

Algorithm 4 Tweak the simulated model to support Warrior 1 before simulating the duel

```
1: function MODELREFINER(cheat)
2:     w1 ← the object that represents Warrior 1 in this model
3:     w1.speed ← w1.speed + w1.speed * 0.5 * cheat
4:     w1.dexterity ← w1.dexterity + w1.dexterity * 0.5 * cheat
5: end function
```

represent a simple project like this duel between two agents. As shown in picture 6.5, this project architecture can be used as the base for any new custom model planned to be simulated in this framework. Its unit tests are also interesting since they show how some framework functions works and how their returned values should be analyzed.

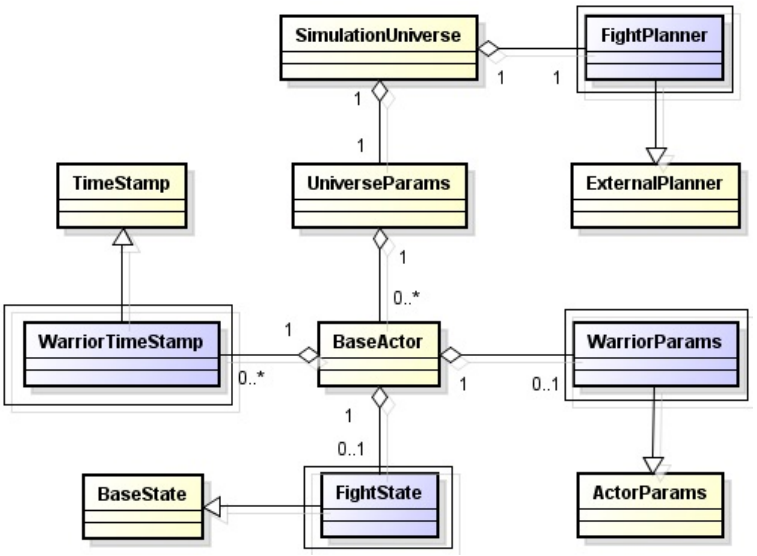

Figure 6.5: A simplified diagram class highlighting the custom classes responsible for representing this duel model inside of our framework with an extra square margin. Note that in this case we did not need to touch in the universe parameters since our model was very simple.

Using D-Engine combined with the Statistical Analysis with the refining step of 0.05 per interaction and $l = 0.1$ on a notebook with an Intel i7-2620M with 6GB of RAM, Windows 7 64Bits and Unity3d 5.0.1f1 Personal[1] in Editor Mode, we can estimate that if we have a cheat value of 0.60 (figure 6.6), Warrior 1 will be the winner in roughly 75 percent of all simulated fights. To achieve this result the simulation analysis spent almost 2.5 seconds to generate roughly 750000 events through 15000 different fights with a mean of 50 events per fight. Since that parameter difference between them is small, the fight will still be happening in a similar way as a fight without any cheat. An animator could

[1] www.unity3d.com

use this kind of analysis strategy to generate a scene that seems to be random but with a controlled result without letting the ending become obvious to the final application's viewers.

Finally, using the timestamps stored on each Warrior agent, we were able to generate a textual output describing each duel. This kind of output generator was so useful since it could describe a simulation without any extra implementation that it became a native function offered by the framework. After checking this output dump, the user can generate any other kind of model representation. In our case we used a Game Engine (Unity3D) to animate this fight using a 3D avatar to represent each warrior (Figure 6.1), as explained in section 4.4. We use fixed animation clips to present each stage, adjusting the clip's speed according to the stage duration.

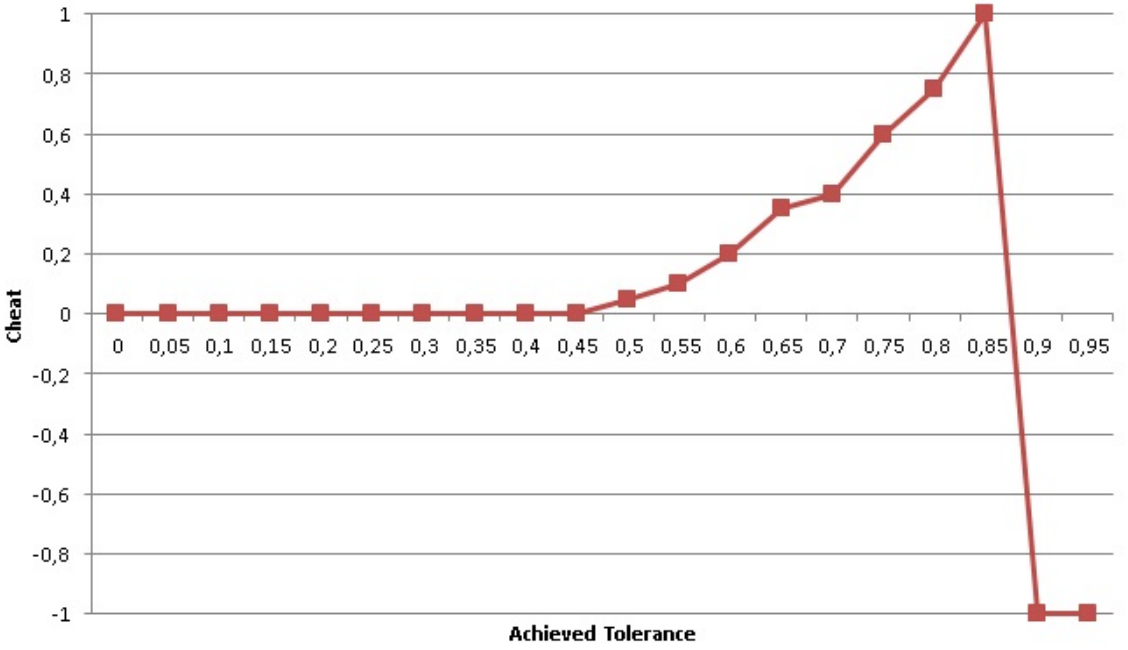

Figure 6.6: A graph created after running the statistical analysis with 20 different tolerance values between 0 and 100% using our framework. It shows the needed cheat to achieve a tolerance as a percentage about the number of times that warrior 1 will win. The cheat needed to ensure that the warrior will win more than 0 until 50% of the duels is almost zero since both warriors have the same parameters when the cheat value is zero. After this value, the necessary cheat will increase until the tolerance is 85%, when it becomes -1. This happens because, in the current model with the current refining function, it is impossible to guarantee that the warrior will have that success ratio.

Figure 6.7: Figure 6.4 shows a war result with no cheat. The top picture shows a war result with cheat value of 0.6 and the bottom one with 1.0. Note how the density of light green increases, indicating that the warrior 1 is winning more times.

7
Proof of Concept Application: Auction Site

When hearing about auctions, people usually think about the classic scene shown in movies where people raise their hands to show their interest to pay more than the last bid. Some people can also think about digital auctions like Ebay[1] where people can create their own auctions that are released on the Internet, which is one recent and popular application of auctions. However, this way of negotiation is still widely used in industry, such as telecommunication (17), agricultural goods (13) or applied in more abstract concepts like task allocation for robots (26).

Since an auction may deal with huge amounts of money, it would be interesting to know ways of increasing the outcome of this kind of negotiation. We can achieve it by reducing the probability of losing money, like frauds, or increasing the profit of the deal by using different strategies to aid the bidders and the auction owners (22). It is also an interesting way of analyzing human behavior, like the auction of a bill of twenty dollars[2] where in the end winner had to pay more than the value of the note (1). Because of that, auctions are an old target of academic studies, having much researches about it, in both theoretical and practical areas from diverse fields of study, such as informatics or economy (22).

Auctions can appear in different flavors: maybe the buyers are only interested in the price of the good to be bought (single dimensional) or we can have other extra interests like quantity and quality (multi dimensional). In some of them, we can access every bid (open cry) information or they can be kept secret until the end of an auction (sealed bid). According to these features we can classify an auction inside of a group with a common name like "Dutch" auctions or "Japanese" auctions or cluster then following a taxonomic classification (22). [3]

[1] http://www.ebay.com/

[2] There is one episode from the History Channel's television program "Brain Games" called "On the Set of Money" where you can see this history.

[3] Read more about this on the lectures notes called "A survey of auction types." from Stanford University CS206-Technical Foundations of Electronic Commerce, by Shoham and Wellman.

