



**Politécnico
Castelo Branco**

Escola Superior
de Tecnologia

ECSMoS: An ECS-based Pedestrian Mobility Simulator

Rafael Souza Cotrim

Orientadores

Alexandre José Pereira Duro da Fonte

João Manuel Leitão Pires Caldeira

Dissertação apresentado à Escola Superior de Tecnologia do Instituto Politécnico de Castelo Branco para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Informática - Área de Especialização em Desenvolvimento de Software e Sistemas Interativos, realizada sob a orientação científica do [categoria profissional do orientador] Doutor Alexandre José Pereira Duro da Fonte e [categoria profissional do orientador] Doutor João Manuel Leitão Pires Caldeira, do Instituto Politécnico de Castelo Branco.

Data

Jury Composition

Presidente do júri

Grau académico, nome do presidente do júri

Vogais

Grau académico, nome do presidente do júri

Categoria profissional e o nome da Instituição

Grau académico, nome do presidente do júri

Categoria profissional e o nome da Instituição

Grau académico, nome do presidente do júri

Categoria profissional e o nome da Instituição

Resumo

Este resumo não deverá ter uma extensão superior a 500 palavras em português, inglês ou espanhol.

Palavras chave

Cinco palavras chave no máximo.

Abstract

Este resumo não deverá ter uma extensão superior a 500 palavras, por regra apresenta-se em inglês. Nos casos em que o documento seja escrito em língua inglesa ou espanhola, o abstract apresenta-se em português.

Keywords

Cinco palavras chave no máximo.

List of Abbreviations

CSV Comma-separated values.

EC Entity-Component.

ECS Entity Component Systems.

ECSMoS Entity Component 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.

General Index

1	Introduction	1
2	Current state of pedestrian modeling and simulation	2
2.1	Pedestrian models	2
2.2	Existing simulation frameworks	5
2.2.1	Survey criteria	5
2.2.2	Description of available simulation frameworks	6
2.2.3	Evaluation of available simulation frameworks	9
3	The Entity Component Systems architecture	9
3.1	General structure	9
3.2	Component storage strategies	10
3.3	Performance characteristics	12
3.4	Benefits and drawbacks	12
4	ECSMoS implementation	13
4.1	ECSMoS Structure	14
5	Evaluation	16
5.1	Performance Evaluation Results	18
5.2	Qualitative Evaluation Results	18
6	Conclusion	21

Figure Index

1	Diagram of repulsive forces between pedestrians i and j	4
2	Diagram of obstacle forces between pedestrian i and obstacle o	4
3	Menge architecture and simulation flow	7
4	Vadere architecture	7
5	Simplified MomenTUMv2 structure and flow	8
6	Example of how an ECS project is divided	10
7	Example of how an ECS can be seen as a database	11
8	Example of how archetype-based ECS stores components	11
9	External and internal parallelism comparison	13
10	Example of how ECSMoS handles inter-module dependencies	16
11	Example of how ECSMoS handles the addition of systems with dependencies	17

Table Index

1	Summary of available simulators	8
2	List and description of test scenarios.	19
3	Description of evaluation criteria and their levels.	20
4	Summary of qualitative results.	22

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 analysis of data collected from studying the movements of people in the real world [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 through a structure may aid during the design phase of a building, allowing for better planning 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 and increase the number of trampling incidents [SRC]. Finally, even when there is little risk of loss of life, they may still be useful for increasing the throughput of infrastructure such as train 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 in 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 in 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 panic [SRC].

When these models are implemented, it is often done on top of an existing 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 come 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.

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 scenarios 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 Component Systems (ECS) offer some benefits. ECS is a software architecture sometimes used in game development due to their much stricter performance requirements 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 in many scenarios, including the study of crowds. Despite this, ECS is still very uncommon in the scientific world.

With this in mind, this work proposes a new simulation framework for pedestrian dynamics: Entity Component 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 while maintaining a high performance. Chapter 2 provides an overview of existing pedestrian simulation models, how they operate, and what simulation frameworks are available. Chapter 3 talks about the details of the ECS architecture, its benefits, and its drawbacks. Chapter 4 goes over the details of how ECSMoS was implemented. Chapter 5 defines

the evaluation criteria used to validate ECSMoS and compares it to other simulators. Finally, Chapter 6 provides the final remarks of the work done and concludes this dissertation.

[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 that affect how one moves. Because of this, there are many competing models in use, each one with its own benefits and limitations. There are 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 discrete 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:

- 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 constraints of the model authors. Exotic requirements may lead to models being completely out of these constraints.

This dissertation 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 are applied upon them and are used to compute their acceleration, velocity, and ultimately position. Through 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 forces, 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 direction of its ultimate destination. If a lone agent were 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.

$$F_i^{\vec{drv}} = \frac{v_i^0 \vec{e}_i^0 - \vec{v}_i}{\tau} \quad (1)$$

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

\vec{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 their 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 guarantee 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 them, and the final force (see Equation 2) on i is the sum of all repulsive forces applied by other pedestrians (see Equation 3).

$$F_i^{\vec{rep}} = \sum_j \vec{f}_{ij} \quad (2)$$

$$\vec{f}_{ij} = [A_i e^{\frac{r_{ij} - d_{ij}}{B_i}} + k g(r_{ij} - d_{ij})] n_{ij} + \kappa g(r_{ij} - d_{ij}) \Delta v_{ji}^t \vec{t}_{ij} \quad (3)$$

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

$\Delta v_{ji}^t = (\vec{v}_j - \vec{v}_i) \cdot \vec{t}_{ij}$ 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 in 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

