Imperial College London

BEng Final Year Project

A Computational Analysis of Linear Segregation Models

Author:
Roxana G. Ursu

Supervisor:
Dr. Paolo Turrini
Second Marker:
Dr. Panagiotis Kouvaros

June 20, 2016

Abstract

Society can segregate by so many characteristics such as social standing, income, educational background, ethnic groups, colour of skin, shared language, gender, age, religion. Segregation can lead to perceived prejudices, such as dislike for other groups and inaccurate judgement of correlations (expecting some social groups to be less intelligent, be more aggressive, have a tendency for extremism, be more tolerant, etc.).

In this project we are going to have a look at one of the first papers that address *racial segregation* and presents a social dynamics model, *Dynamic models of segregation*, **?**. We are going to look at Schelling's linear model and at some of the theoretical statements he made. Schelling's linear model is a *two-type agent model*, that is, the agents are one of the two possible types: blacks or whites. In any other aspects, the agents are identical. This model, although simplistic, helps us understand why segregation is such hard problem to combat.

The main contribution of this project is a rigorous justification for some of Schelling's intuitions and simulation results regarding the relaxation of the turn function and convergence of the system. Furthermore, we give a formal analysis of the impact of turn functions in linear segregation models. We also designed a multi-agent computational model that allowed us to automatically test our formal configurations using `NuSMV` model checker. We formally proved that a particular turn function, constructed by relaxing Schelling's turn function, would always ensure that our system reaches a stable final configuration. The report details the approach taken in analysing the impact of turn functions and the convergence of the system both in analytical and simulation approach.

This projects provides valuable insight into the potential of using a model checker over an object oriented design to test the behaviour of a multi-agent computational model. We are going to see the numerous advantages of this approach, however, we have to keep in mind the *State Explosion Problem* when we are scaling up the system. Further work is required to either ensure the construction of a compact model of the system or to develop techniques that test large size systems by only looking at a fraction of the system.

Acknowledgements

Firstly, I would like to thank my supervisor, Dr. Paolo Turrini, for his guidance and contagious enthusiasm throughout this project. Secondly, I would like to thank my second marker, Dr. Panagiotis Kouvaros, for his advice and assistance especially during the designing and implementation stages of the computational model.

Furthermore, I would like to thank my family and friends for their unconditional love and encouragement throughout my time at Imperial College.

Contents

1 Introduction

In 1917, Marquis Converse produced the very first basketball-only shoes (or sneakers), named Converse All Stars. In the same year, Chuck Taylor started wearing them as a high school basketball player at Columbus High School and in 1921, Taylor started working with Converse to improve the design by enhancing the flexibility and support of the shoes. The new, improved design became known as the Chuck Taylor All-Stars and they are the best selling basketball shoes of all times.

Shortly after, in 1924, the sneakers went international with a German man named Adi Dassle who created the first Adidas sneaker, followed by his brother Rudi who started another famous sports shoe company: Puma.

Although in the first part of the 20th century, sneakers were worn only by people practicing sports, in 1950s children, teens and young adults started wearing them as a fashion statement. Sneakers became even more famous after they appeared in movies such as "Rebel Without a Cause". Nowadays, sneakers have a place in almost everyone's wardrobe and they are not considered as a shoe worn only during sports activities anymore. Although wearing sports shoes in our day-to-day life might be because of an increase in comfort level, the case is not that clear. What we can clearly notice is how fashion spreads gradually because of people emulating the behaviour of others.

An average 120,000,000 pairs of Nike shoes are sold each year according to *Statistic Brain* [**?**] and the numbers are expected to rise. This is just one example where the people's "independence" assumption fails. There are numerous papers studying just this, how our decisions on where to live, what to wear, what to eat, etc. are influenced not only by the value of the goods, but by the choices people around us do. Some of the papers focus on some of the very serious problems of our society like dropping out of school, smoking, committing crimes. Here, we could mention [**?**]; [**?**]; and [**?**]. Also [**?**] discuss A Dynamic Model of Conformity. However, to understand the complex system we know as *society* and to understand why segregation still emerges in our society we have to look at the problem from all perspectives.

The "dual" situation is the one in which individuals do not change their behaviours, but they prefer to move to neighbourhoods where people have same interests as oneself. This is a model that was first published by [**?**]. We are going to look at Schelling's linear model in detail in this paper. The two problems may seem completely different, but there are similar aggregate properties. We are going to see how segregated patterns (spatially homogeneous patterns) are more likely to emerge than heterogeneous patterns in a final configuration.

1.1 Motivation

A social network is a representation of social agents (such as individuals, organisations or software), a social connection and a social interaction between the agents. Understanding how relations between individuals and individual decision-making affect the system on a larger scale, it would enable us to predict how the system will evolve. This is important since we are interested in understanding the factors that lead to a segregated system in the future, in order to try and prevent such outcome.

We live in a world where segregation, be it ethical or economical, is still present at every step, in all societies, even in developed, modern cities like London. There is a huge interest in finding a way to combat any further segregation, but this is not easy and the consequences of people's low tolerances can be seen everywhere. *Poor doors: the segregation of London's inner-city flat dwellers* [**?**] and *What would a ban on 'poor doors' achieve?* [**?**] are only two articles published by [**?**] that are presenting serious concerns that people and governments have regarding the economic segregation that is happening in London, UK.

Osborne and Hill present in their articles a worrying phenomena that is currently happening in London: segregation created by new builds in Central London. Developers are obliged to provide a number of affordable homes when they draw up a new housing project. What was suppose to be a solution to the housing crisis and to combat segregation by keeping low-income residents in central London, lead to even steeper local segregation between the two groups of tenants. The affordable housing tenants are forced to use different entrances to the buildings, they do not have access to facilities such as car or bike parking and in some cases, even their bins and postal deliveries are being separated. The developers say that this is the only way to keep the price low for the low-income tenants since service charges to maintain facilities such as 24/7 reception desk are high. However, all these measures would lead to increasingly divided communities. Authorities, such as the Mayor or government could ban having this division between the two classes that live in the same building or in the same complex, but it could make the prices of "affordable" flats to increase, thereby removing low-income tenants. On a larger scale, it might seem that the segregation problem is solved since the poor and the rich live next-door to each other. However, when we look closely we see just how big the division between the two classes is. This shows us how complex the segregation problem is and how difficult it is to come up with good, practical solutions. We want to look at this persistent problem: the interrelationship between group and individual behaviour from a new perspective, making use of new computational model techniques.

The housing crisis is just one example of how hard it is to create good laws or standards for a government or business. The act of creating laws is called the *policy making*. There is a high volume of research needed behind the scenes before a new law gets to be realised. In Britain, *The Centre for Policy Studies*[**?**] is one of the leading think-tanks, working with brilliant minds from all around the country to develop policies for politicians, media or anyone interested in public debate in Britain. They release regular publications designed to influence government policy. It is a research unit that focuses on complexity science and social simulation. In other words, they use social simulations (`e.g.` multi-agent computational simulation) for informing politicians what to do, what laws to consider that are in the best interest for the country and its citizens. We will have a look at just what social simulation and computational modelling are and their importance in today's research.

1.2 Objectives

Schelling's segregation model presents one of the earliest examples of social dynamics models. In this project, we want to study and understand Schelling's segregation models, focusing on the linear model. We want to check whether some of Schelling's intuitions and simulation results can be rigorously justified. We will also going to use a model checker, `NuSMV`, to automatically test some of the results.

Furthermore, we want to get a better understanding of why segregation could happen even if, as individuals, the people prefer to live in an integrated neighbourhood. We will look whether making small variations to Schelling's model, like relaxing or changing the turn function, will lead to different final configurations. We are interested in our final configurations to be *stable* and the *segregation problem to be minimal.* We want to use the model checker to find out whether non-segregated configurations can be reached and what path we have to follow to get there.

1.3 Contributions

This project makes the following contributions:

- A formal definition and proof of convergence and termination of final configurations in a linear model;
- Rigorous justification for Schelling's intuitions and simulation results when it comes to relaxing the turn function in order to ensure a stable final configuration;
- A formal analysis of the impact of turn functions in linear segregation models;
- An analysis of the degree of segregation of a configuration;
- A computational multi-agent model checker of the linear model, using `NuSMV`;

- Testing convergence and complete segregation with the help of the `NuSMV` model checker.

1.4 Report Outline

Chapter 2 presents a general background for our project. We start by discussing what complex systems are and look at the general idea of emergent properties of a system. Next, we look at the importance of computational models and how we use them to accelerate scientific discoveries. In particular, we have a look at agent-based computational models with one example (Multi-Agent Bargaining Model) to emphasise how agent-based models work. Before we go in detail and talk about Schelling's model in Chapter 3, we have a look at influence Schelling's model have had. Finally, we discuss about the environment we used to automatically test Schelling's model and the reasoning behind our choice.

Chapter 3 focuses on the main paper we studied: Schelling's Dynamic Models Of Segregation, 1971. We start with an overview of the model followed by analysis of the paper. We look at the Spatial Proximity Models where we are mainly interested in the linear model. Next, we have a brief overview of the Bounded-Neighbourhood Model and Tipping Model.

Chapter 4 introduces some formal definitions for the multi-agents line model. We are formally describing the line model as a permutation, defining similarity and what we mean by configurations. We also give formal definitions for turn functions, neighbourhood and happiness. In the second part of Chapter 4, we analyse the impact of turn functions, starting with Schelling's turn function and moving into different turn functions with some examples. Lastly, in this chapter, we talk about convergence and stable configurations. We have a look at a few conjectures based on these definitions and sketch some proofs.

Chapter 5 presents in detail how `NuSMV` model checker works and how we modeled our multi-agent line system. We also look into how we wrote the CTL Specifications to test *convergence* and *the level of segregation*. Finally, we have a brief look over the Python Script that is used to generate the `.smv file` (our model) for a given number of agents, a neighbourhood size, and an initial configuration.

Chapter 6 focuses on the results we get regarding convergence and complete segregation of the system. We analyse the results and also give an overall evaluation of the computational model.

Chapter 6, the final chapter, presents an overview of the project and discusses future work.

Finally, **Appendices** present an example of a `.smv file` model for a specific configuration and an example of counter-example generated by `NuSMV`.

2 General Background

To understand how segregation occurs in a system, we have to look at how the decisions of the individual parts of the system affect the system as a whole. In other words, we have to look at the emergent properties of the system. In this chapter, we will see what complex systems and emergent properties are. We will try to give a definition to segregation, look at computational models, at what they are and how they are being used to accelerate scientific discovery. Then, we will look at a particular set of computational models: the agent-based model. Schelling's segregation model is an agent-based model, but before we will discuss in detail Schelling's model in the next chapter, we will look here at a slightly different model: the multi-agent bargaining model. This will give us a better understanding of how agent-based models work and why they are so important. Next, we will look at a few papers that were influenced by Schelling's model and finally, we will talk about the technical tools we chose to construct our computational model and the reasoning behind our choices.

2.1 Emergent Properties

In philosophy, **emergence** refers to the process where certain patterns or regularities arise from the interaction of smaller identities that do not exhibit such properties. When we talk about **emergent property** we refer to a property that a system or a collection has, but which does not characterise the individual parts of the system. In other words, we apply the term "emergent" to certain properties of the whole complex system. Such a complex system is, for example, the brain with neurons representing the individual parts, or the Earth's global climate is also seen as a complex system.

It is generally agreed that not all properties of a system are emergent properties. But what makes a property *emergent*? Mark Bedau in *Weak Emergence*[**?**] says that a property is to be considered an emergent property if it is both *autonomous* and *constituted* by the underlying processes. It is important that both properties are met simultaneously. Being just constituted by the underlying processes, `i.e.` not autonomous, would not differentiate this property from any resultant property. For example, the weight of a system is calculated by adding up the weight of the individual parts of the system, but it is not autonomous.

In 2003 in *Downward Causation and the Autonomy of Weak Emergence*[**?**], Bedau mentioned a third criterion that should be considered: the property needs to be relevant to scientific practice. In the article *Simulation-Based Definitions of Emergence*[**?**], Alan Baker states that this vague criterion "has led to a focus on the cluster of approaches including neural networks, agent-based models, and dynamical systems theory which has come to be referred to collectively as complexity science".

Thus, the notion of "emergence" can be subdivided into two categories: **weak**

emergence and **strong emergence**. The weak emergence is a type of emergence where the emergent property can be simulated by computers. On the other hand, a strong emergence is an older notion and it implies that the emergent property cannot be simulated computationally.

Bedau [**?**] has the following definition of weak emergent: Given a system and a property P of the system, P is weak emergent if and only if P is delivered from all the system's micro facts but *only* by simulation. Although the idea of a simulation is well-understood, the line between an emergent property that can be deduced only by simulation and an emergent property that can be derived, predicted or explained from a present state is unclear. In his article for the *Journal of Artificial Societies and Social Simulation, Volume 13* [**?**], Alan Baker addresses this issue. He explores the epistemological ramification of the definition by trying to answer questions, such as: How and when emergent properties can be predicted? Without clear boundaries between simulation based and non-simulation based techniques, the definition does not hold. Baker also emphasises how challenging it is to define such boundaries. Hence, tracing a path between weak emergent and strong emergent is not trivial. However, in both cases, weak and strong emergence, we are interested in studying the new properties that emerge once the system gets larger, properties that are not shared by the individual components of the system and do not appear at a previous state. We are interested in how the interaction of each individual part with its immediate surroundings causes a chain of processes that can lead to some kind of pattern or order in the final state. And we are also interested in the role the simulation has.

2.2 Computational models in social sciences

Experiments run by economists and social psychologists and the study of history has shown that human motivations are not entirely self-regarding, they are influenced by others around us and in many cases, we can notice the existence of sacrifice towards a common, larger objective. And although, so far, group effects were the focus of empirical studies by historians, economists and other social scientists, in order to understand how individuals choose their preferences, it is necessary to analyse both the individual and the group interaction. Evolutionary Game Theory is just one example that focuses on the individual, using agent-based models. Furthermore, formal evolutionary models do not take into account the fact that human groups are highly segregated, and in many cases the segregation happens intentionally. The individual interaction is almost never random when it comes to his or her preferences.

In this section, we are going to look at the definition of segregation and then we are going to argue **the need of computational models in general and the agent-based models in particular**. We are going to look into detail to the pros and cons of an agent computation and we will mention different types of agent-based models.

2.2.1 Segregation Overview

Segregation may refer to a separation of people based on their ethnicity, race, sex, religion, geographic location, housing, occupation, economic status, age, etc., or a separation of objects, for example, segregation of materials or particle segregation. We are going to focus on *racial segregation.* However, the agent-based model that we are going to discuss in this project can be applied to any type of segregation, for any two types of agents (a type A and a type B).

Racial Segregation is the separation of humans into ethnic or racial groups in daily life. It may apply to activities such as eating in a restaurant, drinking from a water fountain, using a public toilet, attending school, going to the movies, riding on a bus, or in the rental or purchase of a home[**?**]. Thomas Schelling was one of the first scientists to address Racial Segregation and present a social dynamics model.

With a few exceptions, throughout the history, wherever there have been multiracial citizens, there has been racial segregation. An example of racial segregation in our history could be the 35 acts by Kilkenny in 1366 (The Statutes of Kilkenny) which were forbidding English settlers in Ireland to marry Irish people, adopt Irish children or even have Irish names. We could also mention the continuous battle between whites and blacks in the United States of America. Throughout history, there are plenty of examples where the separation between the people of the two different colour groups in the USA was very clear: we talking about different lanes for movie theatres, different restaurants or housings for white and blacks. Another, more tragic example of racial segregation is the Nazi Germany(1933-1945). Nazi started a *racial hygiene programme* by compulsory extermination of Untermenschen (”sub-humans”). They labelled Jews and Romani (or Gypsies) and persons of colour as inferior, inhuman and thus unworthy of life. [**?**]

Sadly, segregation is still a big issue in nowadays society. We could mention here an example of segregation as recent as 2007. On 28 April 2007, in Bahraini (officially the Kingdom of Bahrain, an island country), the lower house of Parliament passed a law banning unmarried migrant workers from living in residential areas. [**?**]

Even looking at the United Kingdom, which has numerous laws that demand racial equality, due to a large number of cultural differences between communities, racial segregation has emerged in some parts of the country. The separated communities are largely representative of Indians, Pakistanis, and other subcontinentals. The segregation is thought to have occurred due to the ethnic tension, and deterioration of the standard of living, levels of education and employment among ethnic minorities in poorer areas. This type of racial segregation can be noticed in particular in residential areas. There is an indication of a market preference amongst the more wealthy to reside in areas of less ethnic mixture.[**?**] This is something that we are going to analyse: how minor local preferences can lead to large segregation patterns in the system. Thus, we will get an understanding of just why it is so hard to combat racial segregation.