7.1 Discrete Simulation with Auctions

We use auctions as an example of the functionality of our framework because it has multiple people interacting at the same time in a situation of conflict. The auctioneer wants to get more money while the bidders want to get more products for a lower value, so there is conflict between the sellers and the buyers and conflict between the buyers themselves. This kind of environment is suitable for showing that our framework can simulate universes with multiple agents executing different actions and exchanging different messages at the same time.

In this work we are focused on the English ones, which are open cry, single dimensional, the winner bidder pays the value of the highest bid (first price) and the auctioneer has the job of deciding the winning bid (one-sided) or deciding to not sell the product when the highest bid is lower than a reserve price (sell-side). Because of those rules, one emergent behaviour in this kind of auctions is that bids have an ascending order, since it does not make sense to post a bid with a value lower than the actual winner one.

We are interested in this type of auction because it is one of the most well known kinds of auctions. It is simple and direct, pay most to win. With this it is easier to analyze the behaviour of the bidders inside of this context and mimic them as autonomous agents. Also it is easier for a human player to understand those rules and try to compete with others to get more products than them.

7.2 Description of the Simulated Universe

We will present the English auctions in the context of a digital market. Users can access one site where they can search auctions by a specific product. The result of this search shows a list of auctions with basic information such as the current highest bid and remaining time. The user can select one of these auctions to see more detailed information such as the list of past bids and it can bid some value greater than the current highest bid. When some auction is over, the winner is defined by the system and the product is distributed automatically if that bidder has enough money to pay the bid that it did (it is common to call this process of finishing an auction of "clearing"). The user does not need to wait the auction finish to go to other auctions and do bids there. In specific for this work, all auctions start at the same time and have a random duration. The universe simulation ends when the spent time on it exceeds some defined value $t_{max} > 0$ in the beginning of the simulation.

With that we must have two different agents: the bidders that mimics the human buyers and the auction manager that mimics the auction clearing system. Also we must say how the market is defined, what kind of products it has and how an auction is represented. We define the agent's bidding strategy based on the work of (7) which has as base the research made in (5). Since our market must be compatible with those agents, when possible we try to use the same letters that were used in (7) to define some basic concepts or sets from this chapter.

7.3 Simulated Market Definition

Our market contains information about the available bidders, products, auctions and its bids. Every simulation has always only one market m accessed by a set of bidders agents Ba that may be interested in buying one or more products from the set P. These products are made available by auctions from the set A, which contains all auctions that will be available before the simulation ends. The auctions that still active in time t are part of the set $L(t)$. For any auction $i \in A$, σ_i represents its start time and η_i represents its finish time, so $i \in L(t) \leftrightarrow \sigma_i \leq t \leq \eta_i$, $L(t) \subset A$.

Bidders can only bid in an auction i if $i \in L(t)$. If $B_i(t)$ is the set of bids in the auction i on the time t, then this bid must always be greater than any value from it on time t to be accepted as a valid bid. Lets also call the set of products offered by an auction i of $P_i \in P$ and say that the offered amount for each product $p_i \in P_i$ do not need to be an exact number, so auctions can offer fractions of products.

7.4 Bidding Agent's Bidding Strategy

An agent can consider a set C of constraints j to think about the value of its bid and it can give different weights $w_j(t)$ for each constraint in the time t. $\forall j \in C, 0 \leq w_j(t) \leq 1$ and $\sum_{j \in C} w_j(t) = 1$.

For any agent the maximum value $pr(t)$ that it would pay for a single unit of some product in some time t is called **private valuation**. If an agent wins an auction, its satisfaction is proportional with the difference of its private valuation with the final value which it paid. A good strategy to spend less money with that objective is to start with smaller bids and increase its value based in how the constraints C progress with the time.

In this project we use the same four constraints from the base research (7) as a set of tactics to achieve that strategy because it seems to be a good way to

simulate a human facing a marketplace: remaining time, remaining auctions, agent's wish to bargain and agent's despair level to buy. For each of these constraints j we have a function $f_j(t)$ that returns a value between zero and the current privation valuation at time t. Since we have multiple constrains, we can use a weighted sum to have the **current maximum bid** at time t by the equation 7-1:

$$M(t) = \sum_{j \in C} w_j(t) f_j(t) \tag{7-1}$$

$f_j(t)$ is an equation that gives a bid value based on the constraint j. Generally it is presented as $f_j(t) = \alpha_j pr(t)$ with α_j being the constraint influence with value between zero and one.

7.4.1 Polynomial Tactics Family

Every constraint will influence the current maximum bid with a polynomial equation 7-2 based on data related with the constraint j:

$$\alpha_j(t) = k_j + (1 - k_j) r_j^{1/\beta_j}, 0 \leq \alpha_j(t) \leq 1. \tag{7-2}$$

This type of equation can create an infinite range of time-based functions that can be used to model different kinds of agents with different curve shapes. Some of them can start bidding very low values while others can be very aggressive from the beginning. The main parameters from the equation 7-2 are:

- k_j: The initial weight of $\alpha_j(t)$, $0 \leq k_j \leq 1$;
- r_j: Usually it is the result of the division between two variables related to the constraint j, the current value of something and the maximum possible value of it, $0 \leq r_j \leq 1$;
- β_j: The importance of the ratio variance through the time, $\alpha_j(t)$ grows faster as how bigger beta is, $0 \leq \beta_j$.

7.4.2 Remaining Time Tactic

Using the equation 7-2 with r_{rt} as the ratio of the current time t with the maximum time t_{max} we can determine this tactic as:

$$f_{rt}(t) = \alpha_{rt}(t) pr(t), \tag{7-3}$$

where

$$\alpha_{rt}(t) = k_{rt} + (1 - k_{rt})(t/t_{max})^{1/\beta_{rt}}, 0 \leq \alpha_{rt}(t) \leq 1. \tag{7-4}$$

This tactic (figure 7.1) represents the time pressure over a buyer in an auction.

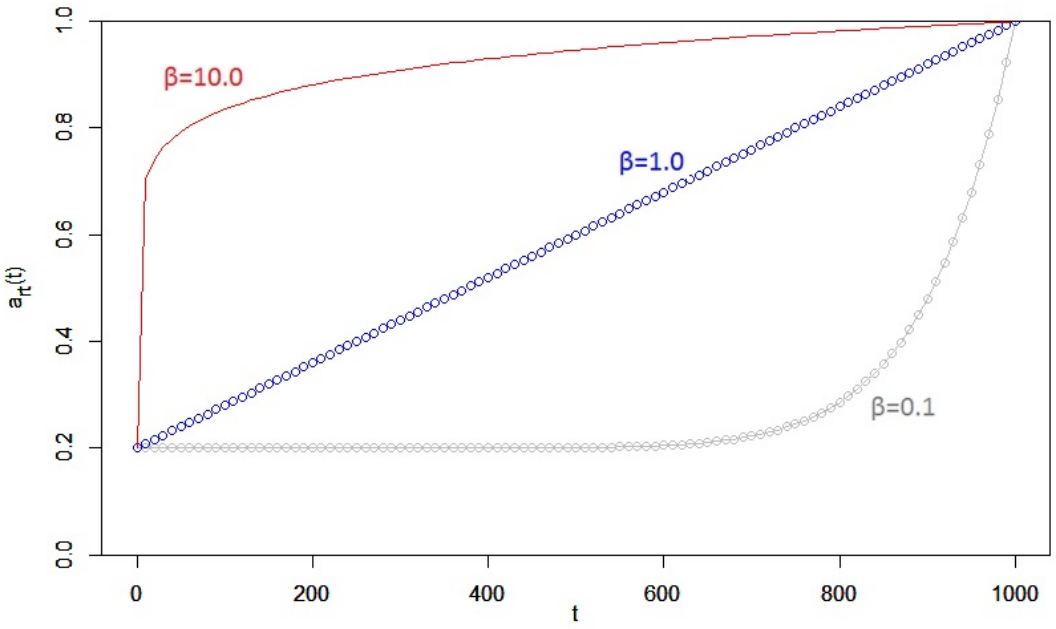

Figure 7.1: An example of how the remaining time tactic $\alpha_{rt}(t)$ works with different parameters. For this picture we used $k_{rt} = 0.2$ and the time going from 0 until 1000 seconds.