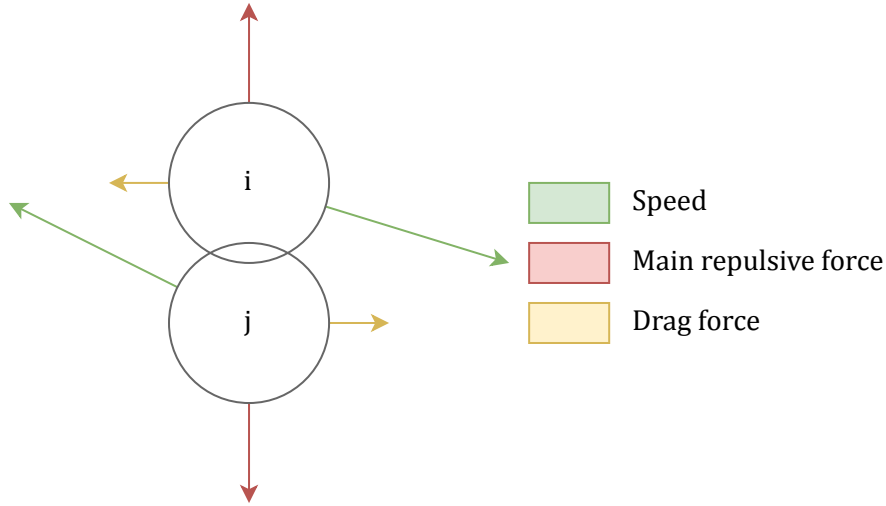


Figure 1 – Diagram of repulsive forces between pedestrians i and j

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

$$F_i^{\vec{obst}} = \sum_o \vec{f}_{io} \quad (4)$$

$$\vec{f}_{io} = [A_i e^{\frac{r_i - d_{io}}{B_i}} + kg(r_i - d_{io})] \vec{n}_{io} + \kappa g(r_i - d_{io})(\vec{v}_i \cdot \vec{t}_{io}) \vec{t}_{io} \quad (5)$$

where:

$F_i^{\vec{obst}}$ 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 the 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 closest point of o

\vec{t}_{io} is a unit vector perpendicular to \vec{n}_{io} rotated counterclockwise

A, B, k, κ are calibration constants

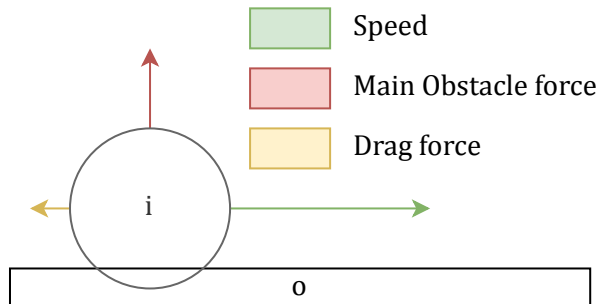


Figure 2 – Diagram of obstacle forces between pedestrian i and obstacle o

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.

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

$$pos_i^{\vec{new}} = pos_i + v_i^{\vec{new}} \cdot \Delta T \quad (7)$$

where:

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

\vec{v}_i is the previous speed of i

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

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

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

m_i is the mass of i

ΔT is the length of the simulation step

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

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 empirical studies. For example, it replicates the "Faster-Is-Slower Effect", which occurs when the flow rate of pedestrians through 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 PTV Viswalk [5], MassMotion [6], Legion Simulator [7] and AnyLogic [8]. Due to these limitations, scientific research tends to focus on more open simulators.

2.2.1 Survey criteria

To determine which simulators were most used and relevant to this work, a survey was performed. The following scientific article databases were searched: Google Scholar, IEEE Xplore, arXiv, SpringerLink, Science Direct and the Association for Computing Machinery Digital Library. For the search, the terms "pedestrian simulation", "pedestrian model", "pedestrian traffic", "pedestrian dynamics", "crow simulation", "crowd dynamics", "pedestrian motion", "crowd motion", "pedestrian movement", and "crowd movement" were used. The results were then filtered by date, removing anything prior to 2010. Finally, the top 50 results from each were evaluated.

After the initial search and evaluation, it was also specifically decided to add the articles from the journal *Collective Dynamics* to the search. This was done because of the simulators found during the initial search, many of them were originally published there, and the journal received a higher than average rate of citation on the other works. Despite this, its articles were not properly indexed by the major databases, possibly because of its small size, relative youth as a journal, and narrow focus on pedestrian and pedestrian and vehicular simulations.

Of the selected articles, most used custom-made software for the specific use case the researchers had in mind or did not reference a specific framework. While some of those in the former case made their code available, it was decided to not take these into account. This was done because such implementations are usually not meant to be extended by other developers. Therefore, this dissertation will only focus on simulation frameworks with the purpose of being used by other people than the authors themselves. Among these, only open-source ones with at least one update since 2015 were considered.

2.2.2 Description of available simulation frameworks

These were the simulation frameworks that passed all the filtering criteria:

Cromosim is a Python library for crowd simulations [9]. While it benefits from Python's large ecosystem of scientific tools, Cromosim has little pre-built infrastructure when it comes to helping researchers. 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 the Technical Research Center of Finland for the purposes of simulating pedestrian movement in building fire scenarios [10]. It is agent-based, uses the social forces model, and is mostly suited for evacuation scenarios in buildings with relatively flat floor plans. Buildings with a large degree of vertical movement within 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 [11]. 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) [12], 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 interactive 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 [13]. (1) Goal selection: Deciding what each pedestrian wants to achieve. (2) Plan computation: deciding on a strategy to achieve a pedestrian's objective. (3) Plan adaptation: Adapt the original plan to the current local environment of the pedestrian. (4) Spatial queries: How to retrieve data around a certain point and decide 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 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 in 2019.

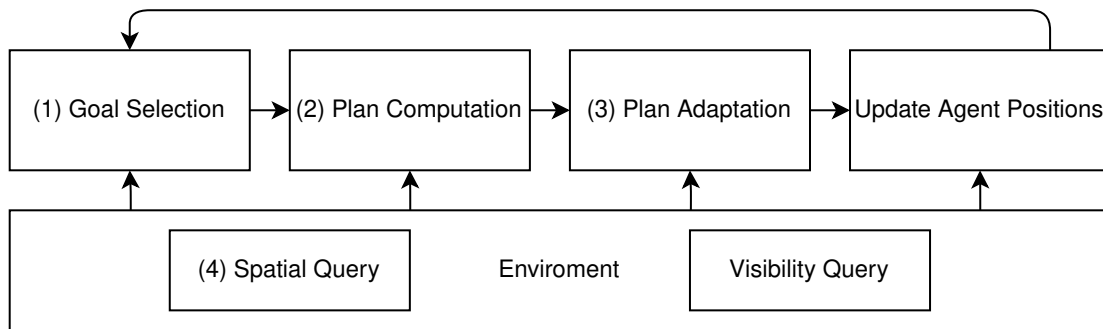


Figure 3 – Menge architecture and simulation flow. Adapted from [13].

Vadere is a situation framework for comparing different locomotion models developed by the Munich University of Applied Sciences [1]. It was designed to be lightweight while still remaining relatively flexible. It contains both a simulation engine that can be executed via command line or via its own UI. As shown in Figure 4, Vadere follows 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 [14], it imposes a performance drop on cellular automata models [1]. Attempting to use other approaches would require rewriting large portions of its internal workings.