2.2.2 Computational models in general

We are surrounded by complex systems, from atoms which spontaneously form molecules to plants, animals, fungi, physical conditions etc. which form ecosystems and to humans that form societies. Social scientists have always been interested in solving the mysteries of these systems, in analysing their behaviours. However, these are in most cases complex, nonlinear systems which cannot be simply analysed using simple-logic or other intuitive analytical solutions. The evolution of computers opens new doors for us, allowing us to experiment on a model with different variables. This allows us to simulate a wide range of scenarios, avoiding the need of deriving a mathematical analytical solution. A computational model uses a large number of variables with each representing a characteristic of the system.

Accelerating scientific discoveries through computational modeling

Computational modelling allows scientists to run thousands of simulated- experiments in a relatively short amount of time; in many cases they are run simultaneously. This way, researchers can quickly identify the physical experiment that is most likely to help them to solve the problem they are working on.

Lets look at an example so we could understand better how computational modeling can save not only time, but also resources when trying to find a new solution or to improve an existing one. Assume that a bakery is trying to come up with a cake composition that would be dense enough to support a multiple level cake for special occasions like weddings. They want it to be solid enough so it would not fall during transportation and keep all the ornaments attached in place, but they do not want to compromise the taste of the cake and they still want a soft middle part that their customers enjoy.

To try to create a new composition by physical experiments is expensive in terms of time, resources etc. Instead, they could use a computational model simulation. They just need to add all the ingredients with their measurements as variables and provide details about what each ingredient does and how it interacts with the others. In minutes, they can run hundreds of simulations by just changing the amount of each ingredient, the time, heat or any other variable. This would take days to try physically. Once they find a potential candidate (the right amount of ingredients, heat and time), they can run this single physical experiment and check that it works.

Examples of computational models used to analyse behaviour in complex systems around us

We use computational modeling to get information about the complex systems that surround us. Being informed not only can help us have a better, more comfortable life, but in so many cases it helps us save lives. Here are some examples:

- We all like to know how the weather is going to be, either for planning our next holiday or to know what kind of clothes would be adequate. **Weather forecasting** is our first example that uses computational models to analyse and make predictions based on many atmospheric factors.

- Next, we have **earthquakes studying**. In order to save lives and to build better constructions, engineers are using computational models to simulate how different materials, structures and surfaces they are building on would interact in case of an earthquake.

- And a final example would be almost every **item in our homes**. Manufactures use computational models during the development of their products, from food to packaging of household chemicals, to clothes and shoes (textiles). To design all these they use many complicated mathematical formulas and modeling tools.

2.2.3 Agent-based models

Agent based model is one class of computational models used to simulate the actions and the interactions of autonomous agents. In most cases, scientists are looking on how this actions affect the system as a whole.

Why use a agent-based computation in social sciences?

Agent based computational modeling has had a huge impact on social sciences. This new technique allows scientists to analyse a 'virtual' society of individual agents: rational, heterogeneous agents (in most cases) represented by software objects. Agent based models allows analysing complex systems of varied domains from economics to ecology. It is a popular model among these domains since they include problems defined by many individuals, giving rise to emergent patterns. It is also important to see how these emergent patterns that appear because of the individuals' demands, constrain the decisions of the individuals.

Agent-Based Computation: Pros and Cons

Robert Axtell in his paper *Why agents? On the varied motivations for agent computing in the social sciences* [?] describes the agent-based computation followed by the strength and weaknesses of this approach.

Most commonly, the individual agents of an agent-based model are represented in software by objects. As objects, they have states and behavioural rules. To run the system, all that needs to be done is to instantiate some agents (creating an agent population) and let them interact, monitoring what happens. In order to get a solution, one can simply spin the model forward in time. An important point is made here, once we created an agent-based model, say A, executing the model and getting to a result, say R, one can formulate the following sufficient theorem: *R if A*.

A first advantage of agent-based computation is the fact that we can limit the rational agents, which are the ideal type, in agent-based computational models. We can create each of the agents with different attributes, different (or random) behaviour. However, even if one would prefer to analyse the system using just rational agents, it would be easy to implement heterogeneous agents. One can simply instantiate a population with a certain preference to begin with.

Another advantage is the fact that one can analyse the entire history of the process in the study since the model is "solved" by executing it. It is easy to run and there is no need to focus only on the equilibria. A final advantage mentioned here, is the fact that for running a computational model, there is no need of physical space and social networks can be computed.

A major disadvantage of agent-based computational modelling compared with the mathematical modelling is the fact that even if each run of a computational model yields a sufficient theorem, a single run provides no information on the robustness of such theorem. Thus, even if agent model A implies R, we have no information on how much A can be changed and still getting R as a result. Using mathematical methods, these type of questions are often formally resolved.

Agent-based computational models classification in terms of uses

Finally, lets have a look on how agent-based computational models can be categorised. Robert Axtell argued that there are three distinct uses for such model.

- First, when we have a numerical realisation, agents can be used to solve classical simulations. These simulations can act as a verification that our mathematical methods are correct, but they can also be used to test and build more sophisticated agent models. We normally start with constructing agent-based models to which we know the solution from a mathematical model. This way we can test the program. We start with ideal agents, check that the results are right and then move to non-ideal or more sophisticated agents.
- Second, when a model cannot be solved completely mathematically (incompletely solved), agent-based model can be used as a complement to mathematics.

- Finally, third, when mathematical models are apparently intractable or even proven insoluble. In these cases, a agent-based approach is the only technique that could be used to analyse the system.

2.2.4 Multi-Agent Bargaining Model

Later we are going to discuss in detail the multi-agent Schelling's model. However, in this section we are going to have a look at the multi-agent bargaining model. This would help us get a better understanding of how agent models work. Furthermore, we are going to look at how classes can emerge even when starting with a classless, norm-free initial world. This will help us understand how the segregation works in Schelling's model. We will see this in detail later on.

Robert Axtell, Joshua Epstein and Peyton Young presented the multi-agent bargaining model in 2001 in *The Emergence of Classes in a Multi-Agent Bargaining Model.* [**?**]

Motivation of Bargaining Model

Norms are part of our everyday life and they govern most social interactions: from dress code, to table manners and forms of communication. The *The Emergence of Classes in a Multi-Agent Bargaining Model* paper is focusing on the distribution of property norms. These can be divided into two big categories: *discriminatory norms* and *equity norms.* In this context, discriminatory norms would allocate different shares of the pie according to gender, ethnicity, religion, age, etc., while equity norms does not discriminate. The paper focuses on how these discriminatory norms occur from a norm-free initial world. It is important to understand why and how the discriminatory norms emerge since they have a significant impact in economy, leading to big differences in economy class. The multi-agent bargaining model combines evolutionary game theory with agent-based computational modeling to show how norms can emerge spontaneously at social level. These norms are thought to emerge simply from the decentralized interactions of numerous agents cumulated over time in a set of social expectations.

Bargaining Model Overview

The model is based on Young's evaluational model of bargaining (Young 1993). The model starts with a norm-free class. The norms emerge from the decentralised interactions of agents. At each time, there are two randomly picked agents that interact, bargaining over parts of their property.

There are no expectations to begin with. Agents develop expectations and behaviour based on previous interactions. People have "tags" that characterize them. These tags are not economically or socially significant to begin with, they are simply distinguishing features, such as brown or blue eyes, light or dark colour. However, over time, because of path dependence effects, they might acquire social significance. It might happen that agents with brown eyes get a larger part of the share due to some coincidences. But, since agents develop their expectations based on previous experiences, it might be the case that now agents with brown eyes will always get a larger part. Hence, a *discriminatory norm* emerged. However, *equity norms* can develop that do not take into consideration any tags when bargaining.

The authors argue that an equity norm is more stable than any discriminatory norm. Based on numerous realizations of agent-based computational models, they estimated that a waiting time that is exponential in memory length and the number of agents would be enough to ensure that the society is near an equal sharing regime.

Bargaining

We said before that at any given time, we have two individual agents bargaining over shares of their property. Now, we are going to look briefly at what the bargaining process involves.

Given two agents, A and B, each demands a portion of the available property. To explain the process we do not care about the property type, but we assume that it is divisible. Once, each of the agents made their demand, they use the *Nash demand game* to decide what each one of them gets. They both get their demanded parts if the sum does not exceed 100 percent of the available property, otherwise they both get nothing.

To make it easier to analyse, we assume that each player can make one of the three demands: low (30%), medium (50%) or high (70%). The following table shows the Nash demand game for these 3 values.

	H	M	L
H	0,0	0,0	**70,30**
M	0,0	**50,50**	50,30
L	**30,70**	30,50	30,30

We notice that there are three Nash equilibria, shown in bold: (L,H), (M,M) and (H,L). However, Axtell, Epstein and Young, do not assume equilibrium, but they are trying to explain the process by which the equilibrium emerges at the aggregate level, from the repeated, decentralized interaction of the agents. This is what we are interested in as well when it comes to Schelling's segregation model: we want to see how we can reach a non-segregated final configuration (an equilibrium).

The model with One Agent Type vs Two Agent type

We look at the two models presented in the paper:

- One Agent Type Model - all agents are the same (indistinguishable from one another), but they have different experiences that conditioned their belifes.
- Two Agent Type Model - the agents differ from one another by one visible "tag", like light or dark skin, or blue and brown eyes. These tags have no economic or social significance to begin with. This is a similar model with what we are studying in Schelling's model.

Analysing the results of these two models, we can conclude that in the Two Agent Type Model long-lived discriminatory norms develop purely by historical chance, while in the homogeneous model there are no discriminatory norms. In other words, various kinds of social orders, like segregated, discriminatory, and class system, can arise in a Two Agent Type model, simply by the decentralised interactions of many agents.

We will see that same segregation is expected in Schelling's model.

2.3 Schelling's Model Impact

In 1971, Thomas Schelling created an agent-based model that helps to explain why segregation occurs and why it is so hard to combat. In his model, agents choose to change those with whom they associate, rather than changing their behaviour to meet the behaviours of their associates. In other words, an agent would choose different neighbours rather than conforming with the existing ones.

Schelling's segregation model is probably one of the most popular segregation models. Although it has been appreciated by many, there are a few who criticized the paper and the models presented, and here we could mention Yinger [**?**]. Yinger argued that the models were not realistic and they could not have been applied in any real-world situation because Schelling did not consider some important factors, such as economics and social mobility. Although these are good arguments and in order to create a realistic model, to reflect modern society, we need to take these characteristics into consideration. What Shelling showed in his paper is that the avoidance of being a minority-class in society could lead to major segregated systems. Therefore, many have understood what Schelling tried to achieve and used the simplistic model in order to try and build more sophisticated ones.

In the next part we are going to look at Peyton Young's paper, *The Dynamics of Conformity*[**?**] to see a similar model to Schelling's linear model. Young shows that spatially homogeneous patterns (segregated patterns) are much more likely to occur than heterogeneous patterns in a long run. Then we are going to look at a couple of different papers which, starting from Schelling's model, describe various segregation problems and patterns, such as segregation based on the religion in different cities.

2.3.1 The Dynamics of Conformity - Peyton Young

Similarly with Schelling's model, Peyton considers a population that is consistent of two types of agents, As and Bs. They could represent different genders, races, religions, ethnicity etc. Each agent chooses whether it wants to associate with As or Bs or a mixture of the two. In other words, each agent has a preference for the composition of the neighbourhood he lives in. He wants a certain proportion of As and Bs, respectively.

Young makes the following assumption: each agent prefers to leave in a neighbourhood that has some people of the same type as oneself, although they might not want to live in a neighbourhood where everyone is like oneself. Another assumption is that everyone has the same utility for the proportion of people that are same type as oneself.

Same as in Schelling's setup, people move from a neighbourhood in which they are discontent to a neighbourhood where their demands are met. An agent's decision to move would have external effects since it would typically alter the composition of the neighbourhood he is moving into and of the neighbourhood he is leaving (moving from).

Another assumption that Young makes is that the agents are located around a circle and that we have an even number of agents of each type. Furthermore, he assumes that each agent cares about the type of the agent on his right and the agent on his left. In other words, a local neighbourhood for an agent consists of the two closest people to that agent, one at his right and one at his left. As we are going to see later, this is a limitation of Schelling's linear model, which analyses the model with various length sizes for an agent local neighbourhood.

A *state* of the system is an assignment of A or B for each position. A state is *completely segregated* if all the As form one contiguous group and all the Bs form another contiguous group. Finally, we say that a state is *completely integrated* if each person lives next to exactly one person of the same type as oneself and one person of different type. There are plenty of intermediate states between the two extremes. In the figure below, we can see a completely integrated state on the left and a completely segregated state on the right.

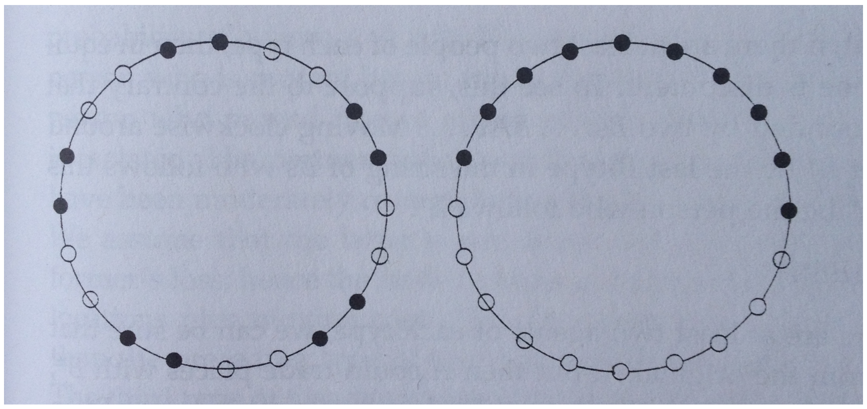


Figure 1: Completely integrated and completely segregated states

In Young's model, an agent can be in one of the following three states:

- *discontent* - if both neighbours are not like oneself
- *moderately content* – if both neighbours are like oneself
- *content* - if one neighbour is same type as oneself and the other neighbour is of different type

Young introduces a different rule to determine the agents' movement than what we are going to see in Schelling's model. Agents trade their places, but there is a cost associated with the movement so there is no incentive to trade unless the overall level of contentment is raised. Between two randomly selected agents we have an *advantageous trade* if both individuals gain from trading places, otherwise we have a *disadvantageous trade.* It is clear that we are talking about an advantageous trade if the trade is happening between two agents of different type.

Young makes the following affirmation: from any initial configuration, there always exists some sequence of improving trades that leads to equilibrium. Furthermore, he states that the particular equilibrium that is reached depends on the order in which people have traded their positions, it is path dependent.

In Schelling's model, agents have perfect logic, that is, they would not move if they were content or their position would not improve after moving. However, in Young's linear model we are introduced to agents who might make a trade for unexplained reasons. Young shows that if the probability of making idiosyncratic trades is small enough, the most likely form of spontaneous order is a completely segregated order. He states that "the segregated states are the only stochastically stable states". Schelling's derives a similar result using a slightly different model.

What is important to notice in Young's model is the fact that we started with a random distribution where nobody wanted to live in a neighbourhood made 100% of people like oneself. In fact, everyone would prefer a perfectly integrated society. But, as we see from the results of the paper, segregated patterns are more likely to occur due to the cumulative impact of many agents which make a trading decision is locally optimal, but not globally so.

2.3.2 The Schelling Model of Ethnic Residential Dynamics: Beyond the Integrated - Segregated Dichotomy of Patterns

The Schelling Model of Ethnic Residential Dynamics: Beyond the Integrated - Segregated Dichotomy of Patterns paper [**?**] tries to use the Schelling model to reproduce the ethnic residential patterns of Israeli people. The paper brings into the equation a new type of model pattern in which only a part of the group will segregate, while the rest will remain in their initial location even if their are discontent. All these patterns are compared with the patterns in the real cities, presenting and discussing the differences that occur.

