Instituto Politécnico
de Castelo Branco
Escola Superior
de Tecnologia

ECSMoS

ECS based pedestrian mobility simulatior

Rafael Souza Cotrim

Orientadores

Alexandre José Pereira Duro da Fonte

João Manuel Leitão Pires Caldeira

Dissertação/Trabalho de Projeto / Relatório de Estágio (Deixar apenas a designação aplicável e sem sublinhado e eliminar esta nota) apresentado à Escola Superior de acrescentar nome da unidade orgânica do Instituto Politécnico de Castelo Branco se aplicável acrescentar nome da instituição associada (caso contrário, eliminar esta nota) para cumprimento dos requisitos necessários à obtenção do grau de Mestre em designação do mestrado, realizada sob a orientação científica do categoria profissional do orientador Doutor nome do orientador, do Instituto Politécnico de Castelo Branco.

Data

List of Abbreviations

CSV Comma-separated values.

EC Entity-Component.

ECS Entity Componet Systems.

ECSMoS Entity Componet Systems Mobility Simulator.

FDS Fire Dynamics Simulator.

MVC Model View Controller.

OOP Object-Oriented programming.

SFM Social Forces Model.

SUMO Simulation of Urban Mobility.

UI User Interface.

1. Introduction

Pedestrian Dynamics is an area of study focused on understanding the movement of pedestrians, which often happens as crowds. Such studies can take the form of an analyzing of data collected from studying the movements of people in the real would [SRC], proposition and evaluation of methods for modeling crowd behavior [SRC], predicting movement patterns in certain spaces [SRC], among others [SRC?].

As such, pedestrian dynamics are of practical use when creating spaces meant to be utilized by people. A better understanding of how people move though a structure may aid during the design phase of a building, allowing for better planing of fire escape routes [SRC]. Similarly, not taking into account how people will behave may turn concerts or other large events into deadly crowd crushes or increase the number of trampling incidents [SRC]. Finally, even when there is little risk of loss of life, they may still be useful for incising the throughput of infrastructure such as trains stations, airports and others [SRC].

One of the best tools from this area of study comes in the form of simulation models. Computer simulations allow us to check how pedestrians will behave in a certain environment without having to spend time and resources on the construction and evaluations of scale models on the real world. This reduces costs and promotes fast iterative designs that may better align with the requirements of spaces.

There are many models for simulating the flow of pedestrians in an environment, however, designing and implementing models capable of reproducing phenomena seen on the real world is a complicated task. Pedestrian Dynamics is an inherently interdisciplinary science [SRC] due to its object of study. It relies on concepts from areas as diverse as physics, engineering, psychology, computer science and sociology. Simpler models may take into account only the physical part of crowd movement and ignore all else, while others deal with the effects of having people with different ages [SRC], disabilities [SRC] or even states of mind such as calm and in panic [SRC].

When these models are implemented, it is often done on top of an exiting simulator or framework. These allow model authors to focus on the most relevant parts of their research while other tasks are handled by code already written and validated by others. Nevertheless, building a model on top of these simulators also comes with certain disadvantages. Their architecture imposes restrictions on how the model can operate, meaning that certain simulators may not be compatible with a model because it breaks one or more of the assumptions made when the simulator was being designed. [NOTE: Example? Here or later?]

Similarly, the architecture and technologies used by the simulator have a performance impact on the simulation. While less impactful than full incompatibility between model and simulator, low performance can significantly slow model development and usage. Larger or more complicated sceneries may take much longer than desired to be evaluated or required additional hardware.

While there are many ways of mitigating such problems, using architectures found in other areas such as Entity Componet Systems (ECS) offer some benefits. ECS is a software architecture sometimes used in game development due to their much stricter performance requirement than standard simulators. Users and developers expect such frameworks to be able to handle thousands of entities updating multiple times a second, allowing them to deliver frame rates of upwards of 60 frames per second. Not only that, but, as this work will show, this paradigm is very flexible, allowing it to be used into many scenarios, including the study of crowds. Despite this, ECS is still very uncommon in the scientific world.

With this in mind, I propose a new simulation framework for pedestrian dynamics: Entity Componet Systems Mobility Simulator (ECSMoS). This simulator is based on the ECS architecture and has the objective of being as flexible as possible for model authors and implementers while maintaining a high performance.

[Section description]

[TLDR of results]

2. Current state of pedestrian modeling and simulation

2.1 Pedestrian models

Modeling pedestrians is a complicated affair. Humans have both conscious, subconscious and even physical aspects which affect how one moves. Because of this, there are many competing models in use, each one with its own benefits and limitations. There are a many ways of categorizing them generally. One of the most common is based on the scale of the entities involved [1]. Microscopic models focus on individual pedestrians and their movement, while macroscopic models avoid dealing with interactions between discreet entities, preferring to model the behaviors of whole crowds directly.

Another way of categorizing models is proposed by [2], which divides models into the following groups:

[These could (should?) be expanded]

- Mechanics-based models: Inspired by continuum mechanics or force models.
- Cellular automata models: These interpret the world as discretized units on a grid. Agents operate under defined rules that determine how they move on the grid.
- Stochastic models: Use random or probabilistic models to determine behavior.
- Agency models: Pedestrians are treated as agents that are able to sense and reason about the world. They make choices about where to move depending on their perceptions of the outside world.
- Data-driven models: Based on data collected from experiments and other sources. Uses that for building and calibrating a model.

However, both kinds of divisions are descriptive, not prescriptive. Models routinely fit into multiple or in between. Where they fall is mostly determined by the requirements and constraint of the model authors. Exotic requirements may lead to models being completely out of these constraints.

This thesis will focus mostly on the Social Forces Model (SFM), a microscopic mechanics-based model originally proposed in 1995 [3]. That is because many implementations of it have been proposed and added in various simulators over the years. Providing an even ground for comparisons between simulators. Its behavior is known and well understood. In particular, the version described in [4] will be used, as it is more similar to the implementations present in today's simulators.

The SFM describes interactions between pedestrians and the environment as forces. Pedestrians are modeled as circles and, at each point in time, certain forces applied upon them and are used to compute its acceleration, velocity and ultimately position. Though careful calibration, these forces can reasonably model the behaviors of people moving. The original model defined three main kinds of forces. However, some implementations may add more force, taking into consideration other aspects.