7.4.3 Remaining Auctions Tactic

Following the same logic of the remaining time tactic, if $ca(t)$ is the number of cleared auctions related to the target product and $ea(t)$ is the number of existent auctions related to the target product until t, we can simulate the pressure of having less options to get a good using the following equation:

$$f_{ra}(t) = \alpha_{ra}(t)pr(t), \tag{7-5}$$

where

$$\alpha_{ra}(t) = k_{ra} + (1 - k_{ra})(ca(t)/ea(t))^{1/\beta_{ra}}, 0 \leq \alpha_{ra}(t) \leq 1. \tag{7-6}$$

Note that differently from the original work where $r_{ra} = ca(t)/|A|$, in this project we use $ea(t)$ instead of $|A|$ because of some occasions where we do not know the total number of auctions that will exist in this universe until we finish the simulation. For example, more auctions would be created after the beginning of the simulation, so $|A|$ would not be considered a constant number.

7.4.4 Bargain Desire Tactic

This tactic is different from the others since it tries to control to increment that the user does when the time progress in the marketplace. For that it considers the progress of every unfinished auction and the value of the highest bid in each unfinished auction until t. Let $L_p(t) \subset L(t)$ be the subset of active auctions which sells the agent's target product p and consider the following equation:

$$f_{ba}(t) = \omega(t) + \alpha_{ba}(t)(pr(t) - \omega(t)), \tag{7-7}$$

with α_{ba} being similar to the formula 7-4:

$$\alpha_{ba}(t) = k_{ba} + (1 - k_{ba})(t/t_{max})^{1/\beta_{ba}}, 0 \leq \alpha_{ba}(t) \leq 1, \tag{7-8}$$

and $\omega(t)$ representing the minimum bid value that this agent would pay as the average value of the highest bid $v_j(t)$ from each auction $j \in L_p(t)$ weighted by the ratio of the auction progress

$$\omega(t) = \frac{1}{|L_p(t)|}\Big(\sum_{1 \leq i \leq |L_p(t)|} \frac{t - \sigma_i}{\eta_i - \sigma_i} v_i(t)\Big), 0 \leq \omega(t) \leq max(v_j(t), j \in L_p(t)). \tag{7-9}$$

Note that we can only ensure that $f_{ba}(t) \leq pr(t)$ if $\forall j \in L_p(t), v_j(t) \leq pr(t)$. While this calculation seems to be smart to an autonomous buyer agent which wants to spend less money, it does not seem to be something that a human would do, since it can consider dozens of auctions values to calculate $\omega(t)$. Maybe a modified $\omega(t)$ calculation considering the values which the agent observed while it was looking for a product would be more interesting for an agent trying to mimic a human buyer, but for this project we kept the calculation inspiration from the base work.

7.4.5 Despair Tactic

Since this tactic share the same interest of controlling the increase of the bid value through the time, it uses the same equations 7-7 and 7-8 to calculate the $f_{de}(t)$ and $\alpha_{de}(t)$. The difference is in the values of the other parameters k_{de} and β_{de}.

7.5 Modeling the bidder agent with D-Engine

In this project we wanted that the agents act like a human trying to buy something in an electronic marketplace based on auctions. For that propose

we must define what an agent needs to execute to access the auctions and how it makes decisions like which product it would buy, which auction it would look for that product and how much it would pay for that.

A human accessing a site can do lots of actions until he arrives in a specific section of it. Section 7.3 shows how an agent choose the value of every bid but does not detail how it will act until it bid in an auction. We use the D-Engine framework to simulate this set of actions as a non-deterministic finite state machine where each state represents one action that it will be discretized by the framework into multiple stages. Those states (figure 7.2) are explained with details in the sections 7.5.2, 7.5.3, 7.5.4 and 7.5.5.

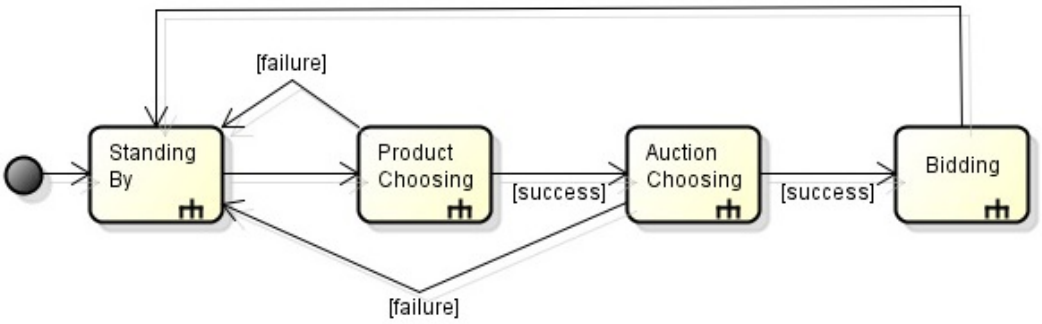

Figure 7.2: A diagram of the bidder agent's possible states. See the subsection 7.5.6 for more details.

7.5.1 Useful agent functions

These functions are used by the agents in different states to mimic how humans would execute some actions that have subjective parameters like choosing things or values. Some of them includes random values to make it look less artificial:

- *Calculating product utility* : Given a product $p \in P$, if the agent has access to the average value of p and it is greater than its amount of money, the utility is zero. Otherwise it is $0.1 + 0.9 * p.rarityFactor$. The "rarity factor" is a parameter to let an agent distinguish different kinds of products as if the products had different popularity or rarity in the simulated universe;
- *Selecting the target product* : Calculates the utility of every product $p \in P$ and for each result it adds until 100% of the original utility to generate randomness, then it selects the product with the maximum modified utility;

- *Selecting the target auction* : Given a product $p \in P$ as its target product, it will access the subset of active auctions $L_p(t) \subset L(t)$ that contains this product and choose the auction $i \in L_p(t)$ with the smallest remaining time. So our agents are worried to not lose time in auctions at its very beginning, since the result of these auction are harder to notice in that moment;
- *Calculate relative private valuation* : In this project the private valuation $pr(t)$ for a single unit of a product $p \in P$ is the amount of money that an agent has at the time t. If an auction $i \in A$ offers the product p, the relative private valuation is given by $min(pr(t), p_i.amount * pr(t))$;
- *Checking if the auction is interesting for bidding* : An auction $i \in A$ is not interesting at time t if it is finished, its current highest bid is higher than the agent's relative private valuation for that product or if the agent itself is the owner of that highest bid. If the agent has any other bid in this auction then the auction i has 30% of chance of being uninteresting, in other cases it will be interesting to the agent bid on it;
- $getHumanTime(x)$: Simulates a function able to calculate the time used by a human to do something using x as a parameter. In this project we use a simple calculation: $getHumanTime(x) = MIN\_TIME + rexp(x)$;
- *Calculate bid value* : See section 7.3, we use the relative private valuation instead of the usual private valuation here.

7.5.2 Standing by

When the user is away from the computer or it is not focused on the auction site we say that the agent is standing by since it is doing nothing useful until it focus its attention on the site again. This usually happens before the user arriving to use the site for the first time or after it doing some important decision like bidding on some auction and leaving from it.

Stages functions:

- *doPrepare()* : Does nothing. Returns 0.0;
- *doDo()* : Simulates the agent spending time doing some arbitrary action not related with auctions or thinking about buying something. Clear the agent's target product and auction. Returns $getHumanTime(1.0)$;
- *doFinish()* : Does nothing. Returns 0.0;
- *doRecover()* : This state is uninterruptible, so it does not have any recovering stage. Returns 0.0.

State status after a discretization cycle:

- If the agent is able to find any target product (see section 7.5.1), it considers the current state as concluded with success;
- Otherwise it executes the cycle again.

7.5.3 Choosing a product

In this moment the agent is looking the list of available products and wondering about which product it would want.

Stages functions:

- *doPrepare()* : The first time when this stage is executed it simulates the agent going to the section of products, returning $getHumanTime(1.0)$. If this function is called again in this state it will return 0.0 since the agent is already on the product section;
- *doDo()* : Simulates the agent spending time looking for some interesting product. Returns $getHumanTime(1.0)$;
- *doFinish()* : Simulate the agent deciding its target product for the moment or giving up to find products, setting the target product case it exists. Returns $getHumanTime(1.0)$;
- *doRecover()* : This state is interrupted only when the agent receives a notification that it won some auction with the product that it was looking for. Returns $getHumanTime(1.0)$.

State status after a discretization cycle:

- If the agent was able to choose its target product on the *Finish* stage and it was not interrupted after that, it considers the state finished with success;
- Otherwise it considers that it failed to choose a product to buy.

7.5.4 Looking for auctions

The agent knows which product it wants, now it needs to find some interesting auction that offers this product.