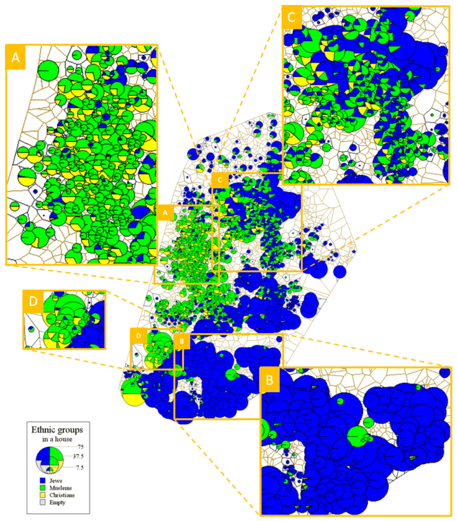


Figure 2: Ethnic residential distribution of Yaffo in 1995

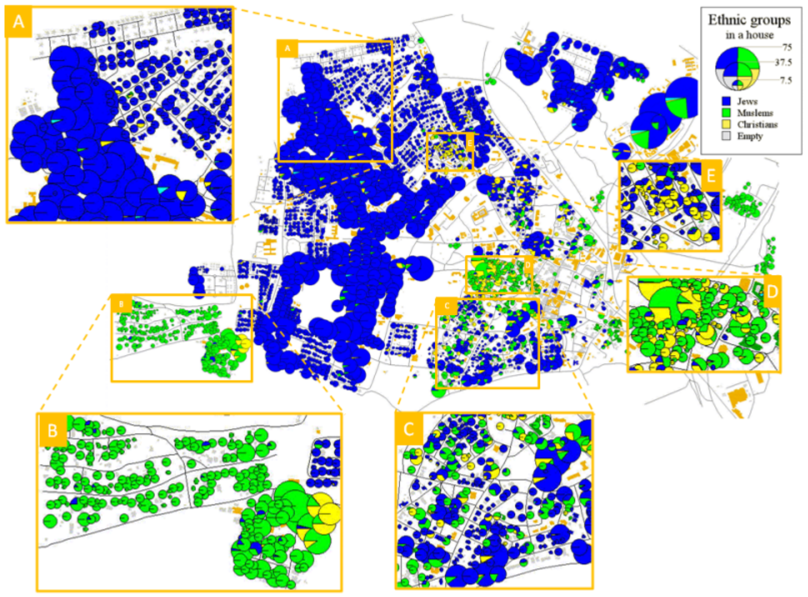


Figure 3: Ethnic residential distribution of Ramle in 1995

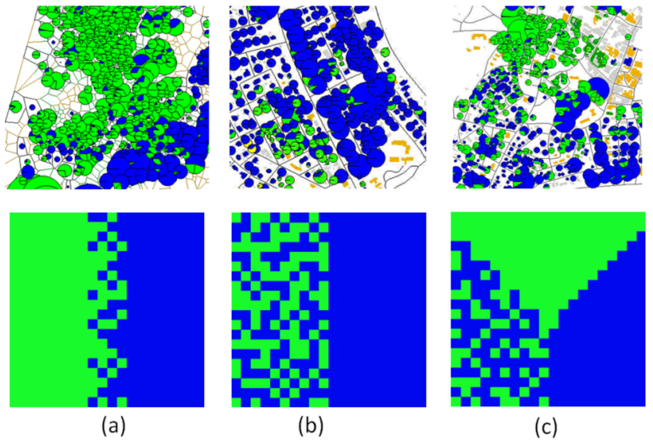


Figure 4: Comparison of actual ethnic residential distribution with the model pattern

2.3.3 A Wealth and Status-Based Model of Residential Segregation

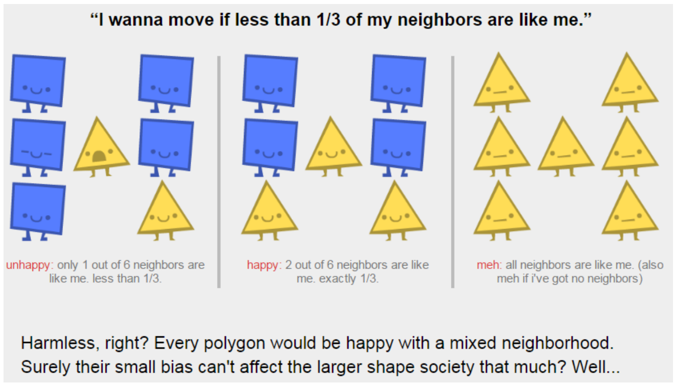
Stephen Benard and Robb Willer in *A Wealth and Status-Based Model of Residential Segregation*[?] present an extension of Schelling's model by introducing some new factors to the model: the wealth and status of the agents and the desirability and affordability of the residences. The authors first look into how correlated the wealth and status are and how does the wealth of residents affect the affordability in that neighbourhood.

The results of the study show that the greater the correlation between wealth and status, the higher the chances of segregation, that is, little to no mixing between people with different wealth and status level in the final configuration. Furthermore, the change of the prices of the houses acts as a precondition for the status segregation.

2.3.4 Parable of the Polygons

Segregation is an important problem in present society and numerous scientists and designers have realised how important is to educate the population from a young age if a change is to be made in future. They designed different games to help students understand how segregation happens and how even a small local bias can lead to a global segregated society.

Parable of the Polygons [**?**] is one of such playable games of Thomas Schelling's model of neighbourhood segregation. The creators of this game saw the importance of understanding Shelling's model in order to deal with segregation. They created a fun, easy to use game in order to help others understand how a small, personal bias could have a large-scale impact, over a whole society.



"I wanna move if less than 1/3 of my neighbors are like me."

unhappy: only 1 out of 6 neighbors are like me. less than 1/3.

happy: 2 out of 6 neighbors are like me. exactly 1/3.

meh: all neighbors are like me. (also meh if i've got no neighbors)

Harmless, right? Every polygon would be happy with a mixed neighborhood.
Surely their small bias can't affect the larger shape society that much? Well...

Figure 5: Helping the user understand how your personal bias might affect the larger neighbourhood

There are two very interesting points that are covered in this article. Firstly, even a small 33% bias could have a significant impact over the society. Secondly, being in an already-segregated society, it turns out that it is almost impossible to go back to a mixed neighbourhood even if everyone would lower their preferences. The following figure shows that even if people lowered their expectation from 33% to 10%, nobody moves. `i.e` Being a segregated society to start with, even if we all stop having any kind of ethnic bias, it will still take a long time to go back to a mixed distribution, if ever.

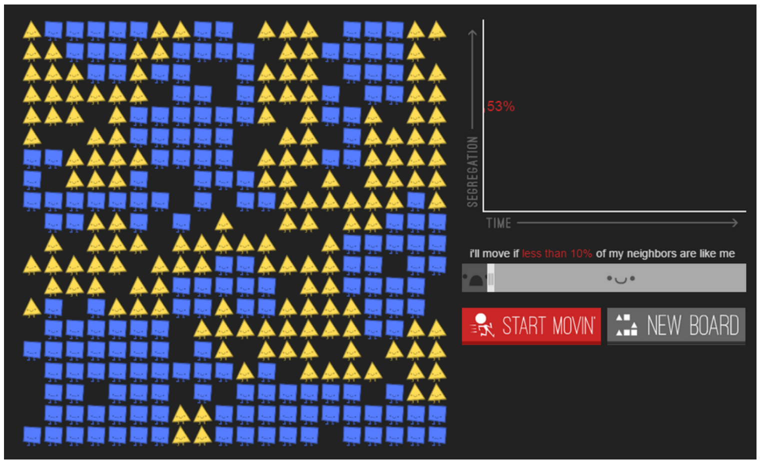


Figure 6: A 33% preference lowered to 10% still does not change the distribution

2.4 Development environment

So far, most multi-agent computational models out there are modeled using object oriented programs. Each agent's characteristics and behaviour are implemented in a class and an agent is an instantiation of the class. The environment rules (in our case would be the turn function) is defined in a different class. The complication with a model designed this way is the *randomness* of the program. For example, in our model we would have to define a strict turn function. We are going to see later how same agent behaviour, but different turn functions can affect the final, resulting configuration. In other words, we get different outcomes depending of which player gets to move first. One could, of course, create a random turn function where each time a random agent is picked from the crowd. However, in normal object oriented programs it is complicated to ensure that everyone is being picked infinitely often.

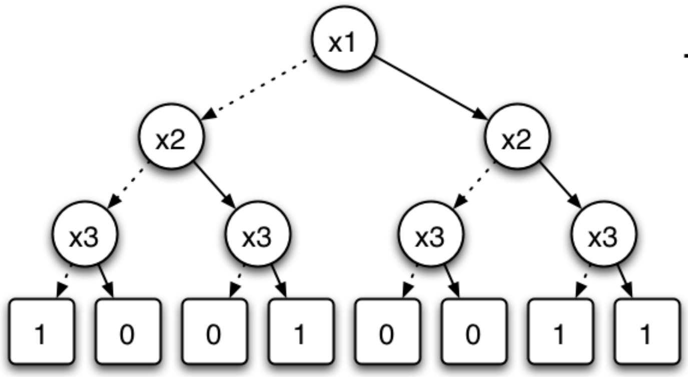
In order to automatically test Schelling's linear model, and variations of the model with random turn functions we used **NuSMV**. To understand what *NuSMV* is and how it works, we first have to talk about *SMV* model checker, *Binary Decision Trees* and *Binary Decision Diagrams.*

SMV - Symbolic Model Checker

When we refer to a **model checking** we talk about the following problem: Given a model of the system `M` and a property (or specification) `P`, we want to *automatically* and *exhaustively* check that `M` meets `P`. In other words, model checking is a technique to check automatically that a specification is met in a *finite-state* system. The specifications about the system must be expressed as temporal logic formulas. Then, we use symbolic algorithms to traverse the model and check if the specifications work or not.

Binary Decision Trees (BDTs)

Binary Decision Trees (BDTs) are a way of representing Boolean formulas just like truth tables. A BDT is a rooted, directed, acyclic graph which consists of several decision nodes and terminal nodes. Terminal nodes are labelled with 0 or 1 and all the non-terminal nodes are labelled with a Boolean variable. Each non-terminal node has exactly two outgoing edges: one labelled 0 and the other one labelled 1. *Figure 7* shows an example of truth table and its corresponding BDT for the formula $(\neg x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (x_1 \wedge x_2) \vee (x_2 \wedge x_3)$. By convention, *dashed* lines are used to represent that we assigned false to the variable and *continuous* lines for when we assign the variable to true.



x1	x2	x3	*f*
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Figure 7: BDT and Truth Table for formula $(\neg x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (x_1 \wedge x_2) \vee (x_2 \wedge x_3)$ [**?**]

BDTs are preferred over Truth Tables since they are more compact and also, more importantly, performing a Boolean operation on the formula means that we have to recompute the entire truth table while we can do operations directly on BDTs rather quickly.

Binary Decision Diagrams (BDDs)

Binary Decision Diagrams (BDDs) are more compact data structures than BDTs, also used to represent Boolean formulas. We can apply the following steps on a BDT to reduce the size and produce a BDD:

- Sharing the terminal nodes of the BDT. In other words, we will have only two terminal nodes: a 0-node and a 1-node.
- Remove redundant decision points. That is, if both outgoing edges of a node `e` point to the same node `n`, remove `e`.
- Remove duplicate non-terminals. If two distinct nodes of the BDD are the roots to two structurally identical subBDDs the remove one of the nodes (and its subBDD) and send all the incoming edges to the other one.

After applying all the steps above to the BDT, the result will be a *Reduced* BDD.

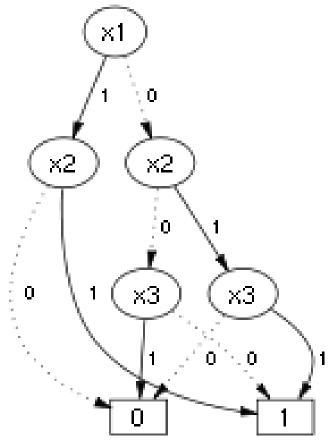


Figure 8: ROBDD for formula $(\neg x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (x_1 \wedge x_2) \vee (x_2 \wedge x_3)$ [?]

Figure 8 shows the resulting Reduced BDD for the above formula $(\neg x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (x_1 \wedge x_2) \vee (x_2 \wedge x_3)$.

We say that a BDD is *ordered* if different variables appear in the same order on all paths from the root. It is important to note that we can represent sets and functions between sets as Boolean formulas. Hence, OBDDs are used to perform symbolic model checking since we can create a OBDD representing a Boolean formula. Functions and operations on sets can be executed on the OBDD. Another important property of a OBDD is the fact that their reduced structure is unique.

NuSMV Model Checker

A *model checker* is a tool that provides a programming language to describe the model succinctly and it performs the check automatically against some given specifications.

NuSMV is an extended version of the symbolic model checker `SMV`. The input language of NuSMV is simply called SMV and the properties that we want to check in SMV are formulae in LTL (Linear Temporal Logic)[**?**] and CTL (Computational Tree Logic) [**?**]. Given the model as an input and a property, NuSMV will return true if the property holds in that model, false otherwise. If the answer is false, NuSMV generates a counter-example.

NuSMV is based on Binary Decision Diagrams (*BDDs*).This was a suitable tool for the task because for any given initial configuration it would generate a binary decision diagram generating this way all the possible final configuration, considering all the possible turn functions. Furthermore, we can introduce *Fairness constraints* such that each agent is going to be picked infinitely often. It allows us to answer questions such as "Does the configuration converge for all possible turn functions?" or "How segregated the final configurations are?". We are going to explain what all these mean and why they are important in the *Formal Analysis* chapter.

Python

A downside of the SMV language is the fact that it does not support control flow statements, such as *for-loops*. Therefore, in order to generate the model in an `smv` file, we used a Python [**?**] script. This way, passing in the initial configuration, we can create the model of the system in SMV, together with the properties that we want to check. Thus, the generated file can be run using NuSMV. The SMV model of the system is explained in detail in the *Simulation* chapter. There, we explain how the modules are generated and how they work.

3 Schelling's Model

3.1 Schelling's Dynamic Models Of Segregation - Overview

Racial segregation has always been a major social problem in the world and especially in the US. Despite all efforts, racial and economical segregation is still around us. But why is it so hard to eradicate segregation?

In 1971, in the *Journal of Mathematical Sociology*, the American economist *Thomas C. Schelling* published a paper on *Dynamic Models of Segregation*. He designed a simple agents-based model in order to explain why segregation is such a hard problem to combat. He showed that even if individuals (or agents) did not mind living in a mixed neighbourhood, they would still choose to move/segregate over time. Schelling did not use a very sophisticated model, but the output of his study was surprising: although individuals had no explicit desire to self-segregate, they would still do so. The model focuses on residential segregation of ethnic groups. It is a *two-type agent model*. That is, the agents are of two types: white or black. This is the only characteristic that distinguishes the agents. In any other aspect, the agents are identical.

3.2 How does the model work?

Now, we are going to explain the two main models that Schelling used. We are going to start with the Spatial Proximity Model and then have a brief overview of the Bounded-Neighborhood Model. For the Spatial Proximity Model, Shelling starts by putting the population in a simple model (the linear model). Next, he organises the people in a two-dimensional area. The general idea is displayed in the linear model as well as in the two-dimensions model. In this project we are focusing on the linear model since we are interested in automatically testing this model, using a computational multi-agent model.

3.2.1 Spatial Proximity Models

In this model, everybody defines their neighbourhood by reference to his own position. An individual who is *unhappy* with his neighbourhood, *i.e.* he is not content with the mixture of his neighbourhood, would try to find a place, and move to that place, where his demands are satisfied. Hence, for each agent what matters is the colour ratio in his own neighbourhood. Schelling looks at what distributions of tolerance among individuals may result in a stable mixture. He looks at how different initial conditions and dynamics of the movement will impact the outcome and if there are any numerical constraints that could alter the results.

For these models, we are going to use two types of agents to help us represent these models: $\circ$ - representing a white individual and $*$ - representing a black individual.

Linear Distribution

The first spatial model, is a model where agents distribute themselves along a *line*. Let us take for example a random distribution of 12 ∘s and 12 ∗s:

∗	∗	∗	∘	∗	∘	∗	∘	∘	∘	∘	∗	∗	∘	∗	∘	∘	∘	∗	∗	∘	∘	∗	∗

Random Linear distribution - 12 whites and 12 blacks

Shelling interprets this reasonably random distribution as people spread in a line, each concerned with the colours of the people in their neighbourhood. Assume that each individual defines his neighbourhood as himself and the two individuals at his right and two individuals at his left. One of the assumptions that Schelling makes regarding the satisfaction of an individual in his neighbourhood is that half of the neighbours of the individual to be the same colour as himself. *e.g.* In our case, with a neighbourhood of 4 people (excluding himself), an individual would be content if at least 2 people are the same colour as himself. In special cases, like when the individual is at the end of the line, the two neighbours on the side towards the central plus maybe one outboard neighbour would represent the local neighbourhood and hence at least half (one out of two neighbours or two out of three neighbours) must be the same colour.

Looking at our random distribution, we can see that, according to the above rule, for a neighbourhood of size two (4 people in total), 10 out of the 24 agents are unhappy (unhappy agents are on positions: 4, 7, 12, 14, 15,19, 20, 21, 22, 23). Here, we put a dot over the unhappy agents.

			·			·					·		·	·				·	·	·	·	·	
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
∗	∗	∗	∘	∗	∘	∗	∘	∘	∘	∘	∗	∗	∘	∗	∘	∘	∘	∗	∗	∘	∘	∗	∗

The dot represents a discontent agent

The next step was to define a rule about how the individuals can move. A discontent person moves to the nearest position where he would be satisfied (at least half of his neighbours are like oneself). Hence, he is looking at passing the minimum number of individuals and intrudes himself between two others when he gets to a place where he is happy (satisfied). The order in which they move is set alternating turns from left to right. *i.e.* the first person to move would be the ∘ on position 4 followed by ∗ on position 23 and so on. Once they start moving, we notice that some people who were discontented become content while other which initially were content will become discontent. The rule is that if an initial unhappy individual becomes happy by the time his turn comes, he will not move. After going through all the initial unsatisfied individuals, any other individuals that became unhappy due to the movement of the previous agents, they will have their turn to move.

After one step, the distribution looks as follows. Agent 4 moves to position 6.