The first kind of force to take into account is the driving or motivation force, computed by equation (1). This is a force that attempts to move the pedestrian in the directions of its ultimate destination. If a lone agent was placed in an empty plane and given a target destination, the motivation force would be a constant vector pointing to its destination. If there are obstacles on the way, the direction of the force might change to steer the pedestrian into the shortest available path. Which is the shortest path needs to be computed by some other means, as the model itself does not specify how.

$$F_i^{\vec{drv}} = \frac{v_i^0 \vec{e_i^0} - \vec{v_i}}{\tau} \tag{1}$$

where:

$F_i^{\vec{drv}}$ is the driving force on pedestrian i

v_i^0 is the desired speed: the speed at which the pedestrian i would prefer to walk

e_i^0 is a unit vector pointing to the desired direction

$\vec{v_i}$ is the current speed of pedestrian i

τ is the reaction time: the time it takes pedestrians to notice changes in theirs surroundings

The environment around an agent may also impose repulsive forces upon them. Pedestrians naturally attempt to keep some distance between themselves and others both for comfort and to guaranty the ability to continue moving forward. This is modeled as a repulsive force between them. The force between pedestrians i and j is symmetrical between then, and the final force (2) on i is the sum of all repulsive forces applied by other pedestrians (3).

$$F_i^{\vec{rep}} = \sum_i \vec{f_{ij}} \tag{2}$$

$$\vec{f_{ij}} = [A_i e^{\frac{r_{ij} - d_{ij}}{B_i}} + kg(r_{ij} - d_{ij})]\vec{n_{ij}} + \kappa g(r_{ij} - d_{ij})\Delta v_{ji}^t \vec{t_{ij}} \tag{3}$$

where:

$F_i^{\vec{rep}}$ is the total repulsive force from other agents on i

$\vec{f_{ij}}$ is repulsive force from j on i

$r_{ij} = r_i + r_j$ is the sum of the radius of i and j

d_{ij} is the distance between the center of i and the center of j

g is the contact distance between the pedestrians. If they are not touching, it is 0, otherwise it is the distance between the center of i and the center of j

$\vec{n_{ij}}$ is a unit vector pointing from j to i

$\vec{t_{ij}}$ is a unit vector perpendicular to $\vec{n_{ij}}$, rotated counterclockwise

$\Delta v_{ji}^t = (\vec{v_j} - \vec{v_i}) \cdot t_{ij}$ is the tangential velocity difference of i and j

A, B, k, κ are calibration constants

In practical terms, the repulsive force has two main components: a normal and a perpendicular vector. The normal forces two pedestrians away from each other, increasing depending on how close they are. If the pedestrians are close enough to touch, that force additionally increases by a secondary factor that is dependent on how much they intersect. In this way, the normal component captures both the psychological desire for space and the physical constraints preventing pedestrians from packing too tightly. The perpendicular vector represents sliding forces. When two pedestrians touch, there is some friction between them that acts to slow them down. Friction acts perpendicularly to the normal and against the direction of movement. These forces can be seen on Figure 1

The final class of force to consider is the obstacle force. Much like the previous repulsive forces, agents also attempt to keep a certain distance from them to walls and other obstacles in their surroundings. If they touch those objects, there is also an additional factor that increases the normal repulsion and a factor

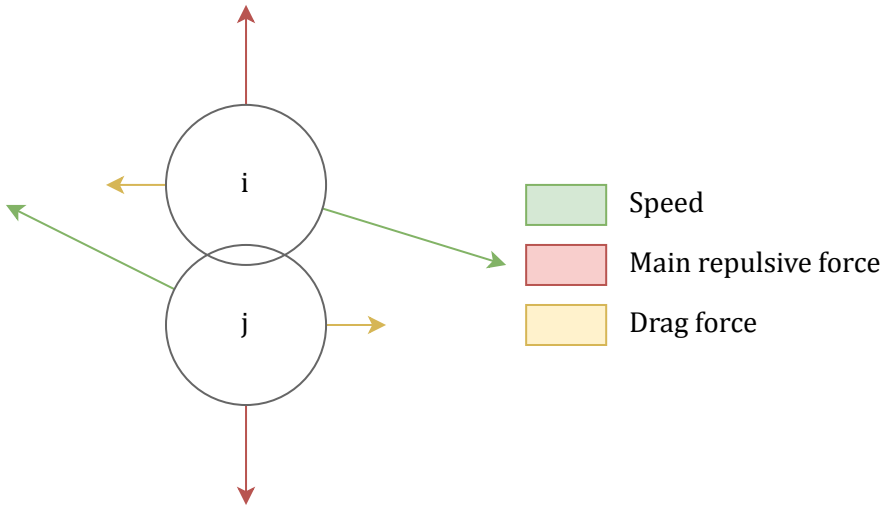

Figure 1 – Diagram of repulsive forces between pedestrians i and j

describing friction. However, most obstacles are not perfect circles, so the function uses the closes point of the obstacle for distance measurements.

$$\vec{F_i^{obst}} = \sum_o \vec{f_{io}} \tag{4}$$

$$\vec{f_{io}} = [A_i e^{\frac{r_i - d_{io}}{B_i}} + kg(r_i - d_{io})]\vec{n_{io}} + \kappa g(r_i - d_{io})(\vec{v_i} \cdot \vec{t_{io}})\vec{t_{io}} \tag{5}$$

where:

$\vec{F_i^{obst}}$ is the total force from obstacles on agent i

$\vec{f_{io}}$ is the force from obstacle o on i

r_i is the radius of i

d_{io} is minimum distance from the center of i to any point of o

g is the contact distance between the pedestrian and the obstacle. If they are not touching, it is 0, otherwise it is equal to d_{io}

$\vec{n_{io}}$ is a unit vector pointing from the center of i to the closes point of o

$\vec{t_{io}}$ is a unit vector perpendicular to $\vec{n_{io}}$, rotated counterclockwise

A, B, k, κ are calibration constants

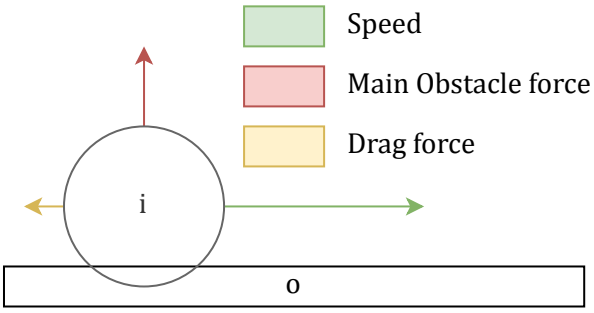

Figure 2 – Diagram of obstacle forces between pedestrian i and obstacle o

In each simulation step, all of these forces are computed for each agent, then they are used to compute its new speed and position. Using basic Newtonian physics, equations (6) and (7) can be derived. Once the agents are moved to their new locations, the simulation forces are calculated once again. This process repeats until the desired results are achieved.

$$\vec{v_i^{new}} = \vec{v_i} + (F_i^{drv} + \frac{\vec{F_i^{rep}} + \vec{F_i^{obst}}}{m_i}) \cdot \Delta T \tag{6}$$

$$\vec{pos_i^{new}} = \vec{pos_i} + \vec{v_i^{new}} \cdot \Delta T \tag{7}$$

where:

$\vec{v_i^{new}}$ is the new speed of the agent i

$\vec{v_i}$ is the previous speed of i

$\vec{F_i^{drv}}$ is the driving force on pedestrian i

$\vec{F_i^{rep}}$ is the total repulsive force from other agents on i

$\vec{F_i^{obst}}$ is the total force from obstacles on agent i

m_i is the mass of i

ΔT is the length of the simulation step

$\vec{pos_i^{new}}$ is the new position of i

$\vec{pos_i}$ is the previous position of i

Despite its relative simplicity, this model is quite robust and is capable of replicating certain phenomena that are seen in imperial studies. For example, it replicates the "Faster-Is-Slower Effect", which happens when the flow rate of pedestrians though an obstruction paradoxically decreases once a certain speed is surpassed [4]. The SFM reproduces lane formation, which is when pedestrians form lanes of people going in the same direction when two streams of people converge in a limited space [SRC].