Stages functions:

- *doPrepare()* : The first time when this stage is executed it simulates the agent going to the section of auctions and typing the name of the product that it wants to find, returning $getHumanTime(1.0)$. If this function is called again in this state it will return 0.0 since the agent has already the list of available auctions;
- *doDo()* : Simulates the agent spending time looking for some auction in the generated list and selecting it if it finds one. Returns $getHumanTime(1.0)$;
- *doFinish()* : Simulates the reaction time of the agent after finding one auction and going to the bidding screen (setting the agent's target auction) or not. Returns $getHumanTime(1.0)$;
- *doRecover()* : This state is interrupted only when the agent receives a notification that it won some auction with the product that it was looking for. Returns $getHumanTime(1.0)$.

State status after a discretization cycle:

- If it was interrupted it considers the state as failed;
- If the agent was able to choose its target auction on the *Finish* stage and it was not interrupted after that, it considers the state finished with success;
- Else if it was not able to find an auction there but there are other auctions to look, it considers the state execution in progress;
- If the agent checked all available auctions from the auction list, it considers the state as failed.

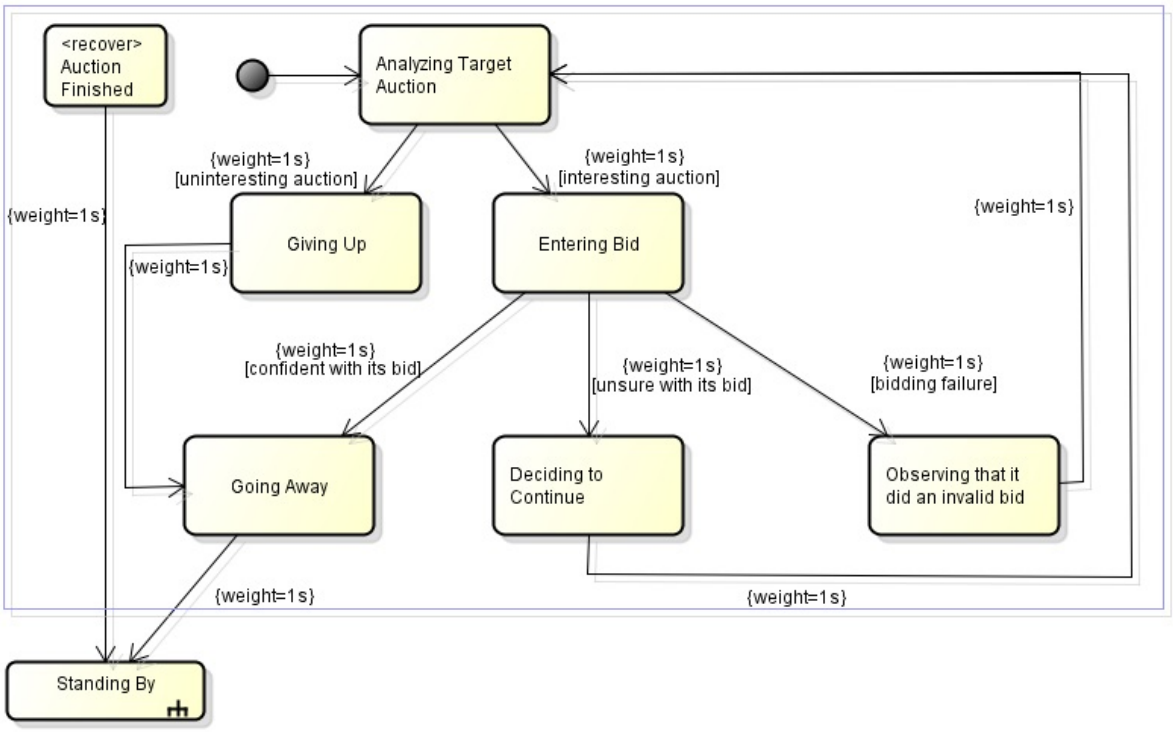

Figure 7.3: An example of the representation of the set of stages used to represent the state of bidding following the pattern from figure 5.3.

7.5.5 Bidding

The final moment where the agent decides if it will really bid to try to get that product or give up and move to another auction (figure 7.3). After doing this, it will stand by for some moments before deciding if it will want more products, starting a new cycle of actions in the site or staying halted for more time.

Stages functions:

- *doPrepare()* : Simulates the agent observing the details about the selected auction. Returns $getHumanTime(1.0)$;
- *doDo()* : Simulates the agent spending time to create a bid or deciding to give up from the target auction. Returns $getHumanTime(1.0)$;
- *doFinish()* : Simulates the agent observing if its bid was accepted or it exiting the auction. Returns $getHumanTime(1.0)$;
- *doRecover()* : This state is interrupted only when the current auction reaches its end, sending a message to this agent to go away since there is nothing to be done there. Returns $getHumanTime(1.0)$.

State status after a discretization cycle:

- If the auction stills interesting for the agent it considers the state execution in progress;
- Otherwise it considers the state as executed with success.

7.5.6 Bidder Agent State Transitions

The state transitions are controlled by an external planner and its idea follows a cyclic flow that ideally would execute the sequence of states *standing by* $\rightarrow$ *product choosing* $\rightarrow$ *auction choosing* $\rightarrow$ *bidding* $\rightarrow$ *standing by* $\rightarrow$ ... until the simulation time expires. However since it is based on a non-deterministic finite state machine, it can generate different execution paths. The planner handles this non-determinism generating the following transitions:

- *standing by* $\rightarrow$ *product choosing* when the *standing by* state is finished;
- *product choosing* $\rightarrow$ *auction choosing* when the *product choosing* state is finished successfully;
- *product choosing* $\rightarrow$ *standing by* when the *product choosing* state is finished with a failure;
- *auction choosing* $\rightarrow$ *bidding* when the *auction choosing* state is finished successfully;
- *auction choosing* $\rightarrow$ *standing by* when the *auction choosing* state is finished with a failure;
- *bidding* $\rightarrow$ *standing by* when the *bidding* state is finished;

7.6 Modeling the auction manager agent with D-Engine

We use of D-Engine's flexibility to create the auction manager agent that is responsible for basically clearing every auction when it is finished. When the simulation starts, this agent has its next event time $tProx$ scheduled to $+\infty$, so it would never be active.

However, when a new auction $i \in A$ is created at some time $\sigma_i < tMax$, it checks if $\sigma_i < tProx$. If that is true, a message is sent to this manager agent, triggering the recovering state that calculates the moment $tProx = \eta_i < tMax$ when an auction must be cleared.

When the simulated time reaches $tProx$, every auction with finishing time between $tProx$ and $tProx + MIN\_TIME$ is cleared. This happens because if some auction was finished in this interval it would be ignored by this agent since no agents can execute 2 stages in sequence with a time interval

smaller than MIN_TIME (excluding recovering cases). Finally if $L(t)$ is not empty it calculates a new $tProx$ as $tProx = min(\eta_j, j \in L(t))$. Otherwise $tProx = +\infty$.

To let this idea compatible with the D-Engine framework, we have the following stage functions for the state of *managing auctions*:

- *doPrepare()* : If this function is being executed for the first time in this state, it returns $+\infty$. Otherwise returns 0.0;
- *doDo()* : Clears every auction between the current time t and $t + MIN\_TIME$, removing them from $L(t)$. Notifies every auction winner and every auction participant at that moment. Returns $+\infty$ if $L(t)$ is empty. Otherwise returns $(\eta_j - t)$, where $j \in L(t)$ is the auction with lesser remaining time;
- *doFinish()* : Does nothing. Returns 0.0;
- *doRecover()* : This state is interrupted only when a new auction $j \in L(t)$, $t = \sigma_j$ has the remaining time smaller than any other auction from $L(t)$. Returns $(\eta_j - t)$.

At the beginning of the simulation, the planner sets this agent's state as *managing auctions* and it stays with this state until the end of the simulation. So the state status is always *in progress*.

7.7 Results

The implementation of this second application was far more complicated than the first one from chapter 6. This one has 6 states, two kinds of different agents, a real planner and it is able to run with more than two agents at the same time (figure 7.4). This application works as a proof of the framework's flexibility, a test suit for the framework features and as an example of how to develop a more complex application using it.

Following the development recommendations cited in section 5.3, we started implementing and testing the model objects that form the market, like people or auctions. Then we keep going from the bottom to the top until becoming able to run the simulation with the entire model representing our final target: an online auction site simulator.