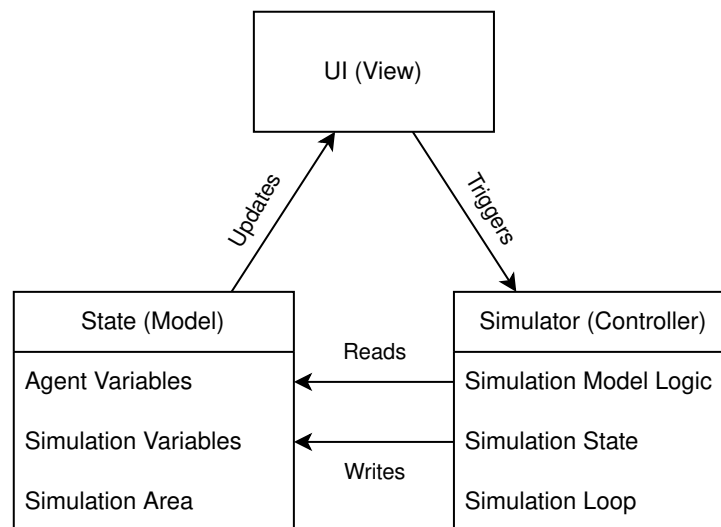


Figure 4 – Vadere architecture. Adapted from [14]

MomentUMv2 is a microscopic agent-based pedestrian simulator written in Java and developed by the Technical University of Munich [15]. 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, pedestrian generators/removers, analyzers, and much more. Models that do not fit into any predefined 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 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, and its last update was in 2020. Since then, 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. A simplified version of MomenTUMv2's architecture and execution flow can be seen in Figure 5.

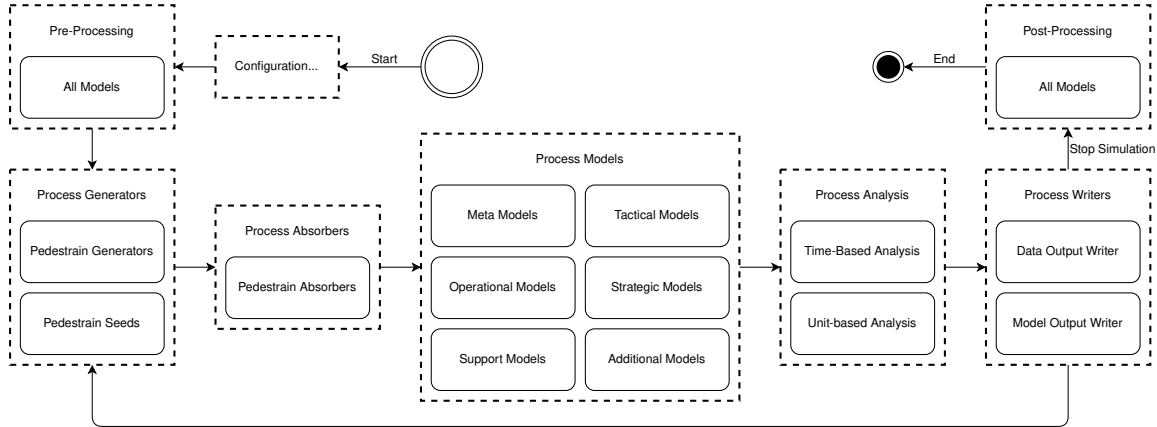


Figure 5 – Simplified MomenTUMv2 structure and flow. Adapted from [15]. Dashed rectangles represent organizational units. Solid rectangles represent a set of generic models and. Sets may contain many other families of sub-models within.

Simulation of Urban Mobility (SUMO) is a traffic simulator widely used in traffic engineering and related topics [16]. While it is mostly focused on vehicular movement, it does have some built-in mechanisms 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; they are restricted to certain lines/paths. For more robust simulation capabilities, it is possible to use JuPedSim together with SUMO. However, this integration is still a work in progress.

Finally, there are other libraries such as **Mesa** [17], **Agents.jl** [18], and many others, which focus on agent-based modeling. While these can be used for pedestrian 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 than other purpose-built simulators. Table 1 contains a summary of the considered frameworks and their characteristics. It is worth noting that projects with no clear status and no recent activity were classified as inactive. Only projects that have been officially terminated were classified as discontinued.

Name	Languages	Has UI	License	Last Update	Project Status
Cromosim	Python	No	GPL-3.0	2024	Supported
FDS+Evac	Fortran	Yes	Custom	2022	Discontinued
jCrowdSimulator	Java	No	LGPL-3.0	2024	Supported
JuPedSim	C++ and Python	No	LGPL-3.0	2025	Supported
Menge	C++	Yes	Apache-2.0	2019	Inactive
MomenTUMv2	Java	Yes	Custom	2020	Inactive
SUMO	C++ and Python	Yes	EPL-2.0	2025	Supported
Vadere	Java	Yes	LGPL-3.0	2025	Supported

Table 1 – Summary of available simulators

2.2.3 Evaluation of available simulation frameworks

Despite the variety of available tools, most of these are only compatible with a subset of models. Menge has a 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 within its architecture. Finally, libraries like Agents.jl are very capable of adapting themselves to many different kinds of models but lack any infrastructure for pedestrian simulations specifically.

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

3. The Entity Component Systems 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 difficult to find concrete statistics 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. Most works on this subject are blogs, videos and personal anecdotes. Because of these factors, this section will discuss the most common aspects of the EC architecture, saving details of the specific implementation used for ECSMoS discussed in Chapter 4. Nevertheless, this architecture has the potential of addressing many of the problems raised in Section 2.2, and merits further exploration.

3.1 General structure

Generally speaking, in an ECS implementation, most things are divided into the following three main kinds of concepts:

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 that 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. In this situation, the component works like a tag, marking the entity as possessing a certain property. For example, an entity that is a pedestrian may have an empty `Pedestrian` component, which serves only as a 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 predefined 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 define by itself when each system is executed. Once all systems have been 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.