2.2 Existing simulation frameworks

[This needs a better "warm-up" before just going into the text itself]

While there are many commercial products for simulating the behavior of pedestrians, they are all for the most part closed source and proprietary, meaning that it is difficult to scrutinize their results or implement new models. Solutions like this include [List some commercial solutions]. Due to their limitations, scientific research tends to focus on more open simulators.

To determine which simulators were most used and relevant to this work, a survey was performed. Various scientific article databases were searched with terms relevant to this area of study and the simulations frameworks used were noted. A substantial portion of the works used custom-made software for the specific use case the researchers had in mind. Of the general simulation frameworks found, only open source ones with at least one update since 2015 were considered. This last requirement is to guarantee that the evaluated simulators are either in active development or occasionally receive additional updates.

Cromosim is a Python library for crowd simulations [5]. While it benefits from Python's large ecosystem of scientific tool, Cromosim has little pre-built infrastructure when it comes to helping researches. Even basic models such as the social forces model are not already included.

FDS+Evac is a module for Fire Dynamics Simulator (FDS) developed by Technical Research Center Of Finland for the purposes of simulating pedestrian moment in building fire scenarios [6]. It is agent-based,

uses the social forces model and is mostly suited for evacuation scenarios on buildings with relatively flat floor plans. Buildings with a large degree of vertical movement withing a single floor (stadiums, concert halls and cinemas for example), may present some challenges. However, this module has been discontinued as of version 6.7.8 of FDS.

jCrowdSimulator is a pedestrian simulator written in Java and maintained by the Fraunhofer Institute for Transport and Infrastructure Systems [7]. It can be used in the form of a library or as a stand-alone application. Currently, it only implements the social forces model and its implementation is tightly coupled to that rest of the simulator, meaning that adding new models can be a challenge.

JuPedSim is a Python package with a C++ core for simulating pedestrian dynamics. Despite not having a User Interface (UI) [8], JuPedSim is simple to use. It also includes various modules for flow/density analysis, image generation, 2D or 3D animations, reporting and geometry generation. Model implementation is divided into three levels. (1) Tactical: route choice and short term decisions. (2) Strategic: Long term decisions and general objectives. (3) Operational: How to perform each action. JuPedSim can be executed inside Jupyter Notebooks for a more interact experience. While extremely flexible for users, the fact that its codebase is split between languages and that its internal structure is tightly bound to the implemented models poses problems for developers.

Menge is a modular framework for simulating crowd movement that decomposes the problem of how to simulate human movement into four main tasks [9]. (1) Goal selection: Deciding what each pedestrian wants to achieve. (2) Plan computation: Deciding on a strategy to achieve a pedestrians objective. (3) Plan adaptation: Adapt the original plan to the current local environment of the pedestrian. (4) Spacial queries: How to retrieve data around a certain point and deciding what kinds of data are useful. Figure 3 provides some additional detail on how these parts interact. To add a new model to the simulator, it is necessary to break it down into these four components while also obeying the other interface restrictions. If the new model is similar enough to another in one of these areas, it is possible to reuse that part of the implementation. However, implementing models that do not fit well into these divisions can be ~~chalenging~~challenging. Menge includes a UI and various examples of its basic features. However, its last stable version was released in 2017 and all development stopped on 2019.

~~Menge architecture diagram~~

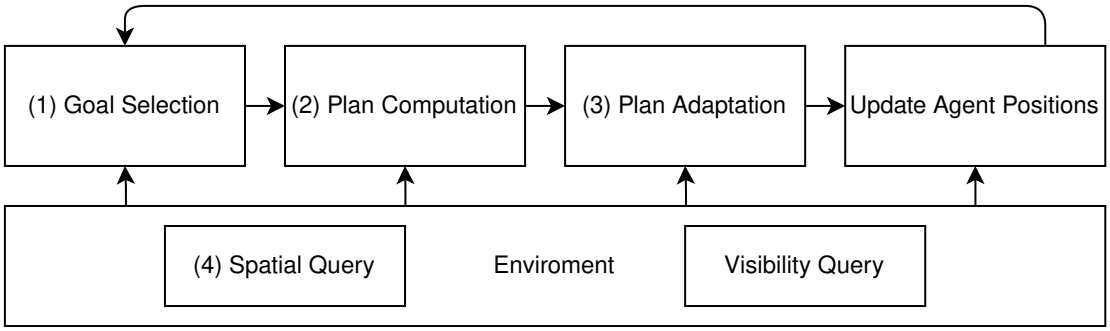

Figure 3 – Menge architecture and simulation flow. Adapted from [9].

MomenTUMv2 is a microscopic agent-based pedestrian simulator written in Java and developed by the Technical University of Munich [10]. Most of its inner workings are implemented in the form of models, which are thought of as "operation providers". These are implementations of the Strategy and Template Method software patterns. This modularity is very broad, including configuration, pedestrians generators/removers, analyzers and much more. Models that do not fit into any pre-defined category can be added in a special section. When compared to other solutions such as Vadere, it is much more adaptable, but it comes with a cost in terms of complexity and implementation time. One of its main benefits is that it allows for the use of hybrid models, meaning models that connect two or more pre-existing

models of the same layer to create behavior that is a mixture of them depending on the circumstances. Finally, it has seen very little change since 2018 ~~with~~ and its last update was on 2020. Since them, there has been no development activity, and its lead developer no longer works at the Technical University of Munich. The most recent available versions of the code are not able to be executed due to the incomplete implementation of certain features.

[Momentum architecture diagram]

Simulation of Urban Mobility (SUMO) is a traffic simulator widely used in traffic engineering and related topics [11]. While it is mostly focused on vehicular movement, it does have some built-in mechanism to simulate pedestrian movement. However, these were built with the purpose of enriching the traffic simulations and are very limited. For example, pedestrians are not allowed to freely roam in the 2D plane, instead they are restricted to certain lines/paths. For more robust simulation capabilities, it is possible to use JuPedSim toughener with SUMO, however this integration is still work in progress.

Vadere is a situation framework for comparing different locomotion model developed by the Munich University of Applied Sciences [1]. It was designed to be lightweight, but while still reframing relatively flexible. It contains both a simulation engine that can be executed via command line or via its own UI. As shown on Figure 4, Vadere following the Model View Controller (MVC) architecture where the UI serves as the view, the state is the model and the simulator core is the controller. Inside the simulation core, models are not implemented following any specific division of responsibilities like those seen in other simulators. Instead, it uses a more integrated approach. However, Vadere was designed to use a mechanism called floor fields to compute the paths of pedestrians. While this implementation has its benefits [12], it imposes a performance drop on cellular automata models [1]. Attempting to use other approaches would require rewriting large portion of its internal workings.