				·		·					·		·	·				·	·	·	·	·	
1	2	3	5	6	4	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
*	*	*	*	○	○	*	○	○	○	○	*	*	○	*	○	○	○	*	*	○	○	*	*

Notice that Agent 6 that was initially content is now discontent. The next to move is Agent 23. The nearest position for Agent 23 is swapping place with Agent 24. This makes Agent 24 discontent.

				·		·					·		·	·				·	·	·	·	·	
1	2	3	5	6	4	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	24	23
*	*	*	*	○	○	*	○	○	○	○	*	*	○	*	○	○	○	*	*	○	○	*	*

Next to move is Agent 7. Notice we skipped Agent 6 since he was not discontent to begin with.

											·		·	·				·	·	·	·	·	
1	2	3	5	7	6	4	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	24	23
*	*	*	*	*	○	○	○	○	○	○	*	*	○	*	○	○	○	*	*	○	○	*	*

The discontent agents continue to move according to Schelling's turn function rules, until we reach the following final configuration.

1	2	3	5	7	6	4	8	9	10	11	13	12	15	14	16	17	18	22	21	19	20	24	23
*	*	*	*	*	○	○	○	○	○	○	*	*	*	○	○	○	○	○	○	*	*	*	*

The result is five clusters of like individuals where every agent is content.

Area Distribution

Looking into two dimensions space, having a relative position is not something that we could implement easily. In the linear model, when an agent was discontent, he was moving from a position in between two individuals to a position between two other individuals, *i.e.* relative position.

For the Area Distribution, Schelling divides the space into rectangle spaces, each agent occupies one rectangle space. Hence, in the model there are going to be rectangle spaces occupied by agents, be them black ($*$) or white ($\circ$), or they will be *vacant*. An agent, is able to move only to a vacant square and when he moves he leaves a place vacant. Similar to the linear model, he would move only if he is "unhappy" in his neighbourhood. Hence, the model's basic assumption is that an individual located in the middle of the neighbourhood that has less than `p` percent individuals like himself, will try to relocate to a neighbourhood for which this percentage is met. The percent, `p`, for which the agents are satisfied is a predefined tolerance threshold. The higher this percentage is, the *less tolerant* the agent is.

Let us now look at one of Schelling's random distribution model (Table 1). There is an equal number of whites ($\circ$) and blacks ($\ast$) and 70 vacant places. There are 25 $\ast$ and 18 $\circ$ that are discontent (have neighbours less than half of like colour). We also can check that $\circ$s on average have 53% neighbours of the same colour and $\ast$s have 46%.

○	∗	∗	∗	∗	○	○	○	○			∗	∗		○	○
○		∗	○	○	○		∗	∗			∗		∗	○	
∗		∗	○	○	∗	∗		∗			○	∗		∗	∗
∗			∗	∗			○	∗	∗	∗	○	○			
○		○	○	∗	∗	∗	∗			∗		∗		○	○
	∗	○	∗		○	○	∗			○	○			∗	∗
∗	○	○	∗					○	○	○	∗	∗	∗		
○		∗	○		∗	∗		∗	○	○	○				∗
○		∗	○					∗	∗	○					∗
○	○				∗			○	∗	○	○	○	○	∗	∗
	○	∗	∗	○	○	○	○		○	∗	∗		○	∗	∗
∗		○	∗	○	∗		○	○	∗	○	∗	○		○	
	○	○			○	∗	○	∗	○	○	○			∗	∗

Table 1: Initial condition of one of Schelling's experiments

In most of Schelling's examples, the neighbourhood of any agent **A** is defined as the 8 surrounding square. See *Table 2*.

N	**N**	**N**
N	**A**	**N**
N	**N**	**N**

Table 2: Neighbourhood

As in the linear model, a demand and a moving rule must be established. For this example, we have the same demand as previously seen: no fewer than half of one's neighbours be of the same colour. Also, the individuals who are discontent will move to the nearest vacant place that satisfies their demands. However, it is more complicated than in the linear model to define a rule to specify the order in which they move.

Starting at the upper-left corner going downwards and to the right, an equilibrum is achive as shown in Table 3.

*	*	*	*	*					*	*	*	*		○	○
*	*	*	*	*	*	*	*	*	*	*	*		*	○	○
*	*	*	*	*	*	*	*	*	*	*		*	*	○	○
*			*	*	*	*	*	*	*	*			*	○	○
○	○	○	○	*	*	*	*	*	*	*		*			
	○	○	○	○	○	*	*	*	○	○		*		*	*
	○	○	○	○	○	○	○	○	○	○	○	*	*		
○			○		○	○	○		○	○	○				*
○			○							○					*
○	○							○		○	○	○		*	*
	○			○	○	○	○		○			○	○	*	*
		○		○			○	○		○		○	○		
	○	○			○		○		○	○	○			*	*

Table 3: Stable segregation - moving left-to-right, up-down

Moving from centre outwards, the following equilibrium is achieved (Table 4):

	*	*	*		○	○	○				*	*		○	○
		*	*		○	○		*			*		*	○	○
*		*			*	*	*	*	*	*	*		*		
*	*		*	*	*			*	*	*	○	○		*	*
	○	○	*	*	*	*	*			○	○	○			
○	○	○	○	*			*	○	○	○	○	*	*	*	*
○	○	○	○	○		*		○	○	○		*	*		
○			○	○		*		○	○	○	○				*
○	○	*	*				*	*	○	○	○				*
○	○	*	*	*	*			*	*	○	○	○	○	*	*
	○	*	*	*	○	○	○		*	*			○	*	*
		○		○	○		○	○	*	*	*	○	○		*
	○	○			○		○	○	○				○		*

Table 4: Stable segregation - moving from centre outwards

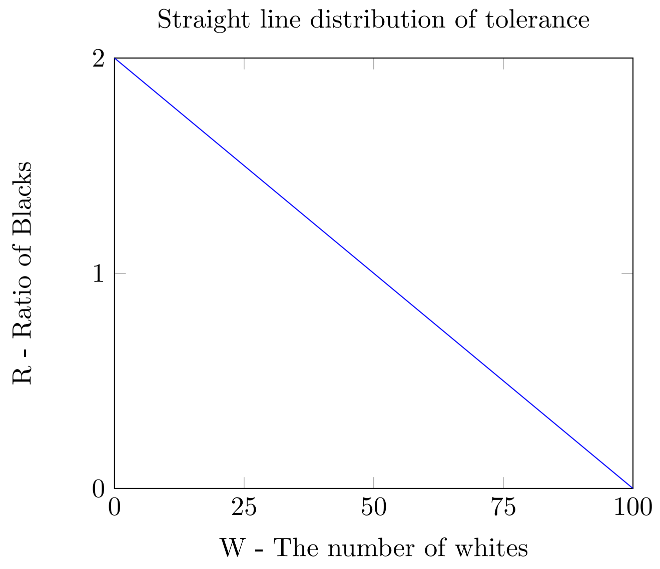
Although, the process started from the same initial random distribution, two fairly different stable states were achieved. While in *Table 3* the segregation is obvious (we could even talk about a complete segregation), in *Table 4* it is not so clear that the segregation happened. Hence, the outcome of the movement depends a lot to which order the individuals move. However, all Schelling is trying to show in the paper is that segregation is still happening, the outcome is always a stable segregation, regardless of the movement rules.

3.2.2 Bounded-Neighbourhood Model

Although a lot less popular than the spatial proximity model, bounded - neighbourhood model looks at a more realistic situation. It looks at neighbourhoods that have a limited capacity, like a school that can take a limited number of pupils or an office that has a certain number of employees.

Individuals do not define anymore their neighbourhood relatively to their location, but instead, there is a common definition of the neighbourhood (`i.e.` a school) and its boundaries. The individual is either inside or outside the neighbourhood.

Another significant change in this model is that each person has their own tolerance, instead of an identical tolerance. `i.e.` an individual might not be interested at all in the colour ratio in the neighbourhood that he is part of, while another(be he black or white) might only tolerate people of same colour like oneself. So the tolerance of each individual is defined as a straight line cumulative distribution plotted against the ration that the individual is willing to accept in the neighbourhood.

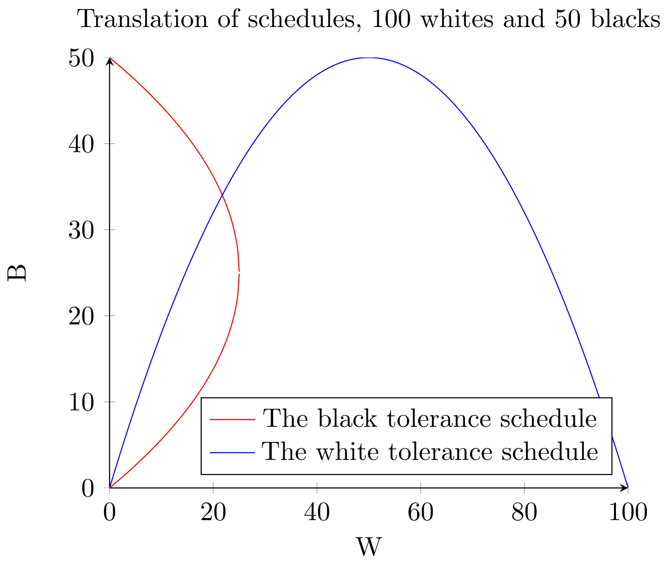


The above graph, represents the tolerance distribution of whites. The most tolerant white person can accept a black-white ratio of 2 to 1, while the least tolerant white person cannot stand the presence of any black people.

In order to study the dynamics in this system, Schelling makes the assumption that people both leave and return. People in the neighbourhood move out if a certain ratio is not met and people outside the neighbourhood move in if they see that their requirements are met. We notice that even if people are permitted to return to the area, they might never do so since their requirements might never be met. `e.g.` If an individual moved out because the cost of living in that location were high, by moving back to the same area the costs will probably still be the same.

Another assumption that is made is that everybody knows the ratio colour at the moment they make a choice. When it comes to moving rules, the assumption is that between two dissatisfied people of same colour, the one that is least tolerant moves out first. When it comes to moving in, the most tolerant one moves in first.

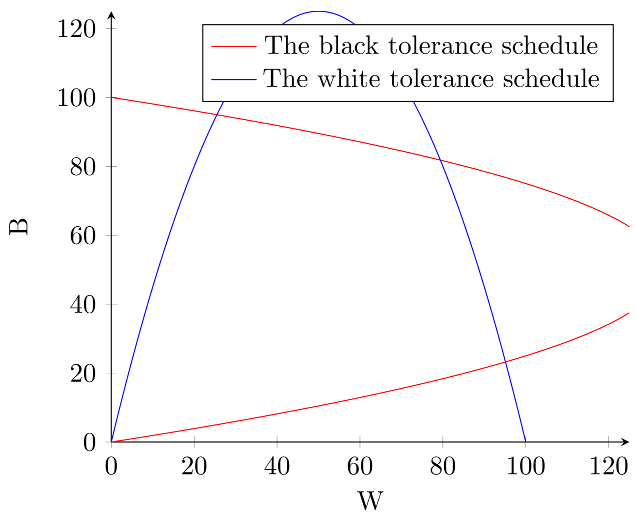
The straight line tolerance schedule can be translated into a parabolic curve as shown below.



In the above example, the number of blacks is half the number of whites. In the intersection of the white tolerance schedule and the black tolerance schedule, we have a good mixture of blacks and whites such that everybody is content. However, if we are under the white tolerance schedule curve, but not under the blacks curve, we are in a situation where all whites are content, but not all the blacks. Similarly for the blacks.

Another example Schelling presents, is having the same number of blacks and whites (100 each). This is represented below.

Translation of schedules, 100 whites and 100 blacks



These graphs allow us to interpret how the population changes within the area, the dynamics motion. For example, in the above graph, if we are located above the blacks tolerance curve, but below the white tolerance curve then we are in a situation where blacks are entering and whites are leaving because they are not content. On the right of the whites curve, below the blacks curve, we have whites coming in while blacks departing.

The area that is under both parabolas is a stable area. However, Schelling claims that at some point or another this "temporary stable equilibrium" will be disturbed since people who are in the neighbourhood will not leave, while people from outside the neighbourhood who would be content inside the area, will enter the neighbourhood. It is a matter of which colour is going to dominate, whether we are going up or right in our graph.

Hence, Schelling makes a very important statement and that is: "There are only two stable equilibria. One consists of all the blacks and no whites, the other all the whites and no blacks."

3.2.3 Tipping Model

In the final model, Schelling presents a combination of the two earlier models. By altering the entering rules, he tries to reproduce a phenomena that he says that was closely observed by *A.J.Mayer (1960)*[**?**], the "tipping phenomenon". Schelling claims that if a large enough minority group enters a neighbourhood, this will lead to "tipping", that is causing the initial residents to evacuate the area and we would end up with a neighbourhood with a majority of the opposite colour (or any characteristic we consider) than the initial one.

4 Formal Analysis

In this section, we are interested in giving a formal definition of the multi-agent line model. We are going to define the black and white groups and we are going to look at the agents on the line from a permutation perspective. We decided to do so since we are only interested to differentiate between the agents based on their colour: black or white. From any other perspective they are identical. Furthermore, we need to formally define what we mean by a *configuration*, what is a *turn function* and when do we say that an agent is *happy*.

In the second part, we are going to look at turn functions and their impact. We are going to describe Schelling's turn function in detail and look at some examples where different turn functions lead to completely different results.

Finally, we are going to look into two of the main characteristics of the linear model: convergence and termination. We are going to define these two terms formally and sketch some proofs.

4.1 General Definitions

`N` is a finite set of all players/agents. `B`, `W` are the sets of black and white agents, respectively. We assume that:

$$N = B \cup W$$
$$B \cap W = \emptyset$$

A permutation is an arrangement of objects in a specific order. Let us define the following permutation function:

$$\pi : N \to N$$

Let `P` be the set of all permutations.

Two permutations, π and π', are **similar** if for each player i:

$$\pi(i) \in W \Leftrightarrow \pi'(i) \in W$$

For example, the following two permutations are **similar**:

1	2	3	4	5	6	7	8	9	10	11	12
*	*	○	○	○	○	○	○	*	*	*	*

11	16	3	7	5	3	4	8	1	2	10	9
*	*	○	○	○	○	○	○	*	*	*	*

`C` is the set of *equivalence* classes under similarity.

$$\big[\pi\big] = \{\pi' \in P | \pi' \text{ is similar to } \pi\}$$

A **configuration** is just one element of `C`.

A **turn** function is a function $\tau : \pi \to N$ such that

$$\tau(\pi) = \tau(\pi') \text{ if } \pi \text{ is similar to } \pi'$$

Turn function associates to each permutation one player.

Given a configuration $[\pi]$ of length $\mathbf{n}$=$|\pi|$ and the neighbourhood size `k`, we define the left and right neighbourhoods of player i, fixing the permutation π. Notice here that the neighbourhood of player i does not comprise himself.

$$L_k(i) = \begin{cases} \bigcup\limits_{j=1}^{i-1} j & \text{for } i < k \\ \bigcup\limits_{j=i-k}^{i-1} j & \text{for } i \geq k \end{cases}$$

$$R_k(i) = \begin{cases} \bigcup\limits_{j=i+1}^{n} j & \text{for } i > n-k \\ \bigcup\limits_{j=i+1}^{i+k} j & \text{for } i \leq n-k \end{cases}$$

The whole neighbourhood of player i is:

$$N_i = L_k(i) \cup R_k(i)$$

A coloring is a function `color`: $N \to \{W, B\}$

and we say that $x \in W$ if $\texttt{color}(x) = W$

Let d_i be the demand of player i. Player i is happy if $i \in W$ implies that $N_i \cap W \geq d_i,\ d_i \in [0, 1]$.

$i \in B$ implies that $N_i \cap B \geq d_i$.

In general, we want that d_i is at least $\frac{1}{2}$.

This can be written as the union of black and whites in that neighbourhood:

$$N_i = W_i \cup B_i$$

A player is **happy** if at least 50% of its neighbourhood is the same colour as oneself. That is,

$$\pi(i) \text{ is happy if } \begin{cases} |W_i| \geq |B_i| & \text{for } i \in W \\ |W_i| \leq |B_i| & \text{for } i \in B \end{cases}$$

where $|A|$ is the cardinality of set A, `i.e.` the number of elements in A.

4.2 On the impact of turn function

In this section, we are looking at how configurations are affected by different turn functions. The final configurations we are looking for must be:

- *Stable* - We say that a configuration reaches a stable point if at some point in the future no unhappy people can improve.
- *Segregation problem is minimal* - We are looking for the least segregated final configuration. That is, when in a line of two types of agents, we have the largest number groups (white and blacks) alternating. For example, in the `Configuration 1` below we can distinguish 5 groups while in `Configuration 2` we have only 3 groups. Hence, `Configuration 1` is least segregated.

$*$	$*$	$*$	$\circ$	$\circ$	$\circ$	$\circ$	$*$	$*$	$\circ$	$\circ$	$\circ$	$*$	$*$	$*$	$*$

Configuration 1

$*$	$*$	$*$	$*$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	$*$	$*$	$*$	$*$	$*$

Configuration 2

4.2.1 Turn function in Schelling's model

Schelling's segregation model uses a specific turn function. We are going to explain how Schelling's turn function works step-by-step. Let us consider the following model so that we can demonstrate on this model each step of the way.