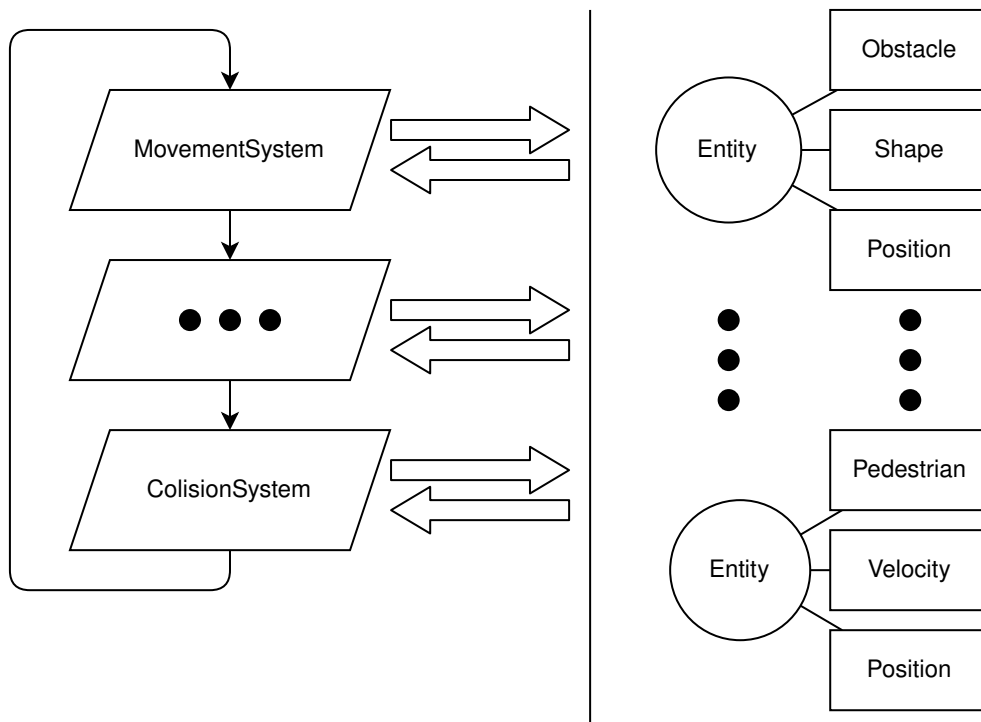


Figure 6 – Example of how an ECS project is divided. On the right, there are entities and their related components. On the left, systems are shown. Systems are executed according to their specified order, and read/write data to and from the components.

Figure 6 contains an example of how these three concepts interact. 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 a primary key. A system then would perform queries for the data in the 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 that 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 7.

3.2 Component storage strategies

While all ECS implementations have the same basic structure, one of the main differentiating factors between them 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,

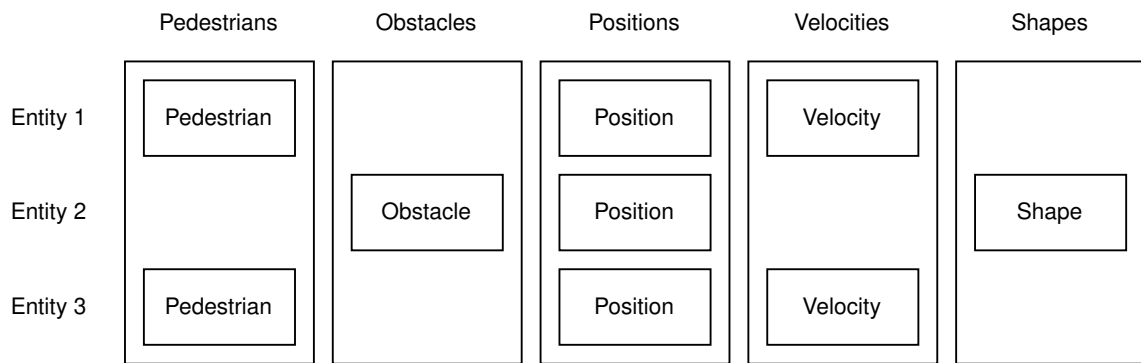


Figure 7 – Example of how an ECS can be seen as a database. Adapted from [19].

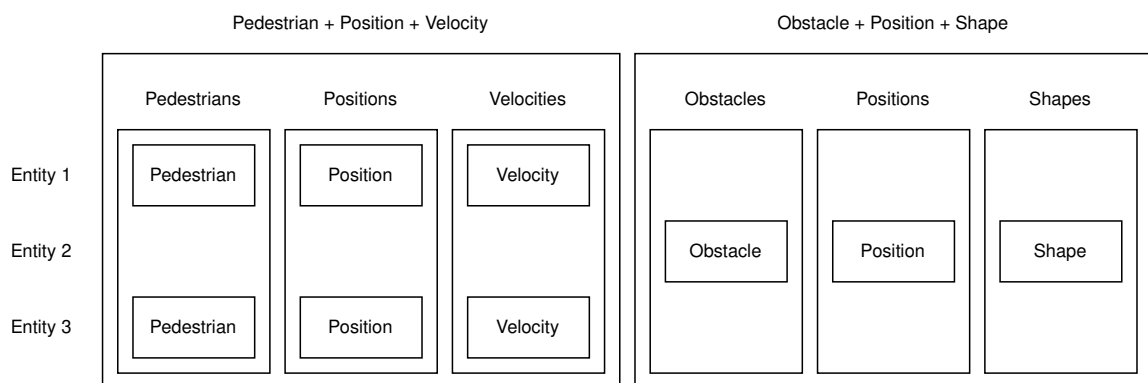


Figure 8 – Example of how archetype-based ECS stores components.

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 spatial locality. When a system requires data, the framework checks which 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 Flems [20], Unity DOTS [21], Bevy ECS [22], Legion [23] and Hecs [24]. For a visual example, see Figure 8.

Sparse set: In 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 [25] and Shipyard [26] are some of the most well-known frameworks that use sparse sets. This approach is the most similar to the one depicted in Figure 7.

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 [27] and Specs [28] 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 on 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 uses, 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 [22] and Specs [28].

3.3 Performance characteristics

Depending on the details of how components are stored, this architecture maximizes the efficiency of the CPU cache when compared to the architectures of other simulators, which 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 spatial locality. As a result, the CPU does not need to check the RAM as often, which 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, and 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. These two modes of parallelism can also work together. In this situation, systems can be executed in parallel and systems can split their data into different parallel processes. Figure 9 shows an example of these approaches.

While equivalent techniques could be employed on existing simulators, such changes would be hard to archive and prone to causing 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 parallelism 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 [22], allow for a seamless implementation of these concepts.