~~Vadere architecture diagram~~

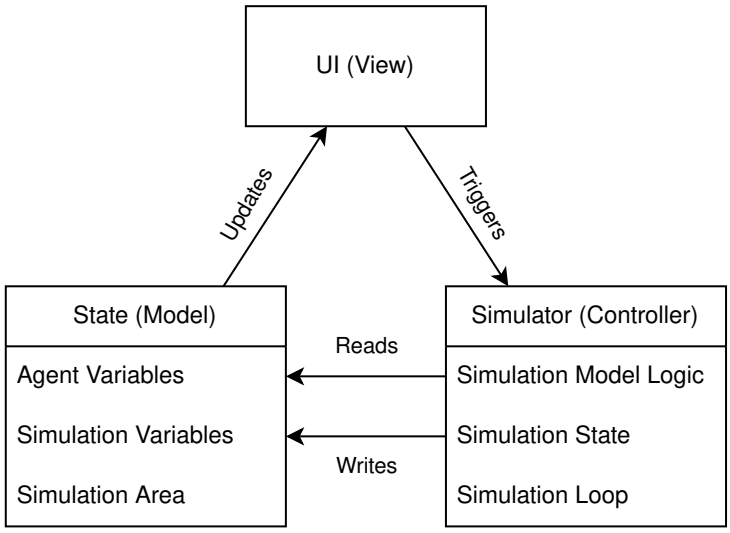

Figure 4 – Vadere architecture. Adapted from [12]

Finally, there are other libraries such as **Mesa** [13], **Agents.jl** [14] and many others, which focus on agent based modeling. While these can be used for pedestrians simulations, they have little to no pre-built infrastructure for model authors, requiring a full implementation to be done for any comparison. However, due to their larger audience, they may have better tools for analysis and documentation that other purpose built simulators.

Despite the variety of available tools, most of these are only compatible with a subset of models. Menge has well-defined structure that facilitates development, but it only works well when a model can be split into its four types of tasks. Vadere and JuPedSim are very popular in general. However, their internal

architecture is too monolithic, preventing code from being effectively shared between pre-existing models and new implementations. MomenTUMv2 is possibly the most modular, however this is achieved via a lot of complexity and its. Finally, libraries like Agents.jl are very capable of adapting themselves to many different kinds of modes, but lack any infrastructure for pedestrians simulations specifically.

As a result of these limitations, many researches on the field end up creating their own purpose built simulations. This makes comparison much harder, as certain implementation specificities can have impacts that are hard to predict and take into account in a comparison. Removing these limitations could greatly improve comparability and convenience for model authors.

3. The ECS architecture

The ECS architecture is a data-oriented design pattern or framework sometimes used in the game development industry due to its flexibility, modularity, and high performance. However, it is hard to find concrete numbers when it comes to its use as most games are closed source and their inner workings are not widely discussed. It also has a loose definition and is sometimes confused with other similar software architectures that are also common in the area, most notably Entity-Component (EC). Additionally, the various implementations can be substantially different due to their underlying technologies and objectives. Finally, there are few academic sources for this topic [this needs more research!], most works on this subject are blogs, videos and personal anecdotes. Because of these factor, this section will discuss the more common aspects of the architecture with details of the specific implementation used for ECSMoS discussed in section 4. Nevertheless, this architecture has the potential of addressing many of the problems raised in subsection 2.2, and merits further exploration.

3.1 General structure

Generally speaking, in a ECS implementation, most things is divided into three main kinds of concepts: entities, components and systems. **Entities** represent general objects in the world. For example, in a crowd simulation, each pedestrian would be an entity, but so would obstacles and other things that might affect them. Usually, they consist of a simple identifier which can be used to group other kinds of data. Entities usually do not carry additional data, but many implementations allow entities to be "disabled", meaning that they are not used/visible to other parts of the architecture.

Components can be considered the attributes of entities. Each component is associated with only one entity and stores the data relevant for a certain aspect of its behavior. In most implementations, each entity may also only have one component of each type associated with it. Following the previous example, each pedestrian would have a position component and a speed component, each of which contains all the data necessary to characterize the entity on the relevant aspect. Entities may have as many components as necessary to describe their behavior. However, components do not contain logic. On the other hand, some components also do not store data within them, their presence is already the data itself. For example, an entity that is a pedestrian may have an empty `Pedestrian` component, which serves as only a marker or tag for filtering.

Systems are processes that can read and modify components. They are the main place for logic in this architecture, they usually have a well-defined singular purpose and can access all data as though it was global. For example, a simple system might move entities to their next position according to their speed component. Systems are executed in a pre-defined order that may be specifically determined by the developer or derived by some set of rules. One common implementation of rules-based ordering is to define the dependencies of each system, meaning systems that must be executed before each one starts. This allows the framework to defined by itself when each system is executed. Once all systems have been

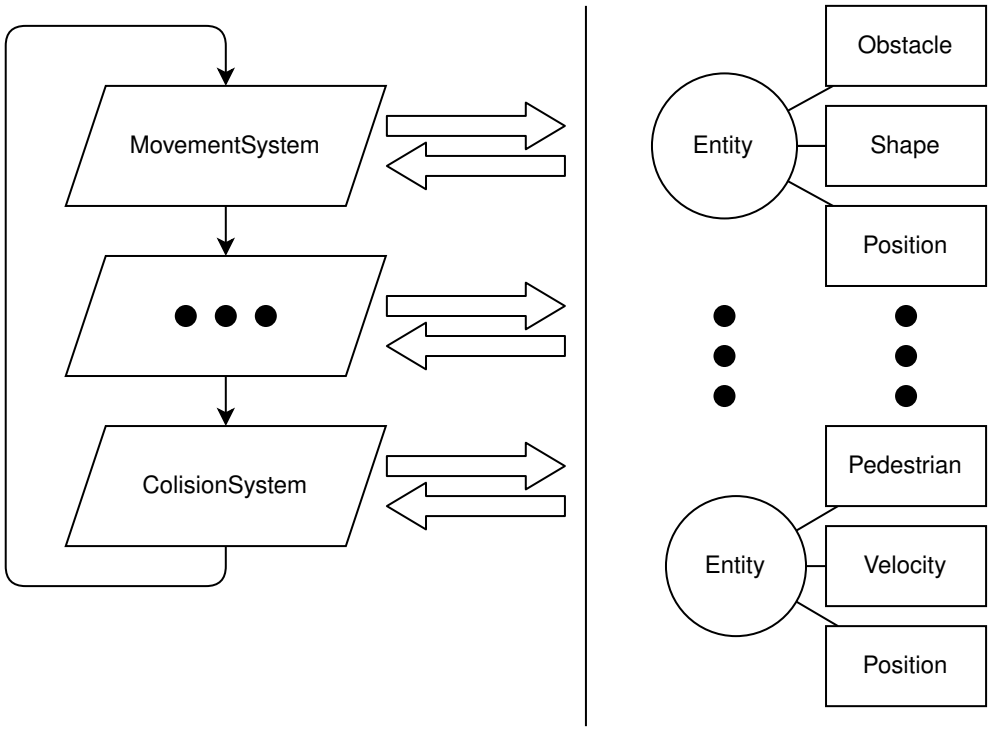

Figure 5 – Example of how an ECS project is divided. On the right, there are entities and their related components. On the left, systems are shown. Systems are executed according to their specified order and read/write data to and from the components.

executed, the cycle is started once more from the first system, creating a loop. Systems are also usually capable of filtering the entities that they want to read data from or operate on by using the presence or absence of certain components.