1	2	3	4	5	6	7	8	9	10	11	12	13
$*$	$*$	$\circ$	$*$	$\circ$	$\circ$	$*$	$\circ$	$*$	$\circ$	$*$	$\circ$	$*$

Let us take 13 players (6 of type $\circ$ and 7 of type $*$). The figure above shows us how they are distributed on the line. We assume that the neighbourhood size is 2, `n=2`. That is, every agent cares about the characteristics of the 2 neighbours on his left and the two on his right. And also lets make the assumption that each agent has a demand of 50%. Hence, they are happy if at least 50% of his neighbourhood are the same colour (have the same characteristics) as oneself.

Step 1: Given an initial configuration and a neighbourhood size, we first mark all the unhappy agents.

In our example, we have 4 unhappy players. We marked them with a dot.

		·	·			·					·	
1	2	3	4	5	6	7	8	9	10	11	12	13
*	*	○	*	○	○	*	○	*	○	*	○	*

Step 2: The initial unhappy players get to move according to the following rules:

- The unhappy players move one-by-one, alternating leftmost to rightmost, starting at the left-end. When their turn comes, the unhappy players move to the closest position that makes them happy. If there are two equidistant positions that would make a player happy, the player can choose between the two randomly.
- Any initially unhappy players, that became happy by the time their turn to move comes, they will not move. They will keep their current position.
- Any initially happy players, that became unhappy due to some other players movements will have to wait for the second round. In other words, they will get to move after everyone who was unhappy initially have moved.

Looking at our example, the initial unhappy players are: 3, 4, 7 and 12. These players get to move first in the order: 3, 12, 4 and finally 7.

The first player to move is the left-most unhappy player, that is the player on position 3. The nearest position that makes him happy is position 4

						·					·	
1	2	4	3	5	6	7	8	9	10	11	12	13
*	*	*	○	○	○	*	○	*	○	*	○	*

We notice that after the first player moved, the player on position 4 who was initially discontent, became happy as well.

The next player to move is the right-most unhappy agent, that is the player number 12. The nearest position that makes player 12 happy is between players 9 and 10. This will make the players 9, 10 and 11 unhappy.

						·		·		·	·	
1	2	4	3	5	6	7	8	9	12	10	11	13
*	*	*	○	○	○	*	○	*	○	○	*	*

Next, it is player number 4's turn. However, player 4 is now content so he will keep his position. Hence, the next player to move is player number 7. There are two equidistant positions to move that would make player 7 happy: between players 4 and 3, or between players 12 and 10. We will randomly pick for our example to move at the right, between 12 and 10

							·			·		
1	2	4	3	5	6	8	9	12	7	10	11	13
*	*	*	○	○	○	○	*	○	*	○	*	*

We finished moving all the initially unhappy players.

Step 3: Create a new list with unhappy players (new initially unhappy list). Start moving these players like in Step 2. Repeat Step 3 until there are no unhappy players, or any existing unhappy players cannot improve their position.

Our new list of unhappy players consists of players 9 and 10.

The first player to move is player 9. The closest position to make him happy is between player 7 and 10.

								·		·		
1	2	4	3	5	6	8	12	7	9	10	11	13
*	*	*	○	○	○	○	○	*	*	○	*	*

Notice, player 7 is now unhappy. The next to move is player 10. The nearest position that makes him happy is between players 12 and 7.

1	2	4	3	5	6	8	12	10	7	9	11	13
*	*	*	○	○	○	○	○	○	*	*	*	*

There are no more unhappy players so we stop. We reached a final, stable configuration which is clearly segregated. All the circles are clustered together.

4.2.2 Examples - Turn function impact

Even the turn function used by Schelling is not ideal. With the following examples we will try to explain just why that is the case and what kind of problems we face. Furthermore, we will show what happens when we slightly alter Schelling's turn function by changing the order in which players can move.

We make the following assumptions: the neighbourhood size is 2 (that is 2 on each side of the player) and that players move to the closest position (either left or right) that makes them content. A player is content if at least 50% of his/her neighbourhood are like oneself. We are trying to see the impact of different turn functions.

A first example we look at is the following:

·							·	
1	2	3	4	5	6	7	8	9
∗	○	○	○	○	○	○	∗	∗

We can see that players at positions 1 and 8 are unhappy. Choosing to move first player 1 we get the following:

2	3	4	5	6	7	1	8	9
○	○	○	○	○	○	∗	∗	∗

In this configuration, we reached total segregation. Everybody is happy. Hence, this is a happy case when we used Schelling's turn function.

However, with the same initial configuration

·							·	
1	2	3	4	5	6	7	8	9
∗	○	○	○	○	○	○	∗	∗

We could choose to move the player 8 first. He would move at the closest position that would make him happy. That is on the position occupied by player 9. The distribution would look like this:

·							·	
1	2	3	4	5	6	7	9	8
∗	○	○	○	○	○	○	∗	∗

Player 1 and player 9 are unhappy. We could choose to move player 1 and ending up with total segregation:

2	3	4	5	6	7	1	9	8
○	○	○	○	○	○	*	*	*

Or we could move player 9, leading to the configuration that we started with:

$\dot{1}$	2	3	4	5	6	7	$\dot{8}$	9
*	○	○	○	○	○	○	*	*

There is a cycle. Hence, depending on the turn function, the configuration might not converge.

Another example we could look at, has the following initial configuration:

1	$\dot{2}$	3	4	5	$\dot{6}$	$\dot{7}$	8	9	10	$\dot{11}$	12
*	*	○	○	○	*	*	○	○	○	*	*

Players 2, 6, 7, and 11 are unhappy. According to Schelling's turn function, we move first player 2, followed by player 11.

2	$\dot{1}$	3	4	5	$\dot{6}$	$\dot{7}$	8	9	10	$\dot{12}$	11
*	*	○	○	○	*	*	○	○	○	*	*

Next we move player 6.

2	1	6	3	4	5	$\dot{7}$	8	9	10	$\dot{12}$	11
*	*	*	○	○	○	*	○	○	○	*	*

The next player to be moved is player 7. However, here we face a problem with Schelling's turn function. There are two equidistant positions that would make player 7 happy. Hence, we have two possible final configurations:

1	2	6	3	4	5	8	9	10	7	11	12
*	*	*	○	○	○	○	○	○	*	*	*

1	2	6	7	3	4	5	8	9	10	$\dot{11}$	12
*	*	*	*	○	○	○	○	○	○	*	*

The latest configuration does not converge. It is not a stable configuration.

So far we saw examples that, depending on the order we move the players, the configurations might or might not converge. Look now at the following initial state:

	·							·	
1	2	3	4	5	6	7	8	9	10
*	*	○	○	○	○	○	○	*	*

Players on positions 2 and 9 are unhappy. Regardless of whom we choose to move first, the configuration does not converge. Hence, the turn function has no effect on this configuration.

4.3 Convergence

Convergence is an important characteristic of our linear model. We are interested in giving formal definitions for convergence and check whether or not it is the case that any configuration converges using Schelling's turn function. We are going to look at a couple of examples where the configurations do not converge and we are going to prove formally that for a specific turn function, termination will always occur.

Definition Convergence: We say that a configuration *converges* if, at some future point, it reaches a *terminal state*. A terminal state is a state where no unhappy player can improve his or her position.

4.3.1 Conjectures

Conjecture 1: Convergence of all initial configurations

We want to check whether it is the case that all initial configurations always converge using Schelling's turn function. However, Schelling's turn function is not ideal. We will have a look at one example where an initial configuration does not reach a stable point using Schelling's turn function.

Conjecture 2: Another approach to convergence

Another related conjecture to convergence that we considered is the following: For every configuration and any turn function, moving an unhappy person will not increase the number of unhappy people. Although this seems to hold and even Schelling makes some suggestions that this is the case, we will have a look at an example where this conjecture is falsified.

Conjecture 3: Termination using a specific turn function

Finally, we want to check the following: For every configuration there exists a turn function such that the configuration reaches a fixed point, `i.e.` there exists final configuration such that no unhappy person is able to improve. We will see in the next part that this holds if we relax the turn functions. In other words, with a specific turn function, we could always ensure termination.

4.3.2 Sketched Proofs

Proof 1: Convergence of all initial configurations

The following scenario does not converge to a fix point.

We will be using a turn function where an unhappy agent moves to the closest place that would meet his requirements, `i.e.` makes him happy. Also, assume for now that the neighbourhood size is 2. `i.e.` each individual will be interested in the colour of the 2 neighbours of his right and the 2 neighbours on his left. To be content, he wants at least 50% of these neighbours to be same colour as oneself.

Looking at the following distribution:

1	$\dot{2}$	3	4	5	6	7	8	9	10	11	12
$*$	$*$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	$*$	$*$	$*$	$*$

We have 6 whites, $\circ$, and 6 blacks, $*$, and the only unhappy person in this configuration is the individual $*$ on position 2. According to the turn function, he will move to the closest position in which he is happy. That would be position one. After the movement, we will have:

2	$\dot{1}$	3	4	5	6	7	8	9	10	11	12
$*$	$*$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	$*$	$*$	$*$	$*$

We can see that we are now back to the initial situation since all we did was swapping position of the first and second player. In this configuration, we have again the individual on position 2 being the only unhappy individual. According to the turn function, player $*$ on new position 2 will move to the closest position that will meet his demands. This is new position 1. Hence, we are entering an infinite cycle. Therefore, for the given turning function this scenario does not converge to a fixed point. $\square$

Proof 2: Another approach to convergence

Now, we are going to look at the other conjecture we made above: For every configuration and any turn function, moving an unhappy person will not increase the number of unhappy people. Although this is true in most cases, it does not always hold. Here is a counter example.

Starting with the following scenario with 8 players (4 white and 4 blacks):

	·					·	
1	2	3	4	5	6	7	8
○	*	○	*	○	*	○	*

With a local neighbourhood of size 2, we see that players on positions 2 and 7 are unhappy. Picking player on position 2 to move first, this will move in between players 4 and 5 since this is the nearest position that will make him happy. After player 2 moves, we have the following configuration.

	·	·		·		·	
1	3	4	2	5	6	7	8
○	○	*	*	○	*	○	*

We started with a configuration with only 2 unhappy players. After a single move, we now have 4 unhappy players. The number of unhappy players increased, hence, the above statement does not hold for all configurations and any turn functions. □

Proof 3: Termination using a specific turn function

We are going to show this by relaxing the constraint that you need to go to the closest neighbourhood that makes you happy. We will allow players to go to any neighbourhood.

Strong assumptions: We have enough black and white people (at least as many as `neighbourhood size + 1`), and people can freely move to a neighbourhood that makes them happy. Then, there are ways of moving the agents that always converge. `i.e.` the algorithm terminates, reaches a final point.

Consider a general scenario: $x_1 ... x_n$

Go from left to right until we find the first unhappy person. If no unhappy person than we are done. Call this unhappy person `A` and let her be white (same reasoning if black). Move her until happy on position `y`.

Assume now that `A'` of same colour is unhappy. Make `A'` move to the left of `A`. So `A` and `A'` are close to each other. Notice that `A` could not have become unhappy because `A'` is close to her.

Repeat until we have that before `A` there is a line of black people followed by a line of white people, which includes `A` and `A'`.

Going to the right of `A'`, until first unhappy person. Let's make her move to the right of `A'`. We are not making `A'` unhappy by doing this. Nor all the other whites before `A'`.

Repeat until all the white people on the right of `A'` are happy. Hence, all the white people are happy.

Now, for blacks, if noone can improve or everyone is happy, we are done.

Suppose there is a black, `B`, that can improve. We notice that no such improvement can happen by moving inside the white line that we have constructed in the middle. So it must be outside. Repeat the procedure for black people, creating a black chain.

We obtain a scenario of the form:

$$Random\ People + White\ Chain + Random\ People + Black\ Chain$$

So now, for every individual in the `Random People` that can improve his/her position (`i.e.` there are unhappy, but they can be happy), join them with the white chain if they are white, or with the black chain if they are black.

Hence, the algorithm terminates. $\square$

5 Implementation - Multi-Agent Computational Model

We are interested in automatically testing Schelling's linear segregation model. We want to answer questions such as "What is the least segregated final configuration we could reach?" or "Given an initial configuration, will this scenario *converge* for any given turn function?". Furthermore, we are interested in getting the path that led to the least segregated final configuration.

Recall, in Schelling's linear model agents are picked alternating left-most to right-most unhappy player and they will move to the closest position that makes them happy. We are going to ignore the *cost* associated to moving, but we are going to keep the idea of agents moving to the closest place that make them happy since, if the moving cost were to be considered, this would generally mean least expensive move. However, when it comes to the order in which the agents get to move, we have decided to go beyond Schelling's model and look at what happens if we use different turn functions. In the previous section, we had a look at some scenarios where the different turn functions led to completely different results and in this section we are going to see if the same results are obtained by the computational model. We will ensure that all the unhappy players have equal chances to move by introducing the fairness constraint.

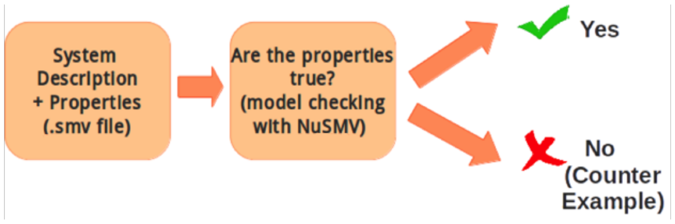

System Description + Properties (.smv file)
Are the properties true? (model checking with NuSMV)
Yes
No (Counter Example)

Figure 9: NuSMV Model Checker

For this project we decided to use *NuSMV* model checker since it allows us to test all the possible scenarios starting from a given initial configuration, using different turn functions. The model checker creates a *Binary Decision Diagram* with all the possible states following the rules of the system. As we can see in *Figure 9*, the first step was to create a `.smv file` with the system description and the properties written in CTL[**?**]. Once we have the model of the system and the specifications, we can run using NuSMV, checking whether the specifications hold.

In the following sections, we are going to first explain how we modeled the linear scenario. We are going to see the SMV definitions of the line, happiness, movement turn and others. After, we are going to see at what specifications we considered and how we defined them. We needed a way to generate the `.smv file` model automatically, depending only on the user inputs like number of agents, size of the local neighbourhood and the colour of the agents at each position. To do all these, we wrote a Python script at which we are going to look briefly below.

5.1 SMV Model

In this section, we are going to look at the *System Description* of the linear model. The system consists of a line with `n` agents (the *whole* world) and a local neighbourhood size `s`. Our model needs to initialise the positions of the line as passed in from the user, `i.e.` the user decides how many players are there and what type (colour) each player is. The model also has to define the rules that determine whether an agent is happy or not and positions he could move to. It defines which are the new potential locations for an agent, that is, which positions are the nearest and make the agent happy. For this model, we defined two modules: a *main module* and a *line module*. Each module is an encapsulated description that can be instantiated within the model. The module *main* forms the root of the model hierarchy and the starting point for building the finite-state model. We have declared local variables within each module and we defined the value of each of these variables in each state using `init` and `next`. Below, we are going to analyse and give a small example for each of these two modules and then look at the fairness constraint we needed.