A great amount of effort was needed to create all the unit tests responsible to check our auction proposed model, but after finishing the tests of each module, the model ran through the D-Engine and statistical analysis algorithms without great troubles. Without these tests it would be very tiresome to run

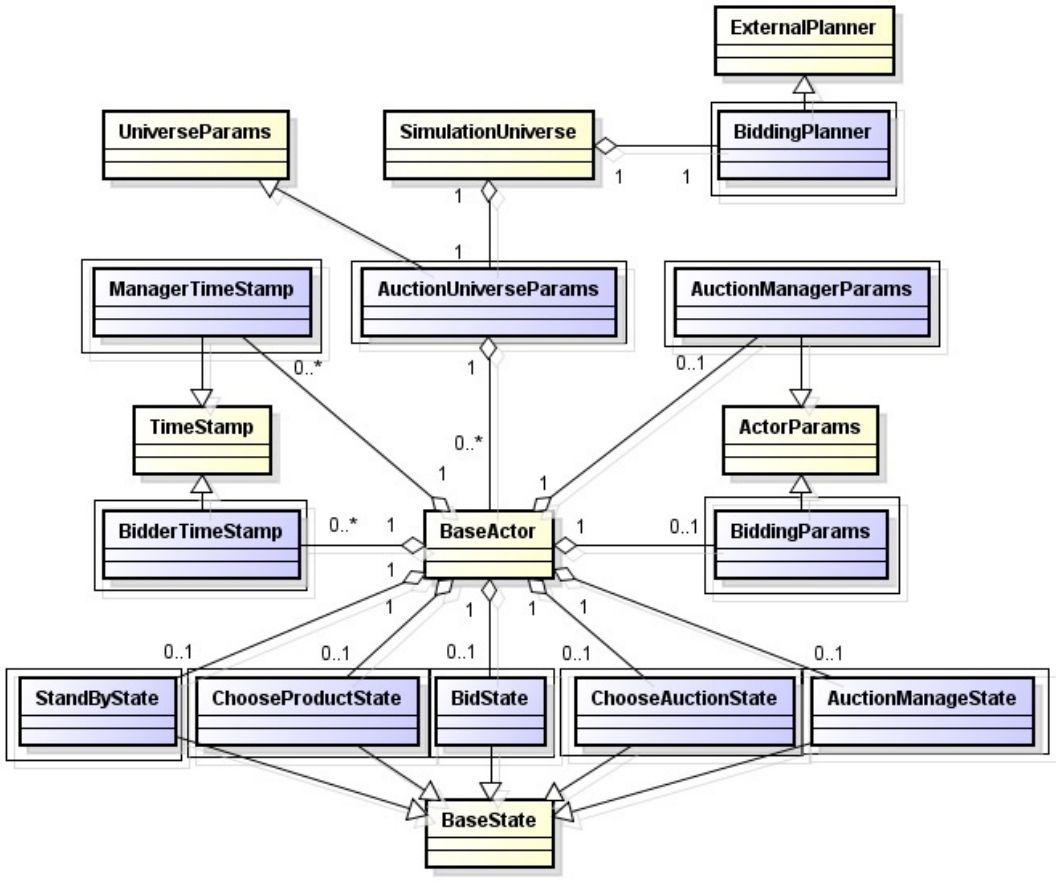

Figure 7.4: A simplified diagram class highlighting the most important custom classes responsible to represent this duel model inside of our framework with an extra square margin. Each actor can have different states but no more than one at the same time. There are different kinds of timestamps and parameters for each kind of actor. For this model we also needed to modify the simulated universe parameters since this case is more complex.

the entire model directly, probably generating a chain of errors that would be hard to debug.

7.7.1 Visual Results

After all those tests we developed a graphic interface to visualize the simulated model and let the user interact with it (figures 7.5, 7.6 and 7.7). For this application we present the model using only user interface elements as lists and buttons, differently from the application from chapter 6 where we used 3D models.

Since the player itself "simulates" its own behavior while it interacts with the interface, we do not need an extra agent to represent this kind of events. We intercept the player's inputs in the interface and convert it directly to modifications in the universe parameters, these inputs are related to the

following effects:

- Change the number of visitors in an auction: When the player access the bidding section where it can read more details about the selected auction and bid some value, we increase the number of visitors in that auction by one. When the user goes back to its product section, we decrease that value by one;
- Apply a bid: When the user confirms its bidding and this value is bigger than the current winner bid or there is no current winner bid in some auction, we insert a new bid in the list of bids of that auction. That bid indicates a special person related to the player. So, if the player's bid won some auction, the model can handle and indicate to him without any modification in the original model without external inputs.

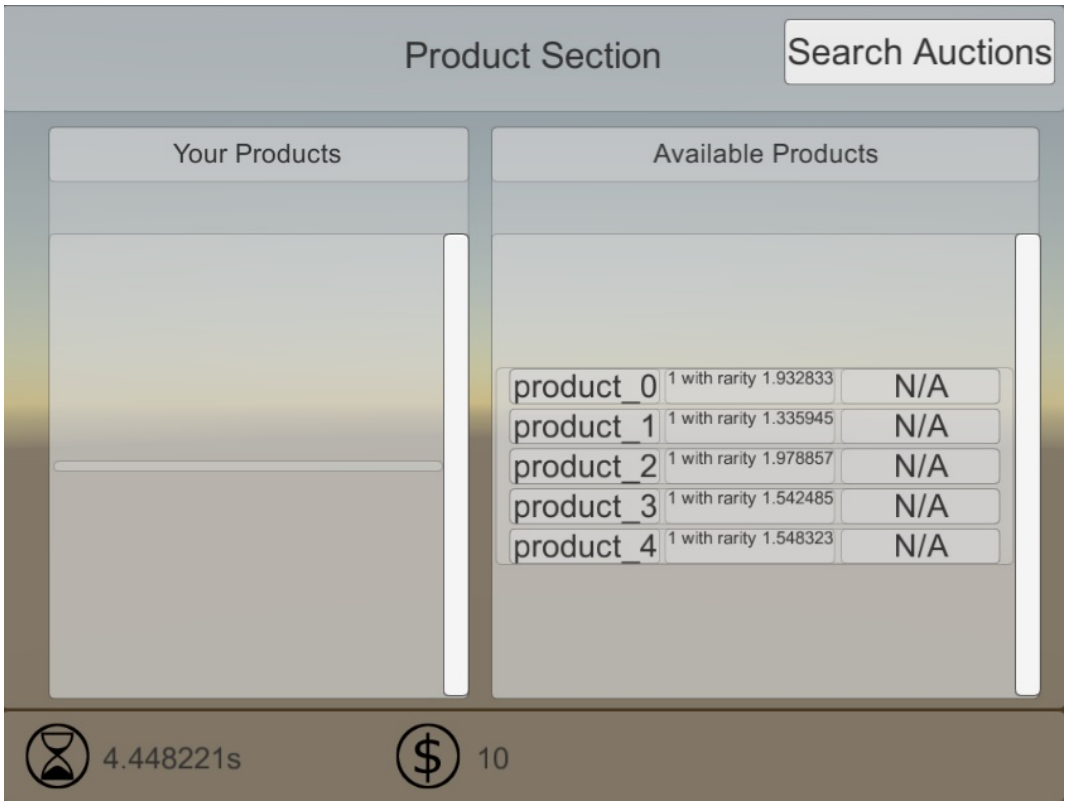

Figure 7.5: The first screen from the site simulator created with our framework. In the left it shows the products owned by the player and in the right it shows which kind of products exists in the current simulation. In the bottom we have the current time and the player‘s amount of money.