Another way of understanding these concepts is to compare them to a relational database. In this paradigm, each component type has its own table and uses the entity it is associated with as primary key. A system then would perform queries for the data in database. The entity information can be used for joining data from different component tables and the presence or absence of each component can be used to filter out certain entities. In this way, systems can perform operations only on entities which are relevant to the systems function. A system may also write back to the database and following systems can access the newly written data. This general structure can be seen in Figure 6.

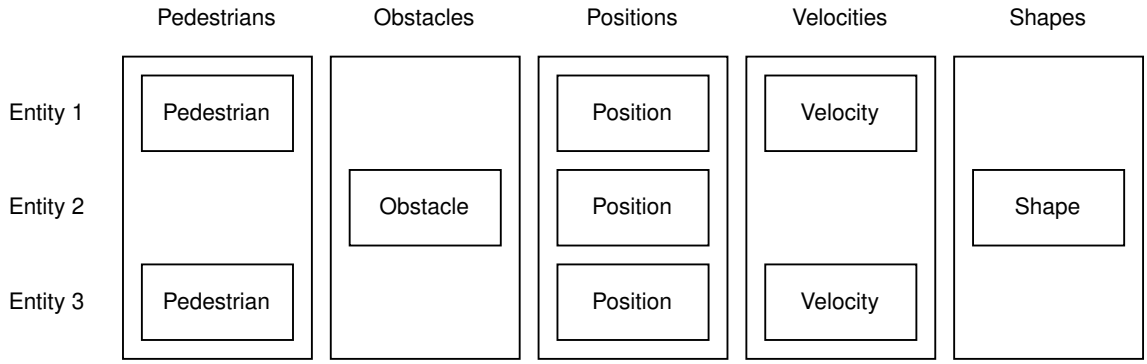

Figure 6 – Example of how an ECS can be seen as a database. [This possibly needs a citation to https://c-sherratt.github.io/blog/posts/specs-and-legion/]

Depending on the details of how components are stored, this architecture maximizes the efficiency of the CPU cache when compared the architectures of other simulators, witch gives it a substantial performance advantage. This happens because the data for each component type can be stored in a single place, usually an array, and can be directly loaded into the cache when a system starts its execution, meaning that it has high cache spacial locality. As a result, the CPU does not need to check the RAM as often, witch reduces the time it takes to retrieve data and consequently improves performance. For comparison, in simulators such as Vadere, the update loop first goes through each entity, then it is able to load the data for each individual property. As a result, the CPU has to go through noncontinuous regions of memory, increasing the number of cache misses.

In addition, the fact that systems are independent allows for a certain level of parallelism to take advantage of multithreaded systems. This comes in two forms: Internal and external parallelism. External parallelism is when two or more systems may be executed at the same time. This can happen when the systems do not modify any pieces of data that are used by each other and there is no logical requirement for their execution to be sequential. Such an approach is useful when multiple kinds of data are necessary for one operation, but computing them is time-consuming. The expensive computations can be split off into multiple different systems that save their data to intermediate components. After all these computation systems have concluded their execution, the final system gathers the intermediate components and performs the operation. Internal parallelism can occur when it is possible to process each entry independently of each other. In this case, a system can split the data it has received into multiple groups that can be processed in parallel.

While equivalent techniques could be employed on existing simulators, such changes would be hard to archive and prone to casing issues. In ECS, these changes are much simpler. Internal parallelism is possible in a large portion of cases, and it is much easier to implement, although it may not be worth it for simple and inexpensive systems. External pararalesim is somewhat harder because it requires an analysis of the inputs of the systems to prevent concurrent modifications to the same piece of data or to handle it gracefully. However, some ECS frameworks such as Bevy ECS [15] allow for a seamless implementation of these concepts.

[external and internal parallelism diagram (make sure to show how dependencies between systems work)]

In addition to the performance benefits, ECS promotes adaptable and reusable design. Components work best when they represent a singular aspect of each entity and are very simple. Such components can be composed into multiple kinds of entities and, because the systems use the components associated with an entity to decide if they will perform operations on them, the behavior of the entities can be changed mid-simulation. For example, a pedestrian may have a `Positiona` and `Speed` component, which causes the `SpeedSytem` to make it move according to its speed. If the pedestrian then decides to sit, the `Speed` component can be removed, witch automatically removes it from the `SpeedSytem`'s list of entities to operate on. Also, if it is desired to add other kinds of moving objects to the simulation, cars for example, it is not necessary to create new logic to handle their movement. Any entity with `Positiona` and `Speed` will automatically be affected by the `SpeedSytem`.

Despite these advantages, ECS also poses some problems which have limited its acceptance even on the fields where it sees some use. The main one is lack of familiarity of programmers who use Object-Oriented programming (OOP) and other common programming paradigm's with the design principals it is build upon. For example, while ECS can be implemented in OOP focused languages, it breaks many of OOP's principals. Making effective use of this architecture requires reframing problems to avoid some aspects such as inheritance and polymorphism. This problem compounds on itself, as the lack of popularity means that, generally speaking, tooling for this architecture is less developed than others and their is less information about it, which naturally perpetuates its status.

ECS also has some problems inherit to its structure. Due to the fact that component data is effec-

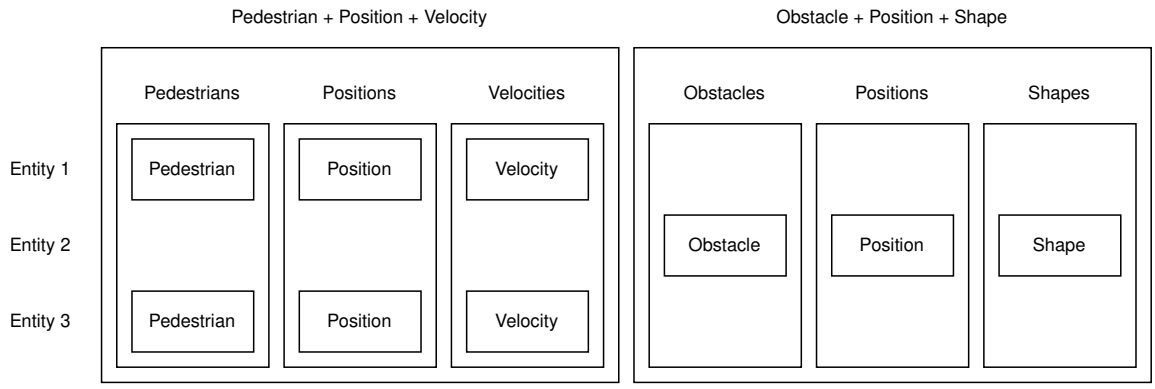

Figure 7 – Example of how archetype-based ECS stores components.

tively global, tracking down issues caused by improper manipulation of data is hard, as they could come from anywhere. Additionally, observing and understanding the global data is a complicated affair during debuging, as the data for each entity is spread around multiple places and using possibly using multiple forms of storage, making it hard to have a wholistic picture of what the systems state is.