MODULE line

This module encodes the agents in the line. As we see below, we have a variable line which is an array with `n` positions and each element of the array can be either 0 or 1; 0 for white and 1 for black. So `line[3] = 0` represents that the person at position 3 is white. It also has a Boolean array called `happy` representing the happiness status of each position in the line. So, for example, `happy[3] = TRUE` represents that fact that the player on position 3 is happy. Furthermore, we can see that the module takes as input `old_pos` (the current position of the person to move) and `new_pos` (the new position of the person to move).

```
MODULE line(old_pos,new_pos)
  VAR
     line  : array 1..n of 0..1;
     happy : array 1..n of boolean;
  ASSIGN
-- Initialise the line, i.e. the first configuration
     init(line[1]) := 1;
     init(line[2]) := 0;
     ...
     init(line[n]) := 1;

-- This is how the colours change in the line when the person in
-- old_pos moves to new_pos.
     next(line[1]) :=
      case
        new_pos = 1 : line[old_pos];
        new_pos > old_pos & 1 >= old_pos & 1 < new_pos : line[2];
        TRUE : line[1];
      esac;
    next(line[2]) :=
      case
        new_pos =  2 : line[old_pos];
        new_pos > old_pos &  2 >= old_pos &  2 < new_pos : line[3];
        new_pos < old_pos & 2 > new_pos & 2 <= old_pos : line[1];
        TRUE : line[2];
      esac;
    ...
    next(line[n]) :=
      case
        new_pos = n : line[old_pos];
        new_pos < old_pos & n > new_pos & n <= old_pos : line[n];
        TRUE : line[n];
      esac;

-- Initialise happiness statuses for a local neighbourhood of size
    1,
-- i.e. each person cares about the colour of the person at
-- his right and the person at his left
    init(happy[1]) :=
      case
        line[1] = 0 & line[2]  <= 0 : TRUE;
        line[1] = 1 & line[2]  >= 1 : TRUE;
        TRUE: FALSE;
      esac;
    init(happy[2]) :=
      case
        line[2] = 0 & line[1] + line[3] <= 1 : TRUE;
        line[2] = 1 & line[2] + line[3] >= 1 : TRUE;
        TRUE: FALSE;
      esac;
    ...
    init(happy[n]) :=
      case
        line[n] = 0 & line[n-1] <= 0 : TRUE;
        line[n] = 1 & line[n-1] >= 1 : TRUE;
        TRUE: FALSE;
      esac;


-- This is how the hapiness statuses change in the line when the
    person in
-- old_pos moves to new_pos.
    next(happy[1]) :=
      case
        next(line[1]) = 0 & next(line[2]) <= 0 : TRUE;
        next(line[1]) = 1 & next(line[2]) >= 1 : TRUE;
        TRUE: FALSE;
      esac;
    next(happy[2]) :=
      case
        next(line[2]) = 0 & next(line[1]) + next(line[3]) <= 1 :
```

```
    TRUE;
        next(line[2]) = 1 & next(line[1]) + next(line[3]) >= 1 :
    TRUE;
        TRUE: FALSE;
      esac;
    ...
    next(happy[3]) :=
      case
        next(line[n]) = 0 & next(line[n-1]) <= 0 : TRUE;
        next(line[n]) = 1 & next(line[n-1]) >= 1 : TRUE;
        TRUE: FALSE;
      esac;
```

MODULE main

There are three local variables: old position, new position and persons. The old and new position variable take an integer value between 1 and n, where n is the total number of agents, and the variable persons represents the people in a line. `persons` is an instance of the module `line`. This allows us to build a structural hierarchy. Here is a small example of the main module for just 3 agents, `n = 3`, and local neighbourhood size of 1, `s = 1`, that is, each agent is interested in the type of the two agents that are the closest, one on his right and one on his left. Recall, we say that an agent is happy if at least 50% of his neighbours are like oneself.

```
MODULE main
  VAR
    old_pos: 1..3;
    new_pos: 1..3;
    change : boolean;
    persons: line(old_pos,new_pos);

  ASSIGN
    init(new_pos) :=
      case
        old_pos=1 & persons.happy[1] = TRUE : 1;
        old_pos=1 & persons.happy[1] = FALSE & persons.line[1] = 0
    &  persons.line[2] + persons.line[3] <= 1 : {2};
        old_pos=1 & persons.happy[1] = FALSE & persons.line[1] = 1
    &  persons.line[2] + persons.line[3] >= 1 : {2};
        old_pos=1 & persons.happy[1] = FALSE & persons.line[1] = 0
    & persons.line[3] <= 0 : {3};
        old_pos=1 & persons.happy[1] = FALSE & persons.line[1] = 1
    & persons.line[3] >= 1 : {3};
        old_pos=2 & persons.happy[2] = TRUE : 2;
        old_pos=2 & persons.happy[2] = FALSE & persons.line[2] = 0
    & persons.line[1] <= 0 & persons.line[3] <= 0 : {1,3};
        old_pos=2 & persons.happy[2] = FALSE & persons.line[2] = 1
    & persons.line[1] >= 1 & persons.line[3] >= 1 : {1,3};
        old_pos=2 & persons.happy[2] = FALSE & persons.line[2] = 0
    & persons.line[1] <= 0 : {1};
        old_pos=2 & persons.happy[2] = FALSE & persons.line[2] = 1
    & persons.line[1] >= 1 : {1};
```

```
        old_pos=2 & persons.happy[2] = FALSE & persons.line[2] = 0
    & persons.line[3] <= 0 : {3};
        old_pos=2 & persons.happy[2] = FALSE & persons.line[2] = 1
    &  persons.line[3] >= 1 : {3};
        old_pos=3 & persons.happy[3] = TRUE : 3;
        old_pos=3 & persons.happy[3] = FALSE & persons.line[3] = 0
    & persons.line[1] + persons.line[2] <= 1 : {2};
        old_pos=3 & persons.happy[3] = FALSE & persons.line[3] = 1
    & persons.line[1] + persons.line[2] >= 1 : {2};
        old_pos=3 & persons.happy[3] = FALSE & persons.line[3] = 0
    & persons.line[1] <= 0 : {1};
        old_pos=3 & persons.happy[3] = FALSE & persons.line[3] = 1
    & persons.line[1] >= 1 : {1};
        TRUE : old_pos;
    esac;

    next(new_pos) :=
      case
        next(old_pos)=1 & next(persons.happy[1]) = TRUE : 1;
        next(old_pos)=1 & next(persons.happy[1]) = FALSE & next(
    persons.line[1]) = 0 & next(persons.line[2]) + next(persons.
    line[3]) <= 1 : {2};
        ...
        next(old_pos)=3 & next(persons.happy[3]) = FALSE & next(
    persons.line[3]) = 1 & next(persons.line[1]) >= 1 : {1};
        TRUE : next(old_pos);
    esac;
```

The `old_pos` variable is chosen arbitrary between 1 and n, where n is the total number of agents. If, at a step, the `olp_pos = 3`, then this represents that is the turn of person on position 3 to move. If the agent at position 3 is happy, (`i.e. persons.happy[3] = TRUE`), then the player will remain in the same position, (`i.e.` we set the value of the `new_pos` variable above to 3). Otherwise, if the agent is not happy, we find the nearest position that makes him happy. We do this from nearest to further. Notice, if there are two equidistant positions that would satisfy the agent then, it can choose randomly between the two positions. In the example below, if it is the turn for player 2 to move and player two is 0 (black) and both position 1 and 3 would meet his demands then it can be chosen randomly between the two positions.

```
old_pos=2 & persons.happy[2] = FALSE & persons.line[2] = 0 &
    persons.line[1] <= 0 & persons.line[3] <= 0 : {1,3};
```

Fairness constraints

At this stage, we can see that an incorrect behaviour might occur. This would corresponds to circumstances that are not of interest, like same player getting to move over and over again. Thus, we have to assume *fairness* for all the agents in the model. We were able to do so by simply adding the following constraints.

```
FAIRNESS old_pos = 1
FAIRNESS old_pos = 2
FAIRNESS old_pos = 3
 ...
FAIRNESS old_pos = n
```

`FAIRNESS old_pos = p` indicates that only paths in which `old_pos = p` happens infinitely often will be traversed. This is enough to ensure that each player will get to move infinitely often.

5.2 CTL Specifications

We first had to consider what properties are we interested in checking, whether they hold or not in our model of the system. Firstly, and very important, we want to look at whether or not the system will converge in all scenarios. The system *converges* if it reaches a terminal state. A terminal state is a state in which no unhappy player is able to improve. Secondly, we are interested in how segregated the final configurations are. We want final configurations that are as least segregated as possible. We are now going to look at what other variables we introduced in order to check the above statements and what specifications we have written.

5.2.1 Convergence

To check convergence, we introduced a Boolean variable in the main module called `change`. This Boolean variable is *true* if at least one player has changed his position, *false* otherwise. Here is how it is defined.

```
    change : boolean;
     ...
    init(change) := FALSE;

    next(change) :=
      case
        next(old_pos) != next(new_pos) : TRUE;
        TRUE : FALSE;
      esac;

```

And here is the CTL specification we wrote.

```
SPEC
 AF AG (!change);

```

Translating this specification from CTL to *English*, would be the following: "For all the possible paths, there will be a point in the future where `change` will always be false". In other words, in all the paths, we will reach a point where nobody will move any longer.

5.2.2 Degree of Segregation

In order to check how segregated our model is, we introduced a new variable `segregation_table` in the `MODULE line`. This variable is an array of zeros and ones. The size of the array is `n-1`, where `n` is the total number of agents. Here is how we defined it.

```
    separation_table : array 1..n-1 of 0..1;
    ...
    init(separation_table[1]) :=
      case
        line[1] != line[2] : 1;
        TRUE : 0;
      esac;

    init(separation_table[2]) :=
      case
        line[2] != line[3] : 1;
        TRUE : 0;
      esac;
    ...
    init(separation_table[n-1]) :=
      case
        line[n-1] != line[n] : 1;
        TRUE : 0;
      esac;

    next(separation_table[1]) :=
      case
        next(line[1]) != next(line[2]) : 1;
        TRUE : 0;
      esac;

    next(separation_table[2]) :=
      case
        next(line[2]) != next(line[3]) : 1;
        TRUE : 0;
      esac;
    ...
    next(separation_table[n-1]) :=
      case
      next(line[n-1]) != next(line[n]) : 1;
      TRUE : 0;
    esac;


```

In the `separation_table` array we store ones if there is a colour change, zeros otherwise. In other words, `separation_table[3] = 1` means that the person on position three has a different colour than person on position 4.

Given the following configuration

○	○	*	*	○	*	○	*

The `separation_table` will look like this:

0	1	0	1	1	1	1

This new variable allows us to write specifications to answer questions like *"Is complete segregation possible?"*. We say we have a **complete segregation** when all the whites form a contiguous group and all the black form another contiguous group. Hence we can define complete segregation as the sum of all the elements of the array `separation_table` being equal to 1, or 0 if everybody in the model is of the same colour.

Complete Segregation:

`separation_table[1] + ... + separation_table[n-1] = 1 or 0` (It is equal to 0 in a special case where all agents are of the same type. This case is not of interest to us, but we will consider it in the specification)

And here is how we would write the specification in CTL to check complete segregation:

```
SPEC
  AF AG (
  persons.separation_table[1] + ... + persons.separation_table[n-1] = 1 |
  persons.separation_table[1] + ... + persons.separation_table[n-1] = 0 );

```

This specification checks that for all the paths, there will be complete segregation at some point.

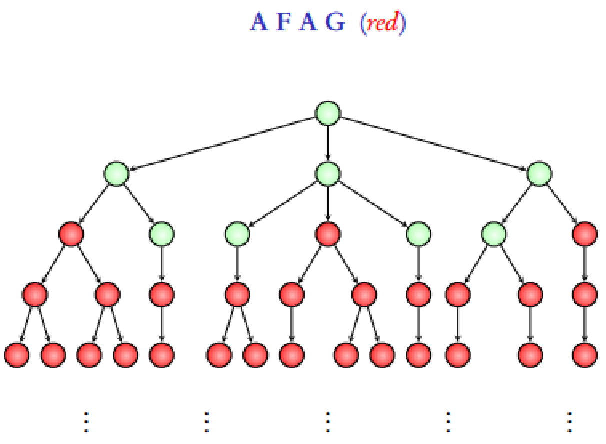


Figure 10: For all paths, red is eventually true forever [?]

We also want to check what is the least segregated, final configuration we can reach. We check the sum of the elements of the `separation_table` array all the way up to n-1, that is, the largest value we can get and it is reachable when whites and blacks are alternating (`i.e.` one white, one black, one white, one black,...). In other words, all the elements of the `separation_table` array will equal to 1 if we have the agents in the line alternating black, white.

```
SPEC
  EF AG (
  persons.separation_table[1] + ... + persons.separation_table[n−1]
     = 1 );

SPEC
  EF AG (
  persons.separation_table[1] + ... + persons.separation_table[n−1]
     = 2 );

  ...

SPEC
  EF AG (
  persons.separation_table[1] + ... + persons.separation_table[n−1]
     = n−1 );

```

`EF AG p` means that *there exists a future point from which, in all the future paths, p will hold.*

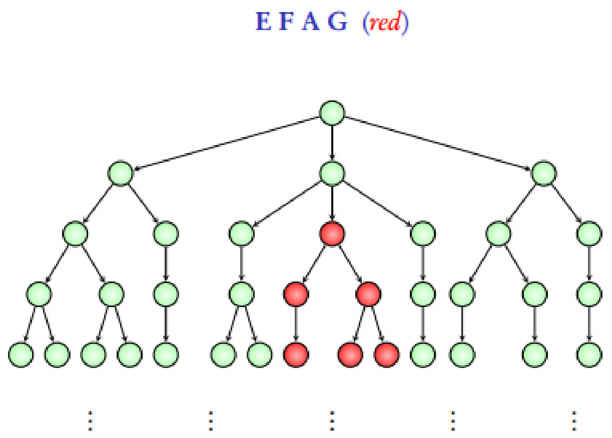


Figure 11: There exists a path such that red is eventually true forever [?]

```
SPEC
  EF AG (
  persons.separation_table[1] + ... + persons.separation_table[n−1]
     = 2 );

```

Hence, the above specification checks whether there is a final configuration with three groups, that is, something like this:

○	○	○	...	*	*	*	...	○	○	○

Once we got the highest number, the least segregated final configuration, we could use NuSMV to generate the path that led to that configuration. In other words, we can ask NuSMV to give us the best scenario where the final configuration is least segregated.

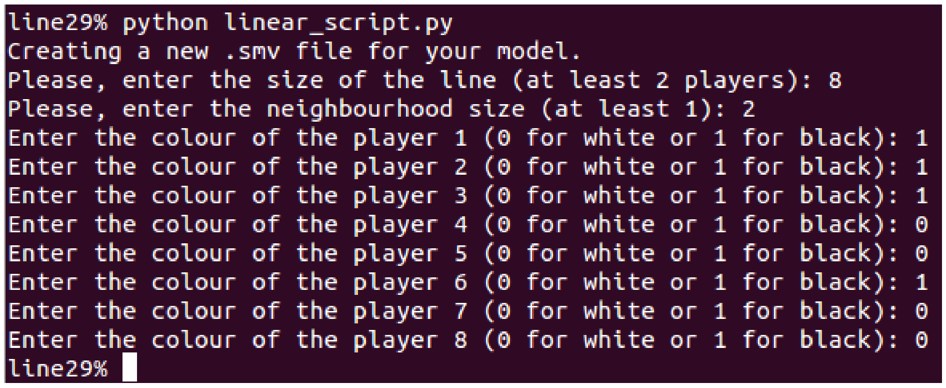
A complete `.smv file` example can be found at Appendix A.

5.3 Python Script

In the sections above, we had a look at how we designed the model of the system and what does the `.smv file` includes in terms of System Description and Properties. However, SMV has some limitations such as the *main* module which forms the root of the model hierarchy does not take any parameters. Furthermore, SMV does not support control flow statements, such as `for-` or `while-` loops. Therefore, we would needed to write a new file, create a new model, for each configuration we want to represent. It was natural to use another programming language that allows control flow statements and can take user inputs and generate the `.smv file`. We chose *Python*[?].

We use the following user inputs:

- The number of agents we have in the line
- The size of the local neighbourhood
- The initial configuration, `i.e.` the colour of the person at each position



```
line29% python linear_script.py
Creating a new .smv file for your model.
Please, enter the size of the line (at least 2 players): 8
Please, enter the neighbourhood size (at least 1): 2
Enter the colour of the player 1 (0 for white or 1 for black): 1
Enter the colour of the player 2 (0 for white or 1 for black): 1
Enter the colour of the player 3 (0 for white or 1 for black): 1
Enter the colour of the player 4 (0 for white or 1 for black): 0
Enter the colour of the player 5 (0 for white or 1 for black): 0
Enter the colour of the player 6 (0 for white or 1 for black): 1
Enter the colour of the player 7 (0 for white or 1 for black): 0
Enter the colour of the player 8 (0 for white or 1 for black): 0
line29%
```

Figure 12: Example of user input when running the Python script

For each local variable in the model, we wrote Python functions which, based on the input values, are able to define the values of these variables in each state.

6 Simulation and Evaluation

In this section, we are looking at how we run the application and what specification we have tested with `NuSMV`. We will analyse the results we get and give an overall evaluation of the computational model.

6.1 Computational Results

It is time to look at some computational results. We want to check that the results we got in the *Formal Analysis* section are the same as the results we get from the model checker. We are also going to look at some further examples and explain some of the counter examples that `NuSMV` generated.

We are mainly interested in testing the *stability* of our configurations and the *degree of segregation*. We say that a configuration is stable if no happy people can improve and the degree of segregation refers to how many contiguous groups are in our line configuration. We want to find out what is the least segregated outcome we can reach.

6.1.1 Stability

For simplicity, we are going to consider the neighbourhood size 2 for all the examples in this section. That is, each agent is interested in the what colour are the two agents at his right and the two agents at his left. Also, our model considers an agent to be *happy* if at least 50% of his neighbours are like oneself.