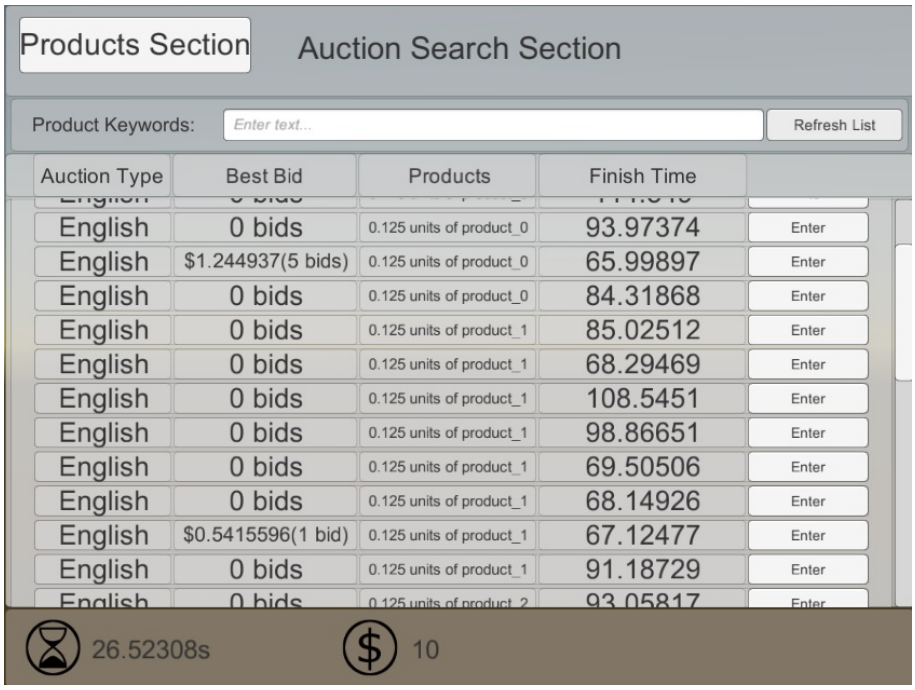

Figure 7.6: The second screen that lets the player find auctions related to the available products from figure 7.5.

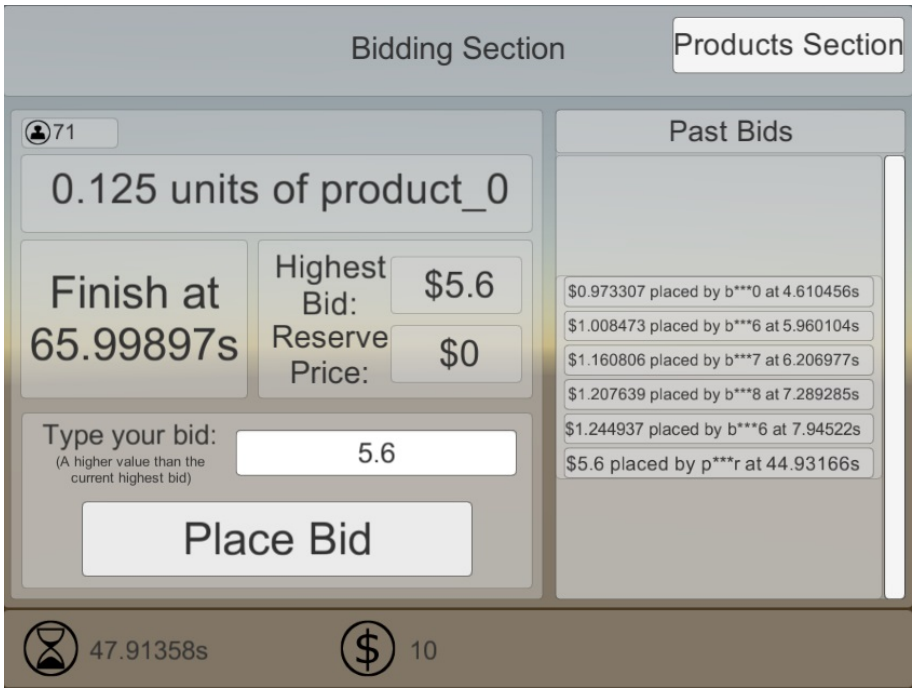

Figure 7.7: The third screen where the player can see more details about a specific auction. In this specific picture you can see that the player had bid a value, having its name exhibit in a obfuscated way on the right section.

7.7.2 Statistical Results

Finally we used statistical analysis to choose better parameters to estimate parameters to achieve some specific predicate. As an example we used the

following predicate:

> Our auction model has M bidders and 5 products. For each product, it exists exactly 1 unit of them in the universe and this amount can be divided to be sold equally in N different auctions, so each auction sells just $1/N$ units of a single product. If M is fixed with some value, which is the smallest round value of N guarantee that in 75% of all model simulations at least 10% of their M bidders will have at least one fraction of a product at the end? Which is the biggest value of M that keeps this predicate true?

We set the model refiner function (algorithm 5) to increase the number of auctions per product N bit by bit. With that refiner we ran the statistical analysis with some round values of M (showed in the table 7.1) and we got some values of N and M that satisfied our desired property. We also noticed that when $M \geq 300$ it becomes impossible to guarantee that predicate, so 250 bidders and 50 auctions would be a good number to use in the final model available for the player.

Algorithm 5 Tweak the simulated model to increase the number of auctions per product

```
1: function MODELREFINE(cheat)
2:     up ← the universe parameters
3:     up.numberAuctionsPerProduct ← 1 + Floor(cheat * 10)
4: end function
```

Table 7.1 Table with results of the statistical analysis ran with different values of M to find the best value of N. Table 7.2 shows the common parameters used in all simulations. Table 7.3 shows the bidding agent's range of parameters.

M	Minimum Cheat Needed	N	Analysis Total Time(s)	Number of Simulations	Number of Stages Executed	Average number of stages per agent
10	0.0	1	0.426	100	120901	120.901
50	0.1	2	7.800	228	1374748	120.591
100	0.3	4	195.047	1862	22600899	121.379
150	0.5	6	608.564	2828	51672815	121.812
200	0.7	8	1213.631	3365	82200385	122.140
250	0.9	10	2428.449	4499	137684667	122.413
300	-1.0	N/A	3338.633	4355	160106321	122.545

The results from table 7.1 were created in a machine with 64 Gigabytes of RAM, a processor Intel i7-3960X, Windows 7 64Bits and Unity3d 5.0.1f1

Personal in the editor mode. The analysis execution time rises with the increment of M because it increases the number of the agents in the simulated model. We can also observe other interesting facts in these results as how the number of auctions per products N grows at the same rate than the number of bidders M or that the average number of stages per person is almost constant for all values of M, maybe being an indicator that the agents expected behavior is independent of the number of auctions and bidders in the simulated model.

Table 7.2 Parameters that were not changed between the multiple executions of the statistical analysis to generate the table 7.1.

Bidder Initial Money	10
Product rarity maximum random increment	100%
Auction minimum finishing time	60 s
Maximum random increment on auction's finishing time	60 s
Desired Tolerance	75%
Desired % of Bidders with a fraction of some product in the end	10%
Number of Products	5
L	0.1
Refining Step	0.1

Table 7.3 Each bidding agent used to generate the table 7.1 is created using these variables to calculate its bidding value (Consider $runif()$ as a random number generator that follows the uniform distribution between zero and one). The weight of each of the four bidding constrains are generated calling the algorithm 6 with $X = 4$ and $Y = 1.0$.

β_{rt}	$1000 * runif()$
k_{rt}	$runif()$
β_{ra}	$1000 * runif()$
k_{ra}	$runif()$
β_{ba}	$runif()$
k_{ba}	$runif()$
β_{de}	$1000 * runif()$
k_{de}	$runif()$

Algorithm 6 Returns a vector with a random set of X positive numbers that sums the value of Y

```
1:  function RANDOMVECTOR(X, Y)
2:      if X = 1 then
3:          returna new array with Y as its single element.
4:      end if
5:      arr ← new empty array with X+1 slots.
6:      ret ← new empty array with X slots.
7:      for i=0 to X-1 do
8:          arr[i] ← runif() * Y
9:      end for
10:     arr[X − 1] ← 0
11:     arr[X] ← Y
12:     sort(arr)
13:     for i=0 to X do
14:         ret[i] ← arr[i + 1] − arr[i]
15:     end for
16:     returnret
17: end function
```

8
Comparison with Related Works

Our framework has been designed to work with generic models using generic animations to present its simulation. Its output is a list of timestamps describing important events that describes how each agent executes each state through time, so the module that receives these timestamps are responsible for detailing how it should be animated.

Initially it was thought to animate crowds as the armies from some real-time strategy game, but it showed to be useful as a generic agent-based model simulator. Being generic means that it is not optimized for a specific use, so the programmer should put more effort to make a model run as well as if it were being executed on some specialist system.