3.4 Benefits and drawbacks

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 elementary. 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 `Position` and `Speed` component, which causes the `SpeedSystem` to make it move according to its speed. If the pedestrian then decides to sit, the `Speed` component can be removed, which automatically removes it from the `SpeedSystem`'s list of entities to

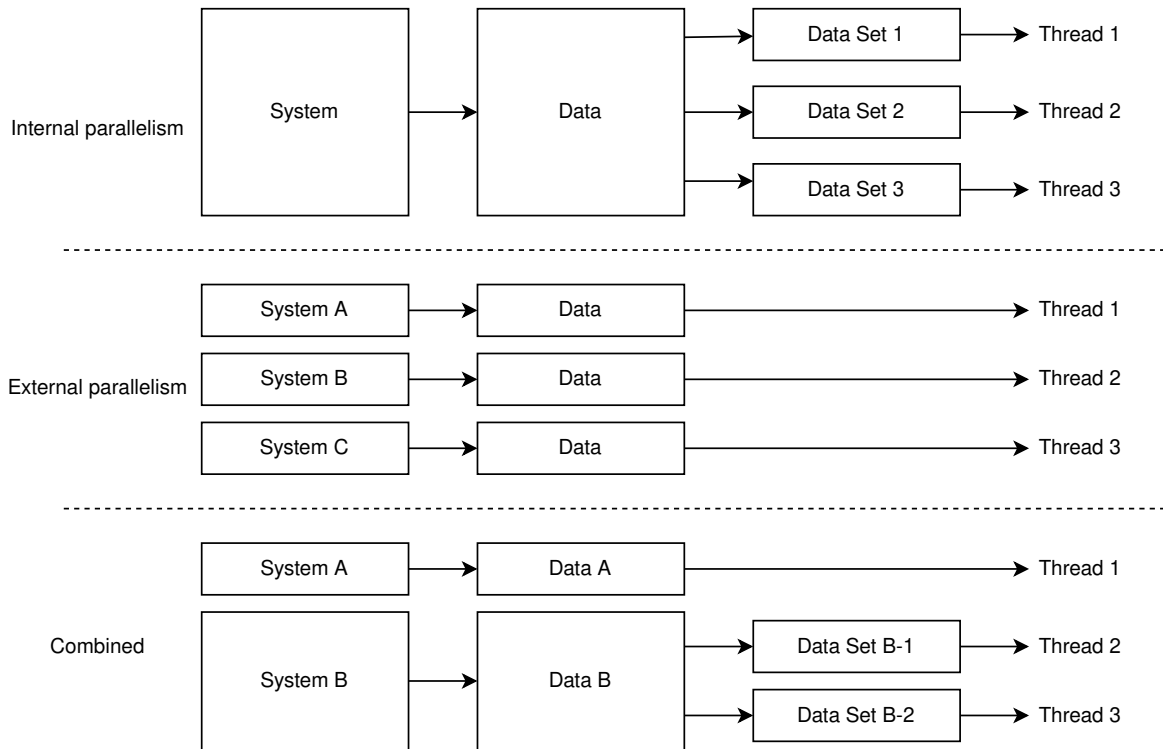


Figure 9 – Example of how internal and external parallelism work and how they operate when combined.

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 `Position` and `Speed` will automatically be affected by the `SpeedSystem`.

Despite these advantages, ECS also poses some problems that 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 paradigms with the design principles it is built upon. For example, while ECS can be implemented in OOP-focused languages, it breaks many of OOP's principles. 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 there is less information about it, which naturally perpetuates its status.

ECS also has some problems inherent to its structure. Due to the fact that component data is effectively 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 debugging, as the data for each entity is spread around multiple places and possibly using multiple forms of storage, making it difficult to have a holistic picture of what the system's state is.

4. ECSSMoS implementation

Considering the problem of other simulators referenced in Section 2.2, the main objective of ECSSMoS is to be more flexible than existing simulators. It should accommodate the addition of new models easily, should remove as many constraints for model authors as 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 or other areas was merely made for the purpose of supporting the goals of the project.

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 feature-rich than the ones already available. Among the available options, Bevy ECS, an 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 their execution order is, 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 method, 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 of 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.

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 at 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 are prevented from leaving. The information provided by this system are also useful for flow field pathfinding, which uses it to 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 10 shows an example of how inter-module dependencies and system sets operate. It is important to note 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 predefined system sets, and the arrows between the systems and system sets represent their dependencies. Systems without a dependency, such as `Compute Obstacle Collision Map` can start execution as soon as possible and, in this example, can also be executed in parallel. `Bevy` executes systems in parallel in no particular order, 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 shown, `Compute Motivation Force` depends on the `Flow Field Pathfinding` module's system set, meaning that it will only be executed after it has 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 system 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 11 shows an example of this. This new set can be anonymous and, as such, remain in use only inside the module or be given an 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.

As a result of this approach, ECSMoS does not have a linear execution/relationship path between its modules like other simulators. Instead, it can be best imagined as a network, where each system is a node with its own relationships. Systems are generally more connected to other systems within the same module, but they often connect to other ones. While this increases the general flexibility of the simulator, it also comes with a drawback, as it allows for developers to unknowingly create circular dependency paths.