Example 1

The first starting configuration we are going to use as input is the following:

1	2	3	4	5	6	7	8	9	10	11	12
*	*	○	○	○	○	○	○	*	*	*	*

We check the convergence of this configuration using the following specification.

```
SPEC
  AF AG (!change);
```

The convergence property is violated by this configuration and a counter-example is generated.

```
-- specification AF (AG !change)  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-- Loop starts here
-> State: 1.1 <-
  old_pos = 8
  new_pos = 8
  change = FALSE
  persons.line[1] = 1
  persons.line[2] = 1
  persons.line[3] = 0
  persons.line[4] = 0
  persons.line[5] = 0
  persons.line[6] = 0
  persons.line[7] = 0
  persons.line[8] = 0
  persons.line[9] = 1
  persons.line[10] = 1
  persons.line[11] = 1
  persons.line[12] = 1
  persons.happy[1] = TRUE
  persons.happy[2] = FALSE
  persons.happy[3] = TRUE
  persons.happy[4] = TRUE
  persons.happy[5] = TRUE
  persons.happy[6] = TRUE
  persons.happy[7] = TRUE
  persons.happy[8] = TRUE
  persons.happy[9] = TRUE
  persons.happy[10] = TRUE
  persons.happy[11] = TRUE
  persons.happy[12] = TRUE
  ...
-> State: 1.2 <-
  old_pos = 12
  new_pos = 12
-> State: 1.3 <-
  old_pos = 11
  new_pos = 11
-> State: 1.4 <-
  old_pos = 10
  new_pos = 10
-> State: 1.5 <-
  old_pos = 9
  new_pos = 9
-- Loop starts here
-> State: 1.6 <-
  old_pos = 8
  new_pos = 8
-- Loop starts here
-> State: 1.6 <-
  old_pos = 8
  new_pos = 8
```

```
-> State: 1.7 <-
  old_pos = 7
  new_pos = 7
-> State: 1.8 <-
  old_pos = 6
  new_pos = 6
-> State: 1.9 <-
  old_pos = 5
  new_pos = 5
-> State: 1.10 <-
  old_pos = 4
  new_pos = 4
-> State: 1.11 <-
  old_pos = 3
  new_pos = 3
-> State: 1.12 <-
  old_pos = 2
  new_pos = 1
  change = TRUE
-> State: 1.13 <-
  old_pos = 1
  change = FALSE
-> State: 1.14 <-
  old_pos = 8
  new_pos = 8

```

We can see the initial configuration and we notice that only the person on position 2 is unhappy. Hence, our counter example shows us in `state: 1.12` that for the old position 2 we have the new position 1 (`i.e.` the unhappy person on position 2 moves to the nearest position that makes him happy and that is position 1). The old position 1 becomes the new position 2 and it becomes unhappy. This is a loop. Thus, the configuration does not converge indeed.

Example 2

The second configuration we are considering is the following.

			·				·			
1	2	3	4	5	6	7	8	9	10	11
○	○	○	*	○	○	○	*	○	○	○

Checking the same specification as in the first example, we get that the property holds for this configuration. That is, in all the paths, we reach a stable configuration where no unhappy people can improve their position.

```
> specification AF (AG !change)  is true

```

Notice that the two unhappy agents do not move because there is no position that would make them happy. That would be a possibility to improve their status is they were to move together (at the same time). However, this is not something that our model allows. Same as in Schelling's model, we move one agent at the time.

Example 3

Another configuration we consider is the following.

	·	·	·	·	·	·	
1	2	3	4	5	6	7	8
*	*	○	○	*	*	○	○

For this example, we checked the following two specifications:

- "For all possible turn functions, we will reach a stable configuration."

```
SPEC
     AF AG (!change);

```

- "There exists a turn function such that we will reach a final, stable configuration."

```
SPEC
     EF AG (!change);

```

The later is true.

```
> specification EF (AG !change)  is true

```

An example of path when this property is true is the following.

First, move player on position 5.

			·		·		
1	2	5	3	4	6	7	8
*	*	*	○	○	*	○	○

Next, move player 6.

1	2	5	6	3	4	7	8
*	*	*	*	○	○	○	○

We get complete segregation, everybody is happy so we stop.

However, the former property does not hold. It is not the case that for any turn function, we will reach a stable segregation. A counter-example generated by `NuSMV` can be found in Appendix B.

6.1.2 Degree of Segregation

In this part, we are interested in checking whether complete segregation always occurs, regardless of the turn function we use or finding out what is the least segregated final configuration that we can reach. Same as in previous examples, we keep the neighbourhood size 2 and the agents being content with 50% of their neighbours being like oneself.

Example 1

The initial configuration we consider here is the following.

						·		
1	2	3	4	5	6	7	8	9
*	*	*	○	○	○	*	○	○

First, we check if this initial scenario will always lead to complete segregation in the final configuration. Below is the CTL specification to test this. Recall, we are using the `segregation_table` array to test this.

```
SPEC
  AF AG (
     persons.separation_table[1] + persons.separation_table[2] +
     persons.separation_table[3] + persons.separation_table[4] +
     persons.separation_table[5] + persons.separation_table[6] +
     persons.separation_table[7] + persons.separation_table[8] =1 |
     persons.separation_table[1] + persons.separation_table[2] +
     persons.separation_table[3] + persons.separation_table[4] +
     persons.separation_table[5] + persons.separation_table[6] +
     persons.separation_table[7] + persons.separation_table[8]  = 0
     );
```

And here is the NuSMV result.

```
> specification AF (AG (((((((persons.separation_table[1] + persons
    .separation_table[2]) + persons.separation_table[3]) + persons.
    separation_table[4]) + persons.separation_table[5]) + persons.
    separation_table[6]) + persons.separation_table[7]) + persons.
    separation_table[8] = 1 | ((((((persons.separation_table[1] +
    persons.separation_table[2]) + persons.separation_table[3]) +
    persons.separation_table[4]) + persons.separation_table[5]) +
    persons.separation_table[6]) + persons.separation_table[7]) +
    persons.separation_table[8] = 0))   is true
```

Hence, we will always get to complete segregation in this scenario.

Example 2

Now, lets consider another initial configuration.

						·				·	
1	2	3	4	5	6	7	8	9	10	11	12
*	*	*	○	○	○	*	○	*	○	*	○

First, we check again if in this scenario we always get to a completely segregated final configuration.

```
> specification AF (AG (((((((((((persons.separation_table[1] +
    persons.separation_table[2]) + persons.separation_table[3]) +
    persons.separation_table[4]) + persons.separation_table[5]) +
    persons.separation_table[6]) + persons.separation_table[7]) +
    persons.separation_table[8]) + persons.separation_table[9]) +
    persons.separation_table[10]) + persons.separation_table[11] =
    1 | (((((((((persons.separation_table[1] + persons.
    separation_table[2]) + persons.separation_table[3]) + persons.
    separation_table[4]) + persons.separation_table[5]) + persons.
    separation_table[6]) + persons.separation_table[7]) + persons.
    separation_table[8]) + persons.separation_table[9]) + persons.
    separation_table[10]) + persons.separation_table[11] = 0))   is
    false
```

We see that this is false. That means that we should be able to reach final configurations that are not completely segregated. We are interested in finding out what is the least segregated final configuration we can reach. To do so we have to check the following specification.

```
EF AG ( persons.separation_table[1] + persons.separation_table[2] +
     persons.separation_table[3] + persons.separation_table[4] +
    persons.separation_table[5] + persons.separation_table[6] +
    persons.separation_table[7] + persons.separation_table[8] +
    persons.separation_table[9] + persons.separation_table[10] +
    persons.separation_table[11] = i );
```

Where `i` takes values from 1 to 11. Checking these CTL specifications in NuSMV, we get the following:

```
> specification EF (AG ((((((((((persons.separation_table[1] +
    persons.separation_table[2]) + persons.separation_table[3]) +
    persons.separation_table[4]) + persons.separation_table[5]) +
    persons.separation_table[6]) + persons.separation_table[7]) +
    persons.separation_table[8]) + persons.separation_table[9]) +
    persons.separation_table[10]) + persons.separation_table[11] =
    1)  is false

> specification EF (AG ((((((((((persons.separation_table[1] +
    persons.separation_table[2]) + persons.separation_table[3]) +
    persons.separation_table[4]) + persons.separation_table[5]) +
    persons.separation_table[6]) + persons.separation_table[7]) +
    persons.separation_table[8]) + persons.separation_table[9]) +
    persons.separation_table[10]) + persons.separation_table[11] =
    2)  is true

> specification EF (AG ((((((((((persons.separation_table[1] +
    persons.separation_table[2]) + persons.separation_table[3]) +
    persons.separation_table[4]) + persons.separation_table[5]) +
    persons.separation_table[6]) + persons.separation_table[7]) +
    persons.separation_table[8]) + persons.separation_table[9]) +
    persons.separation_table[10]) + persons.separation_table[11] =
    3)  is true

> specification EF (AG ((((((((((persons.separation_table[1] +
    persons.separation_table[2]) + persons.separation_table[3]) +
    persons.separation_table[4]) + persons.separation_table[5]) +
    persons.separation_table[6]) + persons.separation_table[7]) +
    persons.separation_table[8]) + persons.separation_table[9]) +
    persons.separation_table[10]) + persons.separation_table[11] =
    4)  is false
 ...
```

We get two out of 11 specifications as being true, for i=2 and i=3. This means that we can reach final configurations with 3 and 4 groups, respectively. In other words, we get two possible levels of segregation.

Here are possible examples of turn functions to reach these final configurations (with 3 and 4 groups). Remember, in our model, players move if they are unhappy to the closest position that makes them happy. However, we allow players to move in any order they want.

For i=2. Initial configuration:

1	2	3	4	5	6	$\dot{7}$	8	9	10	$\dot{11}$	12
*	*	*	○	○	○	*	○	*	○	*	○

Move player 8 to position 7.

1	2	3	4	5	6	8	7	9	$\dot{10}$	$\dot{11}$	12
*	*	*	○	○	○	○	*	*	○	*	○

Move player 10 to position 8.

1	2	3	4	5	6	8	10	7	9	11	$\dot{12}$
*	*	*	○	○	○	○	○	*	*	*	○

Finally, move player 12 next to player 8.

1	2	3	4	5	6	8	12	10	7	9	11
*	*	*	○	○	○	○	○	○	*	*	*

We reached a final, stable configuration where everyone in happy. We have 3 groups.

Next, we look how we get i=3. Initial configuration:

1	2	3	4	5	6	$\dot{7}$	8	9	10	$\dot{11}$	12
*	*	*	○	○	○	*	○	*	○	*	○

Move player from position 11 to position 9.

1	2	3	4	5	6	$\dot{7}$	8	11	$\dot{9}$	$\dot{10}$	12
*	*	*	○	○	○	*	○	*	*	○	○

Move player from position 8 to position 7

1	2	3	4	5	6	8	7	11	9	$\dot{10}$	12
*	*	*	○	○	○	○	*	*	*	○	○

The algorithm terminates here. We notice that player 10 is unhappy. However, from the *degree of segregation* point of view the configuration will have 4 groups forever from this point. This final configuration is not stable since players 10 and 12 can swap places forever(`i.e.` the closest position that makes player 10 happy is position 12).

What is interesting to notice in our last example is the fact that we could have all the players happy, but almost completely segregated (all ◦'s are clustered together), or we could sacrifice the happiness of one of the players, but get a more integrated overall system.

6.2 Evaluation

In this section, we want to have a brief overview of the computational model, focusing on the accuracy of the results and efficiency in terms of time and space constraints. We will also list some pros and cons in using a model checker over an object oriented design.

The difficult part in using a model checker is to make sure that the description of the system is right. In other words, we want to make sure that the definitions, such as happiness, are correct and that the rules for traversing from one state to another are well defined. However, once the model of the system is well defined, `NuSMV` (the model checker) can be used to perform a variety of valuable functions, enabling us to evaluate whether any property claimed in a `SPEC` statement is true in the system. Basically, the model checker analysis all the possible paths to determine whether a specification is true or false. What makes a model checker more valuable than other automatic tests that are out there is the fact that for a property that does not hold (`i.e.` it is false), the model checker generates a counter example.

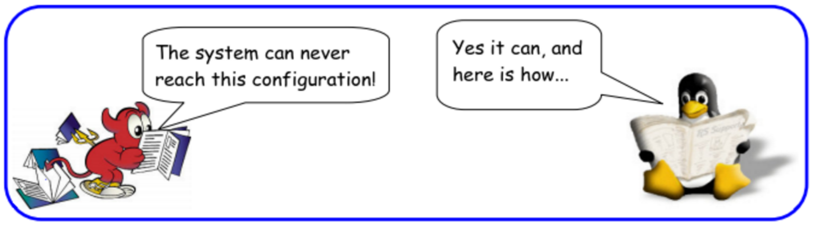

The system can never reach this configuration!
Yes it can, and here is how...

Figure 13: Inside Model Checker [?]

The model checker allowed us to relax Schelling's turn function, being able to test the systems when agents move in all possible orders. And adding the *fairness* constraints ensured that each agent is picked infinitely often. This is something that cannot be achieved in an object oriented design. Furthermore, we were able to check all the results we got in the *Formal Analysis* Chapter.

The series of simulation results we carried out confirmed our formal results in terms of convergence and total segregation of the systems for different turn functions.

On the negative side, the model (`.smv file`) grows exponentially with each agent added to the system and so does the *Binary Decision Diagram (BDD)* generated by the model checker. This is also known as *The State Explosion Problem* [?] and it is due, in our model, to a large number of orders in which we allow the unhappy agents to move. To give an example, the `.smv file` that describes the model for 50 agents has over 14 thousand line and when it comes to BDDs, we are talking about millions of possible states. Having so many states for the model checker to cover, means that the execution time for checking a specification can take tens of minutes. However, there are possible solutions to this problem. One of them could be using a *Bounded Model Checking.* This technique is basically checking just a fraction of the entire model. For example, if we want to check that there exists a path to a final configuration that is stable, it is sufficient to find a path in the bounded model and we would be able to say that the property holds in the system as a whole.

To summarise, here is a brief list of pros and cons for using a model checker for our multi-agent computational model.

Pros:

- Can evaluate whether a variety of proprieties are true of the system model;
- It provides counter-examples for properties that do not hold in our system;
- It allowed us to use a relaxed version of Schelling's turn function. Hence, we were able to test the impact of turn function in terms of convergence of the system and the level of segregation.

Cons:

- Challenging to write the system description in `SMV`;
- The model grows exponentially with each agent added (State Explosion Problem). Thus, we could run into space and time efficiency problems. However, there are techniques that we could use to avoid these problems, such as *Bounded Model Checking.*

7 Conclusion and Open Questions

For the most part, we consider this project a success. We started by giving a detailed description of Schelling's model of segregation, focusing on the spatial proximity model. We gave a formal definition of convergence and we looked into some of Schelling's intuitions regarding stable final configurations. Our investigation showed that Schelling's turn function is not ideal and we were able to give some examples where the convergence of some initial configurations failed when using Schelling's turn function. In other words, we presented examples of configurations that do not reach a stable point where no unhappy player is able to improve. However, we followed up Schelling's intuition of relaxing the turn function in order to ensure termination. We managed to rigorously prove that a specific turn function (that was constructed by relaxing Schelling's turn function) will always guarantee a stable final configuration.

In addition, we successfully implemented a multi-agent computational model using `SMV` model checker. The model was a success because it allowed us to test a variety of linear configurations of different lengths and various neighbourhood sizes. Most importantly, the model was able to generate a Binary Decision Diagram with all the achievable final configurations, considering all the possible orders in which the discontent agents could move. Having all these achievable final configurations allowed us to check how much of an impact the turn function had over the system. We were able to answer questions like "For a given initial configuration and any turn function (unhappy agents are able to move in any order), will the system always converge to a fixed point?" or "Regardless the order in which unhappy agents move, will we always end up with a completed segregated final configuration?". Furthermore, we were able to look at what would be the least segregated final configuration that we could reach. These are all important results since they take us a step closer to understanding if and how we could avoid complete segregation in a system. We are interested in finding the configuration where all the agents are happy and the level of segregation is minimal.

7.1 Future work

In this section, we are going to make some suggestions for possible improvements that can be made in relation to this project:

- **Visualisation Tool**

  A tool that allows the user to visualise the *Binary Decision Diagram*(BDD) generated by the `NuSMV` would be a great addition to this project. This way it would be easy to see just about how many final configurations are achievable, how many of those configurations are stable or completely segregated. However, we could see a potential space problem in generated an image of the whole BDD, especially when we talk of thousands or even millions of states. A more feasible tool (something that we have considered creating if we had had more time) would be one that generates a visualisation of the counter-examples that are generated by the model checker. In Appendix B, we can see an example of a counter-example generated by `NuSMV`. This is not very intuitive and requires a bit of `NuSMV` knowledge to read. Furthermore, a visualisation of the configuration at each step would help the user to acknowledge quicker the direction the counter example is pointing to.

- **More Sophisticated Agents**

  At the moment, our model is a *two-type agent model*. In other words, in our system, we have only two types of agents: whites or blacks. The colour is the only characteristics that distinguish them, in any other aspects, our agents are the same. A possible extension of the project would be considering more sophisticated agents, taking into account, for example, their economic status, religion, sex, occupation etc. It would be interesting to see in the situation where we have more characteristics to distinguish between the agents if we would end up with more integrated society or, on the contrary, these new characteristics would lead to an even more segregated system.

- **2D Spatial Proximity Problem**

  The system in our project is currently represented by a line where agents (or players) live. The next step would be to model a two-dimension system with black and white agents and possible blank spaces, like in Schelling's model. We would be interested to see how much of an impact the turn function would have in such a configuration.