For instance, there are multiple systems specialized in crowd simulators and one of them famous for being used on the Lord of the Rings movies is called MASSIVE (8). It is far more mature, it has a wider scope than from our framework and it uses smart agents with many different techniques to animate different scenes with multiple agents. For example, an agent can have motion trees with hundreds of nodes describing what to do in each situation. These agents have sensors which let them react to other agents or events in their surroundings. Using them it is possible to do things like detect the angle of the ground where it is grounded and adapt its animation for it, blending pre-made animations for different angles. These animations can be done by motion capture from real actors and by describing the agents body with basic geometric shapes and joint limitations, it is possible to calculate all possible body motion combinations that can be used. The results generated with that software are far more realistic than the ones shown by our work, but the software is not free and it needs far more computational power to render each animation.

Another work (12) focused on crowd simulation uses a finite state machine to create a sequence of actions to move agents between two points in the space using A* algorithm. Each state says how the agent should act to move itself in the space with actions like jogging, crawling or jumping and it may contain multiple motion animation clips. That works supports multiple

agents and moving obstacles like falling trees and can be used to animate agents like horses and humans running or riding a skate. For some environments like horses running from moving obstacles it has a nice performance, simulating 16 seconds of animation in just one second.

Going from macro to micro animation generation, there is a research (18) which generates procedural animation to present some competitive game as tag or sword fighting between two-players being able to act simultaneously in a very rich and realistic way. It is based on game theory methods like zero-sum Markov Games and Markov Decision Processes; however, since these methods can require exponential time and storage according to game's state space, it employs an offline learning algorithm to generate a value function to be used in the policy generation that will describe how the game will be presented. It uses motion models to generate custom animation clips for each action in different occasions.

Simulation visualization is also useful in agent-based models (15). In many cases this presentation is very crude but this is not a problem because generally we are interested in finding some emergent pattern or behavior when observing the abstraction of some system. Even animations using simple images as circles to represent some agents moving can be sufficient to let us understand better how some environments work, such as: criminal rebellions, wolf-sheep predation, war logistics and tumor growth.

While it seems possible to simulate agent-based models for that purpose with our framework, it is important to remember that the focus of our work is animation for entertainment. The effort used to design the model and to simulate it with our framework may be excessive and unnecessary for many cases, demanding more computational power than needed by other tools focused on that kind of analysis.

9
Conclusion

9.1
Contributions

In this dissertation we proposed a framework to execute atomic actions from an arbitrary non-deterministic planner in a random way. This framework architecture aids the user to model a custom agent-based model. It offers functions that let the user analyze the model based on the statistics generated by simulating that custom agent model hundreds of times and the output generated by each model simulation is compatible with multiple ways to represent content as graphical or textual methods. Moreover, the models created in this framework are compatible with external interaction, being suitable for applications in areas like interactive medias or digital games.

To test the framework efficiency, generate example cases about how to use it and how to create an agent-based model compatible with the framework we created two different applications:

The first application was a simple duel between two warriors. Its main features are:

- It shows how to send external messages to the simulation's agents (affects how the agents modify the universe);
- Graphical representation of the model simulation based on 3D models;
- The representation focuses on how an agent executes an action;
- Agent model with only one kind of agents and no need of real planners;
- The simulation ending is based on the warriors lifepoints (variable time simulation);
- Since the model is very simple, it works as an introductory project.

The second application is a bigger project of a serious game about an e-commerce application based on English auctions. This one has the following characteristics:

- It shows how to let an external element manipulate the simulated universe parameters directly (affect the universe to change how the agents behave);
- Graphical representation of the model simulation based on user interface elements;
- The representation focuses on the result of the agent actions;
- Compatible with a variable number of agents with different types and with some of them having multiple states, so this model has also a planner to control them;
- The simulation ending is based on the total simulated time (constant time simulation);
- It works as an example of how to develop a complex project, going beyond the basics shown on the duel project.

Even with different objectives and scopes, those applications show that our framework can be useful for different areas and agent models. Moreover each one presents different features that can be combined to be create a new different model.

Besides these projects and this thesis itself, the framework has documentation by the tool Doxygen [1] about its code made to work with Unity3D Personal version 5.0.1f using the C# programming language.

9.2 Discussions

This framework is only recommended for models where each action can be described in discrete points of time. In some occasions it can be possible to abstract a continuous model like those ones based on physics into a discrete model, but we would lose some details on the results, the model generation would be too much complex or it would require too many states to represent all possible actions with different variations.

To achieve an even more realistic presentation, it may be better to use other approaches than the discrete one proposed here. It is interesting to use our framework when the available computation resource is limited by some reason as the nature of the animation or the number of agents in the planned model. Also it is useful when the desired model events can be described in discrete points of time without losing any relevant data.

Another limitation is that a single agent can execute only one action per time, so this framework is only able of handling simultaneous actions

[1] www.doxygen.org/

from different agents at the same time. While some animation tools allows animation blending between multiple animation clips (for example branding a sword while the warrior is running), our framework does not handle it natively. The programmer would need to create a new set of states that represent this kind of combination of actions or a set of timestamps with enough parameters to say that some stage should be executed in a different way.

Another restriction is that the simulation's result is not 100% guaranteed in all configurations of an arbitrary model. This does not mean that it is impossible to achieve this status but it can be troublesome to balance the presentation randomness with this very precise result estimation. For example, in the duel example if we wanted to let warrior 1 win every time it would be hard to make the fight look disputed. Warrior 2 would seem to be too weak, removing the emotion of watching this fight since its result can be quite clear since its very beginning.

This uncertainty in the simulation's final result happens because the execution of a single state may have a random result, affecting the entire set of states needed to achieve a result. Therefore, programmers should be careful when designing a model with some crucial state needing to generate the same result every time. For example if we are presenting an animation with final goal as "save the princess", the hero could do lots of different actions like explore dungeons or collect gold until he reaches the princess. He would fail in some of them but it would still be possible to achieve the final goal. However the real state where the hero saves the princess must always be executed successfully or the planner should have a recovering set of states in the case of failure in that specific state.

9.3 Future Works

In this work we implemented all basic features that the framework should provide to offer the features that we wanted, however there are some points where it can be optimized. One special improvement would be checking whether the D-Engine algorithm can be designed in a way that it would run in multiple threads or maybe execute the statistical analysis running multiple simulations at the same time using multiple cores.

Another way of accelerating the statistical analysis would be to apply variance reduction techniques to reduce the necessary number of simulations executed in an analysis to reach some tolerance value. The framework would provide multiple variance reduction techniques and let the user choose one that would be most suitable for his custom model at the moment.

Maybe it would be interesting to see how recursive D-Engine models would work. It would have a state which is defined by a model that also uses the D-Engine algorithm to describe how it is executed. This way it would be easier to create complex models and focus on the simulation and analysis of a single state.

One interesting kind of output that would be generated by the framework would be, for each kind of agent, the amount of time that it uses in each state and, for each state, the amount of time used in each stage too. This would be useful to have an idea of what each agent does when the model is executed.

For each agent it also would be convenient to have access to a diagram indicating the possible state changes that it can handle and the probability of each transition, abstracting that agent with the specific parameters used in that analysis into a Markov Chain.

Finally it would be interesting to create something like an abstract family of agent-models for different proposes, but with a common set of states, stages or similar planners. With that, final users could create different kinds of similar agent-based models with different parameters or small modifications more easily. For example, if we had a generic model based on people behaviour when they were using a computer product and able to calculate the duration of each stage based on the product's utilization log, we would have a framework to simulate humans in various environments where there are human conflicts, as the auction e-commerce application shown in this project.

9.3.1 Simple Duel Application

It would be interesting to let the duel have more attacks than just the punch, for example it would include kicks or elbow attacks that would be triggered according to the values of the calculated damage or attack duration. It also would include actions and intelligence for dodging, blocking and counter attacks.

The current duel project includes only two warriors fighting, it would be interesting to see more agents battling, maybe including two or more teams. Also it would be interesting to let the agents have more states, like movement to chase other agents or to run away in the case of fear or power inferiority.

The agent model would include different classes of warriors like archers, knights or barbarians which would share some states or including exclusive ones. If we create an easy way of defining different actions for agents in a battle or war environments, we would have a generic agent-based model for this kind of content, being useful for bigger projects as strategy games or war

movies.

For each kind of different model using our framework it would be interesting to have empirical tests about the reaction of the viewers when observing the generated presentation and verify how the randomness can influence the consumer experience and increase the replay factor. For example, we would get a number U of users, show for $U/2$ users a presentation using classic deterministic animations and show for the other half a presentation based in our framework. Then we could give a survey about what all the users observed and compare the results.

9.3.2 Auction Site Application

The auction model introduced by this thesis only presents English auctions, it would be interesting to have different kinds of auctions like Dutch or Vickrey ones. Then the same agent would have to adapt its strategies for each kind of auction, generating more dynamism in the simulated system.