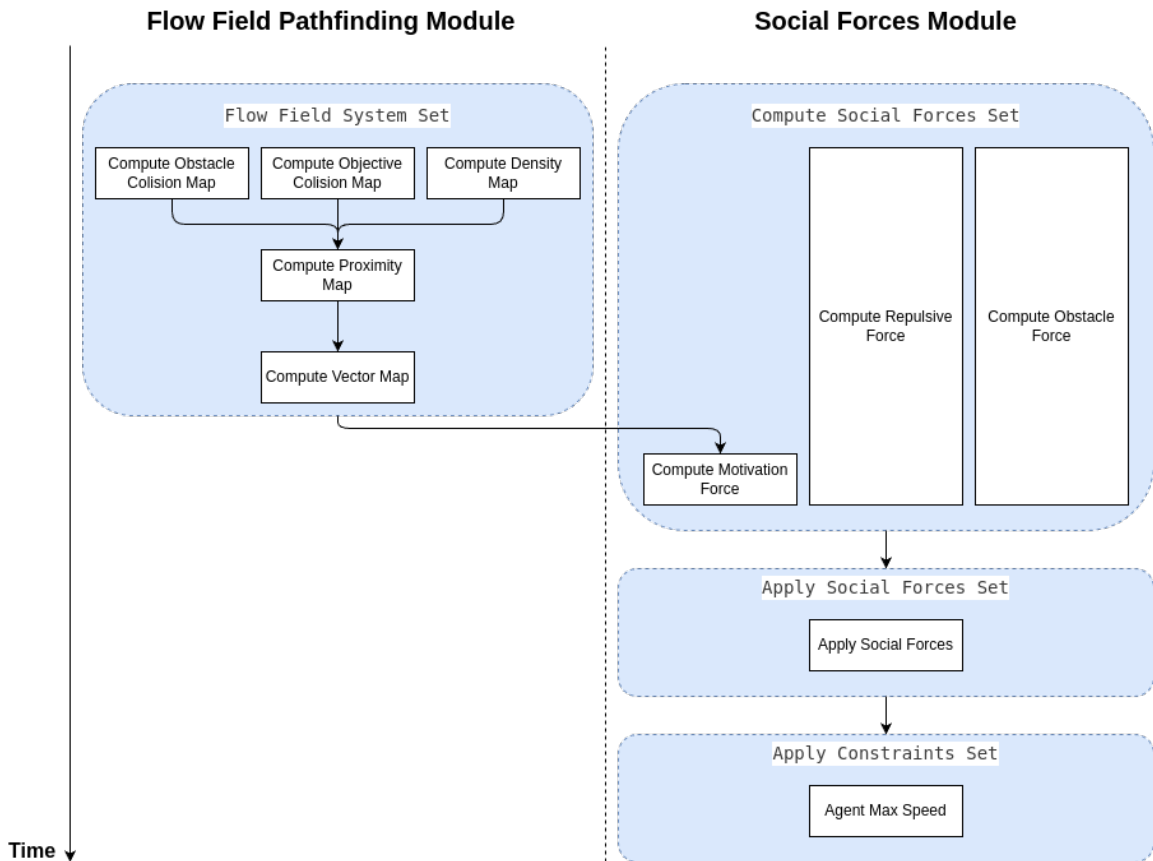


Figure 10 – Example of how ECSMoS handles inter-module dependencies

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 systems which is relatively restrictive, but it is possible to add other UI systems such as Bevy EGUI [29]. 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 a 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 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

For the purposes of this work, two kinds of evaluations were defined for the simulator: quantitative and qualitative. The quantitative evaluation consists of measuring its performance when compared to other simulators in similar test scenarios. The objective of this analysis is to check if the change to this new architecture somehow negatively impacted its performance or if it caused improvements.

In addition to ECSMoS, two other simulators were selected for the comparison: Vadere and Juped-sim. These were chosen because they are the most robust ones still being updated. Out of the remaining simulators, MomenTUMv2 would also have been a suitable candidate due to how flexible its architecture

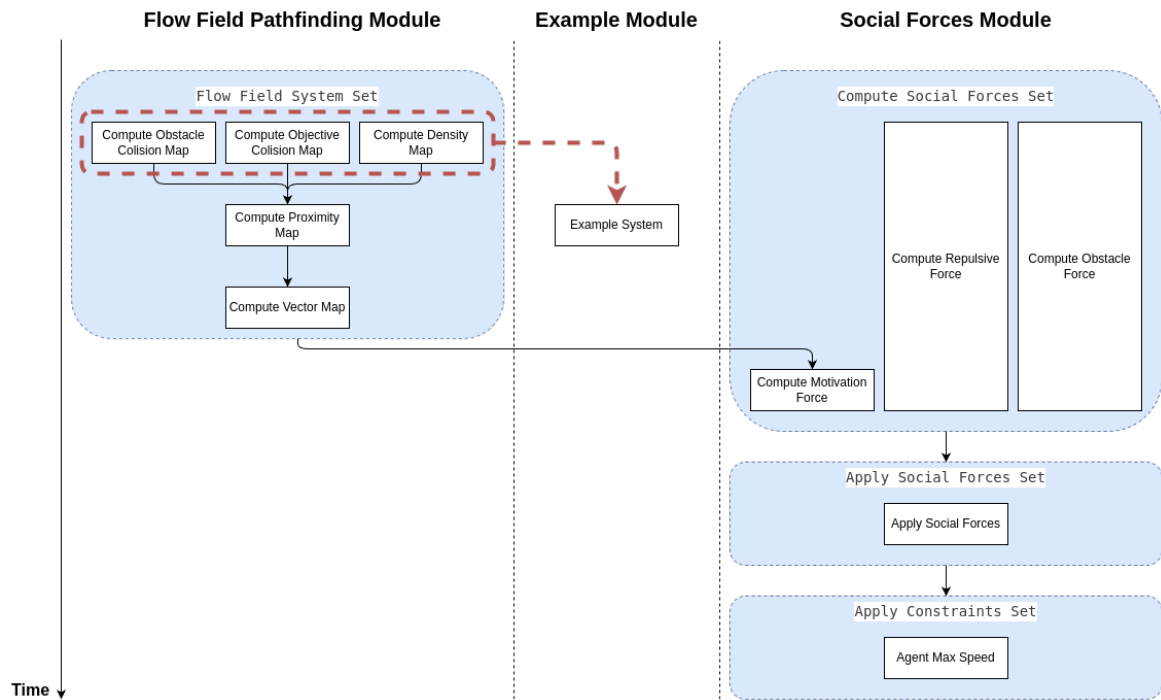


Figure 11 – Example of how ECSMoS handles inter-module dependencies where there is no pre-defined system set grouping only the required systems. The thick dashed and rounded square represents the unnamed set created.

is. However, the fact that its continued development is uncertain and that it was left in a non-functioning state with no clear indication of which previous version could be considered usable meant that it was decided to remove it from the comparison. Conversely, jCrowdSimulator receives regular updates, but was not considered for this comparison for lacking much of the basic infrastructure of a general simulator and due to its tightly coupled nature.

[Performance scenarios]

However, the qualitative evaluation is arguably more important. It focuses on comparing the capabilities of the simulators when it comes to the implementation of new models and adaptation of already existing ones. As such, it touches on the main objective of this project: flexibility. For this evaluation, a set of common scenarios seen during development of new models was proposed. Table 2 contains a list of the scenarios. These were designed to test specific aspects of each simulator. However, they also had to be somewhat flexible to avoid hitting specific weak points of the simulators that could be easily avoided by slight shifts in the definition criteria.

Scenario 1 contains an example of this flexibility. The simulators being evaluated do not all use the same basic pathfinding system. Since what pathfinding systems are available is not an object of study in this dissertation, it was decided to write the scenario in such a way to allow for the use of any one of these. Similar accommodations have been made in the other scenarios.

Scenarios 1 to 3 test basic situations that are likely to come up during the development of most novel models. If they are entirely novel, they would likely go with a path more similar to scenario 1. Models that are mostly based on another but have additional variables/factors would match scenario 2. Lastly, Scenario 3 covers additional implementations that may not cleanly fit into the previous categories.

Scenarios 4 and 6 take aim at features that are less commonly used but that may be hard barriers when it comes to implementation. Some models explicitly consider how an agent makes decisions in a changing environment [30], while others include actions that involve things other than movement, such

as waiting in line and loitering [31, 32].