[Maybe this advantage and disadvantages sections should come after the component storage strategies]

3.2 Component storage strategies

While all ECS implementation have the same basic structure, one of the main differentiating factors between then is the underlying method for storing the component data and their connections to an entity. Since querying entity and component data is one of the most performed operations on this paradigm, having a well-thought-out method for storing and retrieving such data is of utmost importance. The following is a non-exhaustive categorization of ECS implementations when it comes to this aspect:

Archetypes: Implementations that rely on archetypes group entities based on the components they have. Each combination is called an archetype and is stored in a contiguous area of memory, often as a struct of arrays, meaning that they have great cache spacial locality. When a system requires data, the framework checks witch archetypes conform to the requirements and provides the data. This strategy has the best iteration performance of the discussed implementations. However, adding or removing components from an entity forces it to move all the data related to that entity to another archetype. This reduces the efficiency of such operations. Examples that mainly use this approach include Flecs [16], Unity DOTS [17], Bevy ECS [15], Legion [18] and Hecs [19]. For a visual example, see Figure 7.

Sparse set: On sparse set implementations, a sparse set is maintained for each component type. Sets use entities as keys to retrieve components and the framework checks all the sparse sets for the components each system requests to gather the data. As a result, iteration over the entities is much slower due to lack of cache locality, however adding or removing components becomes much faster. Entt [20] and Shipyard [21] are some of the most well known frameworks that use sparse sets. This approach is the most similar to the one depicted in Figure 6.

Bitset based: As the name suggests, bitset based implementations work by using a bitset to say if an entity has a certain component. This can be done by having arrays that contain the component data and a bitset for each entity, or it can use other data structures such as hierarchical bitsets, which may provide better iterative performance. EntityX [22] and Specs [23] are some notable examples in this area.

Due to the many benefits and drawbacks of each approach. Some ECS frameworks allow for the storage type to be defined in a component by component basis. Such an approach allows developers to maxi-

mize the performance of the system. Generally speaking, most components should be stored in something similar to what the archetype implementation use, which gives the best possible iterative performance. However, components that are frequently added and removed in short succession can be stored similarly to sparse set or bitset implementations to reduce the overhead of these operations. Examples of frameworks that allow for such a nuanced approach include Bevy ECS [15] and Specs [23].

4. ECSMoS implementation

Considering the problem of other simulators referenced in subsection 2.2, the main objective of ECSMoS was to be more flexible than existing simulators. It should accommodate the addition of new models easily, should remove as many constraints for models authors as it is possible, allow for the creation of variants of existing models and the creation of hybrid models. Additionally, it should be at least as performant as existing solutions. While there are other factors that greatly influence the usability of a simulator such as UI, documentation and integrations with other tools, these were not focused upon for this work. Any implementation for these pr other areas was merely made for the purpose of supporting the goals of the project.

~~I need to cite what is NOT an objective~~

The main technological choice for the implementation of ECSMoS was the specific ECS framework to be used. While there was the option of creating a bespoke implementation, it would likely be much less mature, stable and featurefull than the ones already available. Of the available options, Bevy ECS, a ECS framework implemented in the Rust programming language, was chosen. This was due to how Rust and Bevy ECS complement each other, allowing for seamless external parallelism enabled by the compile-time guarantees provided by Rust's borrow-checker and explicit declaration of mutable values. These features allow Bevy to know at compile time which components and entities are accessed by any given system and whether they can perform changes to them. As such, a dependency graph can be constructed and Bevy can use that to prevent the execution of systems that conflict with each other (such as those which write to the same components) from being executed in parallel while allowing the non-conflicting ones to do so.

In addition to the standard ECS constituents of entities, components and systems, Bevy also provides a few more features. **Schedules** serve as management units for systems. They defined what systems are executed, what is their execution order and under what conditions they are active. **System Sets** are system organizational units. They can be used to group systems that have similar functions or by any other differentiating factor. **Resources** are a way of storing data globally without association with a specific entity and are useful for representing global state. Bevy also has a built-in **Events** framework, which allows communication between systems without needing to update components. Finally, Bevy has an advanced change detection system and developers are able to use that to only execute certain systems when certain components are added or changed, reducing computational load. These and many other features are all fully compatible with the Bevy's parallelization mechanisms, allowing full use if multithreaded hardware.

4.1 ECSMoS Structure

To improve organization and effectively provide a division of concerns, ECSMoS is built out of modules, which are built on top of what Bevy calls plugins. Each module can add its own systems, resources, components, etc. to the simulation. Modules are not executed in any particular order, each of their systems can individually be configured to be executed before or after others. In cases where modules have a dependency on one another, meaning that a system in module A must be executed after a system in module B, system sets can be used to define this dependency behavior.

~~This dependency behavior needs a visual+text explanationFor example ...~~ Bellow is a list of the currently available modules and their general function:

- Default module: Enables basic internal Bevy modules useful for the general simulation framework
- Auto End Simulation module: Adds functionality for automatically stopping a simulation under certain conditions such as when there are no more pedestrians or when a certain amount of time has passed
- Display module: Enables the UI of the simulator
- Flow Field Pathfinding module: Used to compute the best paths for pedestrians on any given point to their destination. The data provided by this module is later used by the Social Forces module to compute the direction of the motivation force. This module is heavily inspired by the floor fields used on Vadere for the same purpose.
- Kinematics module: Handles basic movement according to the laws of physics. While other modules may handle the computation of forces, this module actually moves pedestrians according to their final speed.
- Movement Tracking module: Used for collecting and exporting data related to the movement of pedestrians in the simulation.
- Simple Objective module: Used for determining if a pedestrian has arrived at its destination (called an objective inside the simulation) and what actions should be taken next, most commonly removing it from the simulation.
- Simulation Area Module: Used for defining the region where the simulation takes place. Pedestrians that attempt to leave the simulation area prevent from leaving. The information provided by this system are also useful for Flow Field Pathfinding, which uses it limit the paths it needs to compute.
- Social Forces Module: Implements the social forces model for simulating pedestrians.
- Spawner Module: Used for creating pedestrians at a defined region at a certain frequency, providing a constant flow of agents in the simulation.
- Start Time Module: Simply records the time at which the simulation started. This information is used during the export for naming and differentiating various simulation runs.

Figure 8 shows an example of how inter-module dependencies and system sets operate. It is important to not that it does not show all systems and the general flow has been simplified for better understanding. The image shows two modules, their systems, system sets and how they interact. The rounded boxes represent pre-defined systems sets and the arrows between the systems and systems sets represent their dependencies. Systems without a dependency such as `Compute Obstacle Colision Map` can start execution as soon as possible and, in this example, can also be executed in parallel. Bevy executes systems in parallel nor particular order or preference, and it can be changed from iteration to iteration. The height of each system denotes the period of time when it can be executed. As it is possible to see, `Compute Motivation Force` depends on the Flow Field Pathfinding module's system set, meaning that it will only be executed after it as finished, unlike the other social forces systems.