Talking about strategies, in our project every agent's behaviour is based on the same strategy concept. While it is possible to simulate different agents by changing their parameters, their strategies will still be similar. It would be interesting to have other agents with different strategy concepts, maybe even agents that acts in groups to take advantage over the others or agents with a "bad" behaviour, trying to gain money by exploring sharp practises like shilling, shielding or rings (22, Chapter 3.7).

Going back to our agents, sometimes they will start to give new bids that are greater than the previous one by just some very small fractions of money (lesser than $0.01), looking very unnatural. It would be interesting to adapt the bidding calculation functions to try to fix this problem. Also they spend a random time in every stage but the parameter used is the same; it would be interesting to have more complex and flexible ways of defining the time in each stage, maybe even based in some empirical study about human speed when using computer interfaces.

This project was focused on the buy-side of auction, it would be interesting to let agents create their own auctions and sell their products, demanding a more advanced strategy to see when it should buy or sell products to have the maximum possible gain. Other nice point would be to give a purpose for each product, consequently being able to give a concrete objective for each agent. As a recreational example, imagine that our agents can create fighting robots, then one agent would want to construct an extreme powerful robot while another agent would want to have an army of average robots and it would have a

rival agent that has as objective having the best army to beat him. An universe with objectives like that would create a bigger, more complex and wide range of possible stories to be played or watched by the final viewers.

Bibliography

[1] SHUBIK, M.. **The dollar auction game: A paradox in noncooperative behavior and escalation**. Journal of Conflict Resolution, p. 109–111, 1971.

[2] HOPCROFT, J. E.. **Introduction to automata theory, languages, and computation**. Pearson Education India, 1979.

[3] FININ, T.; FRITZSON, R.; MCKAY, D. ; MCENTIRE, R.. **Kqml as an agent communication language**. In: PROCEEDINGS OF THE THIRD INTERNATIONAL CONFERENCE ON INFORMATION AND KNOWLEDGE MANAGEMENT, p. 456–463. ACM, 1994.

[4] GAMMA, E.; HELM, R.; JOHNSON, R. ; VLISSIDES, J.. **Design patterns: elements of reusable object-oriented software**. Pearson Education, 1994.

[5] FARATIN, P.; SIERRA, C. ; JENNINGS, N. R.. **Negotiation decision functions for autonomous agents**. Robotics and Autonomous Systems, 24(3):159–182, 1998.

[6] STONE, P.; VELOSO, M.. **Multiagent systems: A survey from a machine learning perspective**. Autonomous Robots, 8(3):345–383, 2000.

[7] ANTHONY, P.; JENNINGS, N. R.. **Developing a bidding agent for multiple heterogeneous auctions**. ACM Transactions on Internet Technology (TOIT), 3(3):185–217, 2003.

[8] AITKEN, M.; BUTLER, G.; LEMMON, D.; SAINDON, E.; PETERS, D. ; WILLIAMS, G.. **The lord of the rings: the visual effects that brought middle earth to the screen**. In: ACM SIGGRAPH 2004 COURSE NOTES, p. 11. ACM, 2004.

[9] GHALLAB, M.; NAU, D. ; TRAVERSO, P.. **Automated planning: theory & practice**. Elsevier, 2004.

[10] GRAEPEL, T.; HERBRICH, R. ; GOLD, J.. **Learning to fight**. In: PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON COMPUTER GAMES: ARTIFICIAL INTELLIGENCE, DESIGN AND EDUCATION, p. 193–200, 2004.

[11] CARLSON, M.; MUCHA, P. J. ; TURK, G.. **Rigid fluid: animating the interplay between rigid bodies and fluid**. In: ACM TRANSACTIONS ON GRAPHICS (TOG), volumen 23, p. 377–384. ACM, 2004.

[12] LAU, M.; KUFFNER, J. J.. **Behavior planning for character animation**. In: PROCEEDINGS OF THE 2005 ACM SIGGRAPH/EUROGRAPHICS SYMPOSIUM ON COMPUTER ANIMATION, p. 271–280. ACM, 2005.

[13] MITRA, S.; BANKER, R.. **Comparing commodity prices in electronic and traditional auctions: empirical evidence from indian coffee auctions**. In: PROCEEDINGS OF THE 7TH INTERNATIONAL CONFERENCE ON ELECTRONIC COMMERCE, p. 233–235. ACM, 2005.

[14] GREENWOOD, D.; LYELL, M.; MALLYA, A. ; SUGURI, H.. **The ieee fipa approach to integrating software agents and web services**. In: PROCEEDINGS OF THE 6TH INTERNATIONAL JOINT CONFERENCE ON AUTONOMOUS AGENTS AND MULTIAGENT SYSTEMS, p. 276. ACM, 2007.

[15] ALLAN, R. J.. **Survey of agent based modelling and simulation tools**. Technical report, 2009.

[16] PASSOS, E. B.; SOUSA, J. W. S.; CLUA, E. W. G.; MONTENEGRO, A. ; MURTA, L.. **Smart composition of game objects using dependency injection**. Computers in Entertainment (CIE), 7(4):53, 2009.

[17] JIA, J.; ZHANG, Q.; ZHANG, Q. ; LIU, M.. **Revenue generation for truthful spectrum auction in dynamic spectrum access**. In: PROCEEDINGS OF THE TENTH ACM INTERNATIONAL SYMPOSIUM ON MOBILE AD HOC NETWORKING AND COMPUTING, p. 3–12. ACM, 2009.

[18] WAMPLER, K.; ANDERSEN, E.; HERBST, E.; LEE, Y. ; POPOVIĆ, Z.. **Character animation in two-player adversarial games**. ACM Transactions on Graphics (TOG), 29(3):26, 2010.

[19] DA SILVA, F. A. G.; CIARLINI, A. E. ; SIQUEIRA, S. W.. **Nondeterministic planning for generating interactive plots**. In: ADVANCES IN ARTIFICIAL INTELLIGENCE–IBERAMIA 2010, p. 133–143. Springer, 2010.

[20] BARBOSA, S. D. J.; FURTADO, A. L. ; CASANOVA, M. A.. **A decision-making process for digital storytelling**. In: GAMES AND DIGITAL ENTERTAINMENT (SBGAMES), 2010 BRAZILIAN SYMPOSIUM ON, p. 1–11. IEEE, 2010.

[21] BONET, B.; HANSEN, E.. **Heuristic search for planning under uncertainty**. Heuristics, Probability and Causality: A Tribute To Judea Pearl, p. 3–22, 2010.

[22] PARSONS, S.; RODRIGUEZ-AGUILAR, J. A. ; KLEIN, M.. **Auctions and bidding: A guide for computer scientists**. ACM Computing Surveys (CSUR), 43(2):10, 2011.

[23] ROTH, C.; VERMEULEN, I.; VORDERER, P. ; KLIMMT, C.. **Exploring replay value: shifts and continuities in user experiences between first and second exposure to an interactive story**. Cyberpsychology, Behavior, and Social Networking, 15(7):378–381, 2012.

[24] KNIGHT, F. H.. **Risk, uncertainty and profit**. Courier Dover Publications, 2012.

[25] ROSS, S. M.. **Simulation**, chapter 7 - The Discrete Event Simulation Approach, p. 111–134. Academic Press, 2012.

[26] ZHANG, K.; COLLINS JR, E. G. ; SHI, D.. **Centralized and distributed task allocation in multi-robot teams via a stochastic clustering auction**. ACM Transactions on Autonomous and Adaptive Systems (TAAS), 7(2):21, 2012.

[27] RABIN, S.. **Game AI Pro: Collected Wisdom of Game AI Professionals**. CRC Press, 2013.

[28] ABELHA, P.; GOTTIN, V.; CIARLINI, A.; ARAUJO, E.; FURTADO, A.; FEIJO, B.; SILVA, F. ; POZZER, C.. **A nondeterministic temporal planning model for generating narratives with continuous change in interactive storytelling**. In: NINTH ARTIFICIAL INTELLIGENCE AND INTERACTIVE DIGITAL ENTERTAINMENT CONFERENCE, 2013.

[29] DE MELLO CAMANHO, M.. **A Model for Stream-based Interactive Storytelling**. PhD thesis, PUC-Rio, 2014.

[30] ROBERTSON, G.; WATSON, I.. **A review of real-time strategy game ai**. 2014.