A simulator's pathfinding algorithm may also have limitations or performance issues. These algorithms are used to compute what path the pedestrians would attempt to take if the other agents around them were static and, for the Social Forces Model, are used as a basis for the driving force. Vadere and ECSMoS use similar implementations that work well for the SFM but that generally have worse performance for cellular automata models [1]. As such, it may be necessary to change them under some set of circumstances. Scenario 5 tests that change.

After the scenario definitions, an attempt was made to implement them on each simulator. Implementations were then graded according to three metrics: implementability (meaning ease of implementation), reusability, and complexity. These grades were given on a three-level scale, going from poor to good. The details of each level are described in Table 3. In summary, a grade of "good" indicates that a simulator can easily handle the implementation of the scenario. Its architecture poses minimal challenges in regard to that criteria. Conversely, a grade of "poor" implies significant challenges, to the point that it may not be feasible to implement the scenario. Finally, a grade of "fair" indicates that there are some challenges, but these are possible to circumvent with a reasonable level of effort.

5.1 Performance Evaluation Results

5.2 Qualitative Evaluation Results

[This section needs a warm-up]

Each simulator had very similar results in scenarios 1, 2, and 3. These are not complicated setups and serve as a comparison basis for the other ones. However, they already reveal some trends that will persist over most of the other scenarios. Firstly, even in basic scenarios, ECSMoS is more complicated than desired. This is due to how open-ended its architecture is. Because new systems can be added to any particular point in the schedule, in practice means that it is always somewhat complicated to understand the control flow of the simulator. However, implementing model variations and hybrid models works well, as these can be added on top of existing models easily via the addition of a few systems that work together with the older ones.

Secondly, Jupedsim generally performs worse than the others being studied under these criteria. Such a pattern stems from its highly coupled internal structure. Even in basic scenarios, Jupedsim requires many changes all around the codebase in the form of string comparisons and others that define how execution will go depending on each model. This is most notable in the interface between Python and C++ code, where there are large sections of model-specific code that are executed depending on a sequence of chained conditions.

Finally, Vadere receives a downgrade in reusability scores starting on scenario 2 due to the fact that its pre-existing model implementations have a somewhat limited level of extensibility. With better choices when it comes to how the code is divided or how classes and interfaces are structured, it could cut down on the amount of code duplication significantly.

Scenario 4 introduces more challenges when it comes to flexibility. Both Vadere and Jupedsim suffer on this front, as they assume static environments, which is a reasonable expectation for some simpler situations, but that is not a universal property. Within Vadere, it is still possible to accommodate these with some work, but it will require reengineering some portions of the simulator itself. However, they

Nº	Name	Description	Objective
1	New Model	Add a new simple model that uses a basic pathfinding system already implemented on the simulator	Test reusability of basic simulator infrastructure
2	Model variation	Create a variation of the Social Forces Model available in the simulator with an additional attractive force that guides agents to areas where pedestrians would natural concentrate as defined in [3]	Test reusability of existing model implementations when the variation is small
3	Hybrid model	Create a model based on an already existing one that changes how it operates under some set of circumstances, similar to models that takes panic into account like those in [33]	Test general adaptability of already existing models to larger changes
4	Procedural environment changes	Create a simulation environment that changes over time, adding new obstacles that change the pedestrian's paths as seen on [30]	Determine whether the simulator is able to handle updates to data that is not related to the pedestrians directly
5	Implementing a new pathfinding algorithm	Add a new algorithm for determining the shortest route between a pedestrian's current location and their destination	Check flexibility of current models to changes in base infrastructure, which may be necessary for performance or other limitations [1]
6	Queuing and loitering	Add procedures for pedestrians forming queues and for pedestrians staying still while waiting for something, but with some level of reactivity to the surrounding environment	Queuing and loitering are behaviors that significantly change how a pedestrian moves and interacts with its environment. They also break many assumptions that are usually made by models, which often require continuous movement [31]

Table 2 – List and description of test scenarios.

Criteria	Grade	Description
Implementability	Good	The simulator can easily accommodate the required additions. It has a clear place and method for receiving the new model, requiring minimal changes to existing code.
Implementability	Fair	Implementation is feasible but requires moderate exploration of the code to identify where changes are needed or there are some tight coupling/rigid designs that slow down development.
Implementability	Poor	Simulator lacks extension points for this scenario, requires deep internal changes or requires modification to parts of its code that would be very difficult to decouple. Implementation is slow, messy or may not be feasible.
Reusability	Good	Existing code can easily be reused in this scenario. The related models, utility methods, etc. are generic, have good abstractions and are composed of easily swappable units.
Reusability	Fair	Some reuse is possible, but components are partially coupled to specific behaviors or scenarios. Reusing core infrastructure may impose minor restrictions or require small modifications. Model-specific logic may need to duplicate existing features if interfaces are too narrow or rigid.
Reusability	Poor	Simulator components are hard-coded or specialized for specific agents or scenarios. Implementing the scenario requires rewriting or duplicating most, if not all, of a model or internal system to operate.
Complexity	Good	Implementing the scenario is possible with minimal cognitive effort. The changes are well isolated, have logical continuity, touch as few other systems and the data/execution path they follow is direct and predictable.
Complexity	Fair	Implementing the scenario requires working with several components whose relationships aren't always intuitive. Changes may need to be performed in many regions of the code. Understanding the control flow is unintuitive or time-consuming, but possible.
Complexity	Poor	Fully integrating the scenario is very complicated. Control flow is chaotic, multiple tightly coupled parts have to be modified, or they may require a large redesign of the architecture of the simulator itself.

Table 3 – Description of evaluation criteria and their levels.

are basically unimplementable in Jupedsim, as it would require reengineering most of the simulator. In comparison, ECSMoS handles such things much better. Bevy ECS, provides methods for detecting this kind of changes, and, in many situations, these can be added with little to no modification to the code of the systems themselves. In the situation where changes to the systems are necessary, those are limited and much more predictable.

Scenario 5 causes a reversal in the grades between Vadere and Jupedsim. Vadere's pathfinding algorithm, referred to as floor fields, is much more integrated with the simulator architecture and existing models than Jupedsim's, which uses triangulation to determine the best possible path for each pedestrian. While it is possible to replace floor fields with some other mechanism, it would involve changing much of the simulator code and all current implementations of models, making it impractical. In ECSMoS, such a thing is perfectly possible. Reuse of code between the current pathfinding system and a new one may be limited, but the architecture will not impose additional challenges, and the changes necessary to allow already existing models to use it are small and concentrated in singular places.