In the event that it becomes necessary to create a new module that has systems that must be executed after the first three systems of the Flow Field Pathfinding module, but that does not need to wait for the remaining systems, the current systems sets are not sufficient. In this situation, the new module could create a new system set that exists on top of the current structure. This set can then only include the desired systems and be used for defining dependencies. Figure 9 shows an example of this. This new

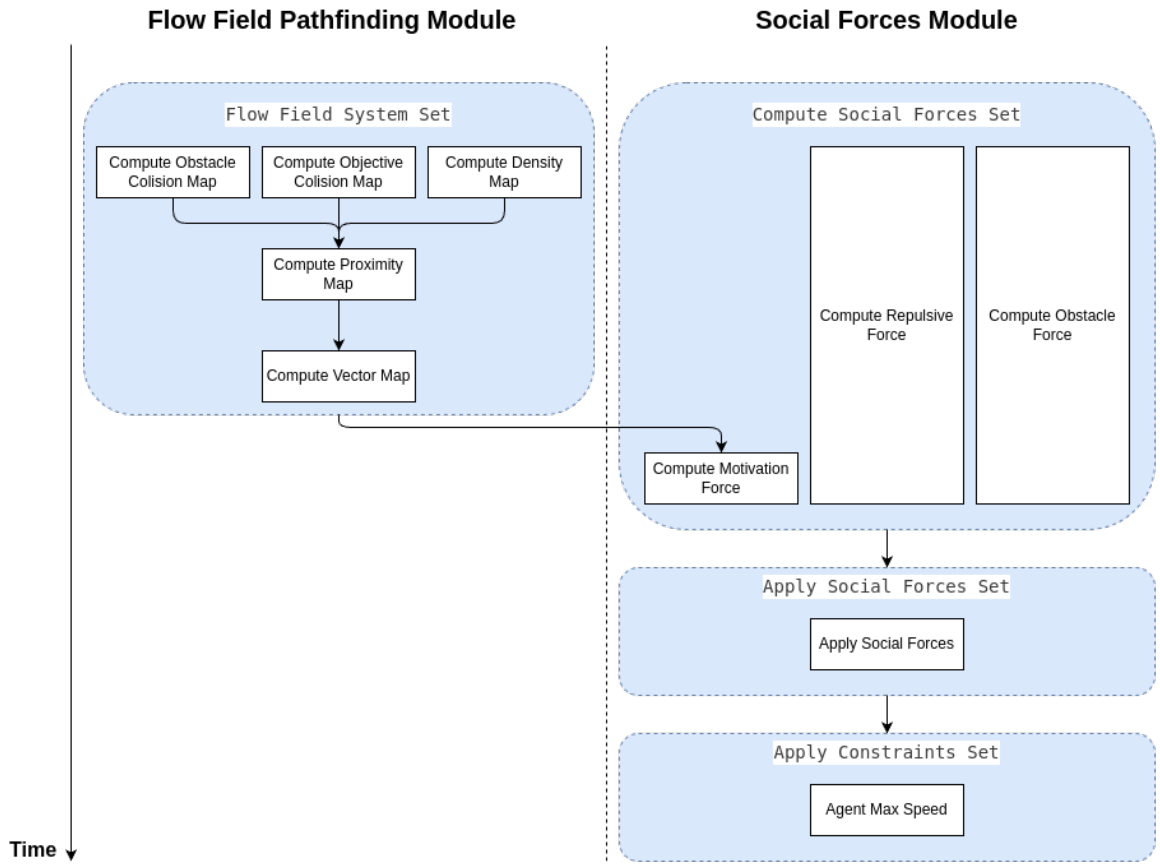

Figure 8 – Example of how ECSMoS handles inter-module dependencies

set can be anonymous and as such remain in use only inside the module, or be given and identifier that would allow it to be used by other modules. Either way, it has no impact on the pre-existing system set or any of its relationships.

While generally a module is added as a whole. It is also possible to add only certain parts of it. For modules where there is a clear expectation that certain parts are only needed sometimes, it is possible to pass configuration variables to the module that enable or disable certain sections. If there was no expectation of this need during development, it is possible to bypass standard module initialization and add the required parts manually. While more time-consuming, this allows a large degree of control even over modules created by third parties. This is particularly useful when creating derivatives of pre-existing modules, as it allows the child module to only keep the parts of the parent that it desires while removing the others.

4.2 ~~Features?~~

In its current state, ECSMoS has a simple UI that is useful for debugging and visualizing the results during execution. The interface is built on top of Bevy's default rendering system which is relatively restrictive, but it is possible to add other UI systems such as Bevy EGUI [24]. This rendering functionality is not directly tied to the simulation state, there are systems that are executed at the end of each simulation loop that translate simulation data into values comprehensible by the UI systems. As such, it is possible to add interface elements that are model specific.

Like Vadere, ECSMoS does not have unified data collection or output mechanism. Both use data collectors which only store certain fields and post processors that output the collected data to one or

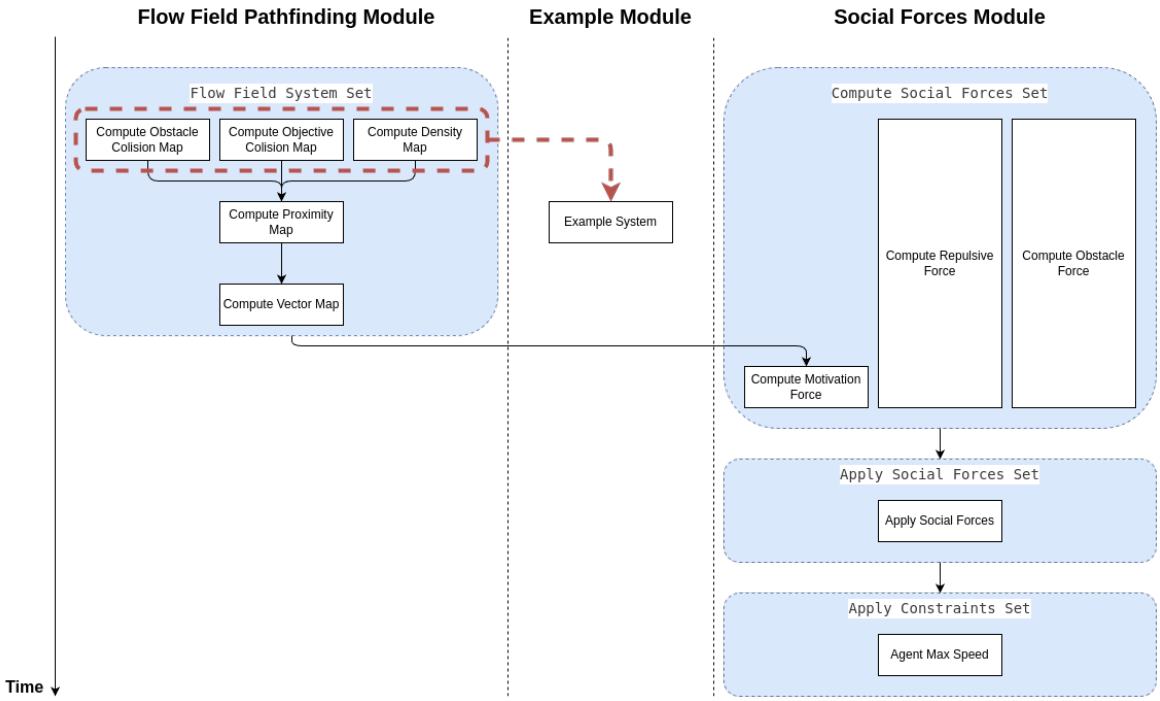

Figure 9 – Example of how ECSMoS handles inter-module dependencies where there is not pre-defined system set grouping only the required systems. The thick dashed and rounded square represents the unnamed set created.

multiple files. In both cases, a trajectory data collector is provided that is able to track the positions of pedestrians and export them in the Comma-separated values (CSV) format or equivalent. However, more complex data collection may require additional systems to be created.