- **Solution for State Explosion Problem**

  We saw that a model checker has lots of advantages over an object oriented language when designing a multi-agent computational model. The model checker allowed us to be very flexible in terms of the turn functions we tested. The *randomness* of the order in which the unhappy agents could move, allowed us to test more scenarios than we would have if, for example, we would have used strictly Schelling's turn function. However, together with the flexibility in the turn function, comes the large-size system problem. For every new agent, the size of the system grows exponentially and so does the *Binary Decision Diagram* generated by the `NuSMV`. As we mentioned in the previous chapter, this is known as the *State Explosion Problem.* Scientists are working on ways to solve this problem, and one of them could be *Bounded Model Checking.* As an extension for this project, we could implement some of the Bounded Model Checking techniques to check whether certain specifications are met, without having to compile the whole Binary Decision Diagram.

Appendices

A .smv File Example

An example of the `.smv file` for the following configuration with neighbourhood size 1.

1	2	3
*	○	*

```
-- The module below encodes the line. It has an array called line where each
-- element can be either 0 or 1; 0 for white, and 1 for black. So line[3]=0
-- represents that there is a white person at position 3. It also has a
-- boolean array called happy representing whether the happiness status of each
-- position in the line. So happy[2]=TRUE represents that the person at
-- position 2 is happy.

-- The module takes as input old_pos (the current position of the person to
-- move) and new_pos (the new position of the person to move)

-- Separation table is an array of size (n-1) of integers 1 and 0
-- separation_table[i] is 1 if line[i] != line[i+1], 0 otherwise

MODULE line(old_pos,new_pos)
  VAR
    line  : array 1..3 of 0..1;
    happy : array 1..3 of boolean;
    separation_table : array 1..2 of 0..1;

  ASSIGN

-- Initialise the line with zeros and ones that are passed as an input
-- i.e. Construct the initial line configuration

    init(line[1]) := 1;
    init(line[2]) := 0;
    init(line[3]) := 1;

-- This is how the colours change in the line when the person in
-- old_pos moves to new_pos.

    next(line[1]) :=
      case
        new_pos = 1 : line[old_pos];
        new_pos > old_pos & 1 >= old_pos & 1 < new_pos : line[2];
        TRUE : line[1];
      esac;
    next(line[2]) :=
      case
        new_pos =  2 : line[old_pos];
        new_pos > old_pos &  2 >= old_pos &  2 < new_pos : line[3];
        new_pos < old_pos & 2 > new_pos & 2 <= old_pos : line[1];
        TRUE : line[2];
      esac;
    next(line[3]) :=
      case
        new_pos = 3 : line[old_pos];
        new_pos < old_pos & 3 > new_pos & 3 <= old_pos : line[2];
        TRUE : line[3];
      esac;

-- We initialise the happiness status.

    init(happy[1]) :=
      case
        line[1] = 0 & line[1] + line[2] - line[1]  <= 0 : TRUE;
        line[1] = 1 & line[1] + line[2] - line[1]  >= 1 : TRUE;
        TRUE: FALSE;
      esac;
    init(happy[2]) :=
      case
        line[2] = 0 & line[1] + line[2] + line[3] - line[2]  <= 1 :
      TRUE;
        line[2] = 1 & line[1] + line[2] + line[3] - line[2]  >= 1 :
      TRUE;
        TRUE: FALSE;
      esac;
    init(happy[3]) :=
      case
        line[3] = 0 & line[2] + line[3] - line[3]  <= 0 : TRUE;
        line[3] = 1 & line[2] + line[3] - line[3]  >= 1 : TRUE;
        TRUE: FALSE;
      esac;


-- This is how the hapiness statuses change in the line when the
    person in
-- old_pos moves to new_pos.

    next(happy[1]) :=
      case
        next(line[1]) = 0 & next(line[1]) +next(line[2]) - next(
    line[1])  <= 0 : TRUE;
        next(line[1]) = 1 & next(line[1]) +next(line[2]) - next(
    line[1])  >= 1 : TRUE;
        TRUE: FALSE;
      esac;
    next(happy[2]) :=
      case
        next(line[2]) = 0 & next(line[1]) +next(line[2]) + next(
    line[3]) - next(line[2])  <= 1 : TRUE;
        next(line[2]) = 1 & next(line[1]) +next(line[2]) + next(
    line[3]) - next(line[2])  >= 1 : TRUE;
        TRUE: FALSE;
      esac;
    next(happy[3]) :=
      case
```

```
        next(line[3]) = 0 & next(line[2]) +next(line[3]) - next(
    line[3])  <= 0 : TRUE;
        next(line[3]) = 1 & next(line[2]) +next(line[3]) - next(
    line[3])  >= 1 : TRUE;
        TRUE: FALSE;
      esac;


-- Create the separation_table array
    init(separation_table[1]) :=
      case
        line[1] != line[2] : 1;
        TRUE : 0;
      esac;

    init(separation_table[2]) :=
      case
        line[2] != line[3] : 1;
        TRUE : 0;
      esac;

    next(separation_table[1]) :=
      case
        next(line[1]) != next(line[2]) : 1;
        TRUE : 0;
      esac;

    next(separation_table[2]) :=
      case
        next(line[2]) != next(line[3]) : 1;
        TRUE : 0;
      esac;



-- The main module has an old_pos variable. The value of this
    variable is
-- always arbitrary from 1 to n. If, at a step, old_pos = 3, then
    we represent
-- that is the turn of the person in position 3 to move. If this
    person is
-- already happy (i.e., in the module above happy[3] = TRUE), then
    it remains
-- in the same position (i.e., we set the new_pos variable below to
    3).
-- Otherwise, if the person at position 3 is not happy, then we
    find the
-- nearest position where it could be happy. We do this in cases,
    from nearest
-- to furthest.

-- The boolean variable -change- is true if at least one agent
    changed his/her position
-- It is false if no player has moved

MODULE main
  VAR
```

```
    old_pos: 1..3;
    new_pos: 1..3;
    change : boolean;
    persons: line(old_pos,new_pos);

  ASSIGN
    init(new_pos) :=
      case
        old_pos=1 & persons.happy[1] = TRUE : 1;
        old_pos=1 & persons.happy[1] = FALSE & persons.line[1] = 0 & persons.line[1] + persons.line[2] + persons.line[3] - persons.line[1] <= 1 : {2};
        old_pos=1 & persons.happy[1] = FALSE & persons.line[1] = 1 & persons.line[1] + persons.line[2] + persons.line[3] - persons.line[1] >= 1 : {2};
        old_pos=1 & persons.happy[1] = FALSE & persons.line[1] = 0 & persons.line[3] <= 0 : {3};
        old_pos=1 & persons.happy[1] = FALSE & persons.line[1] = 1 & persons.line[3] >= 1 : {3};
        old_pos=2 & persons.happy[2] = TRUE : 2;
        old_pos=2 & persons.happy[2] = FALSE & persons.line[2] = 0 & persons.line[1] + persons.line[2] - persons.line[2] <= 0 & persons.line[2] + persons.line[3] - persons.line[2] <= 0 : {1,3};
        old_pos=2 & persons.happy[2] = FALSE & persons.line[2] = 1 & persons.line[1] + persons.line[2] - persons.line[2] >= 1 & persons.line[2] + persons.line[3] - persons.line[2]  >= 1 : {1,3};
        old_pos=2 & persons.happy[2] = FALSE & persons.line[2] = 0 & persons.line[1] + persons.line[2] - persons.line[2] <= 0 : {1};
        old_pos=2 & persons.happy[2] = FALSE & persons.line[2] = 1 & persons.line[1] + persons.line[2] - persons.line[2] >= 1 : {1};
        old_pos=2 & persons.happy[2] = FALSE & persons.line[2] = 0 & persons.line[2] + persons.line[3] - persons.line[2] <= 0 : {3};
        old_pos=2 & persons.happy[2] = FALSE & persons.line[2] = 1 & persons.line[2] + persons.line[3] - persons.line[2] >= 1 : {3};
        old_pos=3 & persons.happy[3] = TRUE : 3;
        old_pos=3 & persons.happy[3] = FALSE & persons.line[3] = 0 & persons.line[1] + persons.line[2] + persons.line[3] - persons.line[3] <= 1 : {2};
        old_pos=3 & persons.happy[3] = FALSE & persons.line[3] = 1 & persons.line[1] + persons.line[2] + persons.line[3] - persons.line[3] >= 1 : {2};
        old_pos=3 & persons.happy[3] = FALSE & persons.line[3] = 0 & persons.line[1] <= 0 : {1};
        old_pos=3 & persons.happy[3] = FALSE & persons.line[3] = 1 & persons.line[1] >= 1 : {1};
        TRUE : old_pos;
    esac;

    next(new_pos) :=
      case
        next(old_pos)=1 & next(persons.happy[1]) = TRUE : 1;
        next(old_pos)=1 & next(persons.happy[1]) = FALSE & next(persons.line[1]) = 0 & next(persons.line[1]) + next(persons.line[2]) + next(persons.line[3]) - next(persons.line[1]) <= 1 : {2};
        next(old_pos)=1 & next(persons.happy[1]) = FALSE & next(persons.line[1]) = 1 & next(persons.line[1]) + next(persons.line[2]) + next(persons.line[3]) - next(persons.line[1]) >= 1 : {2};
        next(old_pos)=1 & next(persons.happy[1]) = FALSE & next(persons.line[1]) = 0 & next(persons.line[3]) <= 0 : {3};
        next(old_pos)=1 & next(persons.happy[1]) = FALSE & next(persons.line[1]) = 1 & next(persons.line[3]) >= 1 : {3};
        next(old_pos)=2 & next(persons.happy[2]) = TRUE : 2;
        next(old_pos)=2 & next(persons.happy[2]) = FALSE & next(persons.line[2]) = 0 & next(persons.line[1]) + next(persons.line[2]) - next(persons.line[2]) <= 0 & next(persons.line[2]) + next(persons.line[3]) - next(persons.line[2]) <= 0 : {1,3};
        next(old_pos)=2 & next(persons.happy[2]) = FALSE & next(persons.line[2]) = 1 & next(persons.line[1]) + next(persons.line[2]) - next(persons.line[2]) >= 1 & next(persons.line[2]) + next(persons.line[3]) - next(persons.line[2])  >= 1 : {1,3};
        next(old_pos)=2 & next(persons.happy[2]) = FALSE & next(persons.line[2]) = 0 & next(persons.line[1]) + next(persons.line[2]) - next(persons.line[2]) <= 0 : {1};
        next(old_pos)=2 & next(persons.happy[2]) = FALSE & next(persons.line[2]) = 1 & next(persons.line[1]) + next(persons.line[2]) - next(persons.line[2]) >= 1 : {1};
        next(old_pos)=2 & next(persons.happy[2]) = FALSE & next(persons.line[2]) = 0 & next(persons.line[2]) + next(persons.line[3]) - next(persons.line[2]) <= 0 : {3};
        next(old_pos)=2 & next(persons.happy[2]) = FALSE & next(persons.line[2]) = 1 & next(persons.line[2]) + next(persons.line[3]) - next(persons.line[2]) >= 1 : {3};
        next(old_pos)=3 & next(persons.happy[3]) = TRUE : 3;
        next(old_pos)=3 & next(persons.happy[3]) = FALSE & next(persons.line[3]) = 0 & next(persons.line[1]) + next(persons.line[2]) + next(persons.line[3]) - next(persons.line[3]) <= 1 : {2};
        next(old_pos)=3 & next(persons.happy[3]) = FALSE & next(persons.line[3]) = 1 & next(persons.line[1]) + next(persons.line[2]) + next(persons.line[3]) - next(persons.line[3]) >= 1 : {2};
        next(old_pos)=3 & next(persons.happy[3]) = FALSE & next(persons.line[3]) = 0 & next(persons.line[1]) <= 0 : {1};
        next(old_pos)=3 & next(persons.happy[3]) = FALSE & next(persons.line[3]) = 1 & next(persons.line[1]) >= 1 : {1};
        TRUE : next(old_pos);
    esac;

-- Initilise change variable
    init(change) := FALSE;

    next(change) :=
      case
        next(old_pos) != next(new_pos) : TRUE;
        TRUE : FALSE;
      esac;
```

```

  FAIRNESS old_pos = 1
  FAIRNESS old_pos = 2
  FAIRNESS old_pos = 3


SPEC
  AF AG (!change);

-- Is complete segregation going to occur in all scenarios?
SPEC
  AF AG ( persons.separation_table[1] + persons.separation_table[2]
     =1 | persons.separation_table[1] + persons.separation_table[2]
      = 0 );

-- Testing for different degrees of segregation
SPEC
  EF AG ( persons.separation_table[1] + persons.separation_table[2]
     = 1 );

SPEC
  EF AG ( persons.separation_table[1] + persons.separation_table[2]
     = 2 );
```

B Convergence - NuSMV Counter Example

Initial Configuration:

	·	·	·	·	·	·	
1	2	3	4	5	6	7	8
*	*	○	○	*	*	○	○

Specification Tested:

```
SPEC
    AF AG (!change);

```

Counter Example:

```
-- specification AF (AG !change)  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  old_pos = 8
  new_pos = 8
  change = FALSE
  persons.line[1] = 1
  persons.line[2] = 1
  persons.line[3] = 0
  persons.line[4] = 0
  persons.line[5] = 1
  persons.line[6] = 1
  persons.line[7] = 0
  persons.line[8] = 0
  persons.happy[1] = TRUE
  persons.happy[2] = FALSE
  persons.happy[3] = FALSE
  persons.happy[4] = FALSE
  persons.happy[5] = FALSE
  persons.happy[6] = FALSE
  persons.happy[7] = FALSE
  persons.happy[8] = TRUE
  persons.separation_table[1] = 0
  persons.separation_table[2] = 1
  persons.separation_table[3] = 0
  persons.separation_table[4] = 1
  persons.separation_table[5] = 0
  persons.separation_table[6] = 1
  persons.separation_table[7] = 0
  persons.separation_table[8] = 0
-> State: 1.2 <-
  old_pos = 7
  new_pos = 6
  change = TRUE
```

```
-> State: 1.3 <-
  old_pos = 6
  change = FALSE
  persons.line[6] = 0
  persons.line[7] = 1
  persons.happy[4] = TRUE
  persons.happy[6] = TRUE
  persons.separation_table[5] = 1
  persons.separation_table[7] = 1
-> State: 1.4 <-
  old_pos = 2
  new_pos = 3
  change = TRUE
-> State: 1.5 <-
  old_pos = 5
  new_pos = 5
  change = FALSE
  persons.line[2] = 0
  persons.line[3] = 1
  persons.happy[3] = TRUE
  persons.happy[5] = TRUE
  persons.separation_table[1] = 1
  persons.separation_table[3] = 1
-> State: 1.6 <-
  old_pos = 4
  new_pos = 4
-> State: 1.7 <-
  old_pos = 3
  new_pos = 3
-> State: 1.8 <-
  old_pos = 2
  new_pos = 4
  change = TRUE
-> State: 1.9 <-
  old_pos = 1
  new_pos = 1
  change = FALSE
  persons.line[2] = 1
  persons.line[3] = 0
  persons.happy[3] = FALSE
  persons.happy[5] = FALSE
  persons.separation_table[1] = 0
  persons.separation_table[3] = 0
-- Loop starts here
-> State: 1.10 <-
  old_pos = 8
  new_pos = 8
-> State: 1.11 <-
  old_pos = 2
  new_pos = 3
  change = TRUE
-> State: 1.12 <-
  old_pos = 7
  new_pos = 5
  persons.line[2] = 0
  persons.line[3] = 1
  persons.happy[3] = TRUE
  persons.happy[5] = TRUE
  persons.separation_table[1] = 1
  persons.separation_table[3] = 1
-> State: 1.13 <-
  old_pos = 6
  persons.line[6] = 1
  persons.line[7] = 0
  persons.happy[4] = FALSE
  persons.happy[6] = FALSE
  persons.separation_table[5] = 0
  persons.separation_table[7] = 0
-> State: 1.14 <-
  old_pos = 5
  change = FALSE
-> State: 1.15 <-
  old_pos = 7
  new_pos = 6
  change = TRUE
-> State: 1.16 <-
  old_pos = 4
  new_pos = 4
  change = FALSE
  persons.line[6] = 0
  persons.line[7] = 1
  persons.happy[4] = TRUE
  persons.happy[6] = TRUE
  persons.separation_table[5] = 1
  persons.separation_table[7] = 1
-> State: 1.17 <-
  old_pos = 3
  new_pos = 3
-> State: 1.18 <-
  old_pos = 2
  new_pos = 4
  change = TRUE
-> State: 1.19 <-
  old_pos = 1
  new_pos = 1
  change = FALSE
  persons.line[2] = 1
  persons.line[3] = 0
  persons.happy[3] = FALSE
  persons.happy[5] = FALSE
  persons.separation_table[1] = 0
  persons.separation_table[3] = 0
-> State: 1.20 <-
  old_pos = 8
  new_pos = 8

```