Finally, in scenario 6, Jupedsim performs very well. Its architecture actively expects such behavior, so it is well integrated and quite flexible to changes. While ECSMoS was not designed with this in mind, this can be accomplished similarly to how hybrid models were implemented in scenario 3. Queuing and loitering work as an extension on top of the already available model. Vadere's implementation in theory could be done in the same way as ECSMoS's, but in practice it would require rewriting the whole model to take this into account due to some limits to its extensibility. Therefore, it gets a downgrade in the reusability criteria.

A summary of the results can be seen in Table 4. As the results show, Vadere and ECSMoS are the best for basic scenarios, with Vadere having a slight edge. Each one has its strong and weak points, but they are generally good choices for implementing a novel pedestrian model that does not take into account more complicated things such as changes to the environment. However, once more involved changes are necessary, ECSMoS pulls ahead because of its flexibility. In contrast, Jupedsim has generally worse results than these two, with its only standout result on scenario 6.

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).

Scenario \ Simulator	ECSMoS			Vadere			Jupedsim		
	Implementability	Reusability	Complexity	Implementability	Reusability	Complexity	Implementability	Reusability	Complexity
(1) New Model	Good	Good	Fair	Good	Good	Good	Fair	Fair	Fair
(2) Model variation	Good	Good	Fair	Good	Fair	Good	Fair	Fair	Fair
(3) Hybrid model	Good	Good	Fair	Good	Fair	Good	Fair	Fair	Fair
(4) Procedural environment changes	Good	Good	Fair	Fair	Fair	Poor	Poor	Poor	Poor
(5) Implementing new pathfinding algorithm	Good	Fair	Fair	Poor	Fair	Poor	Fair	Fair	Fair
(6) Queuing and loitering	Good	Good	Fair	Good	Poor	Good	Good	Good	Good

Table 4 – Summary of qualitative results.

- [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] *Pedestrian simulation software | PTV Viswalk | PTV Group*. <https://www.ptvgroup.com/en/products/pedestrian-simulation-software-ptv-viswalk>. Accessed: 2025-09-27.
- [6] *Oasys | MassMotion*. <https://www.oasys-software.com/products/massmotion/>. Accessed: 2025-09-27.
- [7] *LEGION | Bentley Systems | Infrastructure Engineering Software Company*. <https://br.bentley.com/software/legion/>. Accessed: 2025-09-27.
- [8] *AnyLogic: Simulation Modeling Software Tools & Solutions*. <https://www.anylogic.com/>. Accessed: 2025-09-27.
- [9] Sylvain Faure. *Cromosim: Crowd Motion Simulation*. Version 2.1.0. Dec. 30, 2024. URL: <https://github.com/sylvain-faure/cromosim>.
- [10] 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.
- [11] Axel Meinert et al. “Simulation von Menschenmengen Im Urbanen Umfeld Mit OpenStreetMap-Daten”. In: *gis. Science* (2019).
- [12] Armel Ulrich Kemloh Wagoum et al. “JuPedSim: An Open Framework for Simulating and Analyzing the Dynamics of Pedestrians”. In: Dec. 2015.

- [13] 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).
- [14] 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).
- [15] Peter M. Kielar, Daniel H. Biedermann, and André Borrmann. “MomenTUMv2: A Modular, Extensible, and Generic Agent-Based Pedestrian Behavior Simulation Framework”. In: (Jan. 2016).
- [16] 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/>.
- [17] 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).
- [18] 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>.
- [19] *Specs and Legion, two very different approaches to ECS*. <https://csherratt.github.io/blog/posts/specs-and-legion/>. Accessed: 2025-09-27.
- [20] SanderMertens. *Flecs*. Version 4.0.5. Mar. 18, 2025. URL: <https://www.flecs.dev/flecs/>.
- [21] Unity Technologies. *DOTS - Unity’s Data-Oriented Technology Stack*. Version 6.1. June 10, 2025. URL: <https://unity.com/dots>.
- [22] Bevy Engine. *Bevy*. Version 0.16.0. June 24, 2025. URL: <https://bevyengine.org/>.
- [23] Amethyst Foundation. *Legion*. Version 0.4.0. Feb. 25, 2021. URL: <https://github.com/amethyst/legion>.
- [24] Benjamin Saunders. *hecs*. Version 0.10.5. May 5, 2024. URL: <https://github.com/Ralith/hecs>.
- [25] Michele Caini. *EntT*. Version 3.15.0. Mar. 19, 2025. URL: <https://github.com/skypjack/entt>.
- [26] Michele Caini. *Dylan Ancel*. Version 0.8. Mar. 31, 2025. URL: <https://github.com/leudz/shipyard>.
- [27] Alec Thomas. *EntityX - A fast, type-safe C++ Entity Component System*. Version 1.1.2. Apr. 25, 2025. URL: <https://github.com/alecthomas/entityx>.
- [28] Amethyst Foundation. *Specs Parallel ECS*. Version 0.20.0. Sept. 25, 2023. URL: <https://amethyst.github.io/specs/>.

- [29] Vladyslav Batyrenko. *Bevy egui*. Version 0.36.0. Sept. 13, 2025. URL: https://github.com/vladbat00/bevy_egui.
- [30] Melissa De Iuliis et al. "Large scale simulation of pedestrian seismic evacuation including panic behavior". In: *Sustainable Cities and Society* 94 (2023), p. 104527. ISSN: 2210-6707. DOI: <https://doi.org/10.1016/j.scs.2023.104527>. URL: <https://www.sciencedirect.com/science/article/pii/S2210670723001385>.
- [31] Inhi Kim, Ronald Galiza, and Luis Ferreira. "Modeling pedestrian queuing using micro-simulation". In: *Transportation Research Part A: Policy and Practice* 49 (2013), pp. 232–240. ISSN: 0965-8564. DOI: <https://doi.org/10.1016/j.tra.2013.01.018>. URL: <https://www.sciencedirect.com/science/article/pii/S0965856413000256>.
- [32] MD Jahedul Alam, Muhammad Ahsanul Habib, and Devin Holmes. "Pedestrian movement simulation for an airport considering social distancing strategy". In: *Transportation Research Interdisciplinary Perspectives* 13 (2022), p. 100527. ISSN: 2590-1982. DOI: <https://doi.org/10.1016/j.trip.2021.100527>. URL: <https://www.sciencedirect.com/science/article/pii/S2590198221002323>.
- [33] Mohammed Alrashed and Jeff Shamma. "Agent Based Modelling and Simulation of Pedestrian Crowds in Panic Situations". In: *Collective Dynamics* 5 (Aug. 2020), pp. 463–466. DOI: 10.17815/CD.2020.100. URL: <https://collective-dynamics.eu/index.php/cod/article/view/A100>.