5. Evaluation

5.1 Evaluation criteria

[For example. Imagine when want to simulate a crosswalk. In this case we need an obstacle that exists than disappears. In Vadere, this would need to be done inside the model itself, witch seems out of scope [are you sure this is how it would work?] because changing the simulator itself is hard. In this architecture, it could be added with another system and a few components. The same goes for one way paths. Menge could do this because it splits the problem into parts [I think], but Vadere has a problem. This is an interesting example. It shows how ECS is kind of a natural progression. Vadere works as a monolith. Menge has some spliting, but it may be restrictive if you want more granularity or somethign that crosses the boundaries. ECSMoS has "infinite" granularity and agregation capabilities]

6. Conclusion

References

[1] Benedikt Kleinmeier et al. “Vadere: An Open-Source Simulation Framework to Promote Interdisciplinary Understanding”. In: *Collective Dynamics* 4 (Sept. 3, 2019), pp. 1–34. ISSN: 2366-8539. DOI: 10.17815/CD.2019.21. URL: https://collective-dynamics.eu/index.php/cod/article/view/A21 (visited on 11/08/2024).

[2] Francisco Martinez-Gil et al. “Modeling, Evaluation, and Scale on Artificial Pedestrians: A Literature Review”. In: *ACM Comput. Surv.* 50.5 (Sept. 2017), 72:1–72:35. ISSN: 0360-0300. DOI: 10.1145/3117808. (Visited on 05/17/2025).

[3] Dirk Helbing and Peter Molnar. “Social Force Model for Pedestrian Dynamics”. In: *Physical Review E* 51.5 (May 1, 1995), pp. 4282–4286. ISSN: 1063-651X, 1095-3787. DOI: 10.1103/PhysRevE.51.4282. arXiv: cond-mat/9805244. URL: http://arxiv.org/abs/cond-mat/9805244 (visited on 05/17/2025).

[4] Dirk Helbing, Illés Farkas, and Tamás Vicsek. “Simulating Dynamic Features of Escape Panic”. In: *Nature* 407 (Sept. 28, 2000), pp. 487–490. DOI: 10.1038/35035023.

[5] Sylvain Faure. *Cromosim: Crowd Motion Simulation*. Version 2.1.0. Dec. 30, 2024. URL: https://github.com/sylvain-faure/cromosim.

[6] Timo Korhonen and Simo Hostikka. *Fire Dynamics Simulator with Evacuation: FDS+Evac: Technical Reference and User’s Guide*. VTT Working Papers. Espoo: VTT Technical Research Centre of Finland, 2009.

[7] Axel Meinert et al. “Simulation von Menschenmengen Im Urbanen Umfeld Mit OpenStreetMap-Daten”. In: *gis. Science* (2019).

[8] Armel Ulrich Kemloh Wagoum et al. “JuPedSim: An Open Framework for Simulating and Analyzing the Dynamics of Pedestrians”. In: Dec. 2015.

[9] Sean Curtis, Andrew Best, and Dinesh Manocha. “Menge: A Modular Framework for Simulating Crowd Movement”. In: *Collective Dynamics* 1 (Mar. 15, 2016), pp. 1–40. ISSN: 2366-8539. DOI: 10.17815/CD.2016.1. URL: https://collective-dynamics.eu/index.php/cod/article/view/A1 (visited on 11/08/2024).

[10] Peter M. Kielar, Daniel H. Biedermann, and André Borrmann. “MomenTUMv2: A Modular, Extensible, and Generic Agent-Based Pedestrian Behavior Simulation Framework”. In: (Jan. 2016).

[11] Pablo Alvarez Lopez et al. “Microscopic Traffic Simulation using SUMO”. In: *The 21st IEEE International Conference on Intelligent Transportation Systems*. IEEE, 2018. URL: https://elib.dlr.de/124092/.

[12] Michael J. Seitz et al. “The Superposition Principle: A Conceptual Perspective on Pedestrian Stream Simulations”. In: *Collective Dynamics* 1 (Mar. 15, 2016), pp. 1–19. ISSN: 2366-8539. DOI: 10.17815/CD.2016.2. URL: https://collective-dynamics.eu/index.php/cod/article/view/A2 (visited on 11/26/2024).

[13] Ewout ter Hoeven et al. “Mesa 3: Agent-based Modeling with Python in 2025”. In: *Journal of Open Source Software* 10.107 (Mar. 2025), p. 7668. ISSN: 2475-9066. DOI: 10.21105/joss.07668. (Visited on 07/21/2025).

[14] George Datseris, Ali R. Vahdati, and Timothy C. DuBois. “Agents.jl: a performant and feature-full agent-based modeling software of minimal code complexity”. In: *SIMULATION* 0.0 (Jan. 2022), p. 003754972110688. DOI: 10.1177/00375497211068820. URL: https://doi.org/10.1177/00375497211068820.

[15] Bevy Engine. *Bevy*. Version 0.16.0. June 24, 2025. URL: https://bevyengine.org/.

[16] SanderMertens. *Flecs*. Version 4.0.5. Mar. 18, 2025. URL: https://www.flecs.dev/flecs/.

[17] Unity Technologies. *DOTS - Unity’s Data-Oriented Technology Stack*. Version 6.1. June 10, 2025. URL: https://unity.com/dots.

[18] Amethyst Foundation. *Legion*. Version 0.4.0. Feb. 25, 2021. URL: https://github.com/amethyst/legion.

[19] Benjamin Saunders. *hecs*. Version 0.10.5. May 5, 2024. URL: https://github.com/Ralith/hecs.

[20] Michele Caini. *EnTT*. Version 3.15.0. Mar. 19, 2025. URL: https://github.com/skypjack/entt.

[21] Michele Caini. *Dylan Ancel*. Version 0.8. Mar. 31, 2025. URL: https://github.com/leudz/shipyard.

[22] Alec Thomas. *EntityX - A fast, type-safe C++ Entity Component System*. Version 1.1.2. Apr. 25, 2025. URL: https://github.com/alecthomas/entityx.

[23] Amethyst Foundation. *Specs Parallel ECS*. Version 0.20.0. Sept. 25, 2023. URL: https://amethyst.github.io/specs/.

[24] Vladyslav Batyrenko. *Bevy egui*. Version 0.36.0. Sept. 13, 2025. URL: https://github.com/vladbat00/bevy_egui.