Self-Assembly and Synchronization: Crafting Music with Multi-Agent Embodied Oscillators

Pedro Lucas*, Alexander Szorkovszky†, Stefano Fasciani‡, and Kyrre Glette§
*†§ *RITMO, Department of Informatics*
‡ *Department of Musicology*
University of Oslo, Norway
{pedroplu, alexansz, stefano.fasciani, kyrrehg}@uio.no

***Abstract*—This paper proposes a self-assembly algorithm that generates rhythmic music. It uses multiple pulsed oscillators embedded in cube-shaped agents in a virtual 3D space. When these units connect with each other, their oscillators synchronize, triggering regular sound events that produce musical notes whose sound dynamics change based on the size of the structures formed. This study examines the synchronization time of these oscillators and the emergent properties of the structures formed during the algorithm's execution. Moreover, the resulting sound, determined by multiple interactions among agents, is analyzed in the time and frequency domains from its signal. The results show that the synchronization time slightly increases when more agents participate, although with high variability. Also, a quasi-regular pattern of increase and decrease in the number of structures over time is observed. Additionally, the signal analysis illustrates the effect of the self-assembly strategy in terms of rhythmical patterns and sound energy over time. We discuss these results and the potential applications of this multi-agent approach in the sound and music field.**

***Index Terms*—Self-assembly, Synchronization, Multi-Agent Systems, Swarm Intelligence, Oscillators, Generative Music**

I. Introduction

Self-assembly is a natural growth mechanism that occurs on different scales. It involves components that follow local interaction rules, leading to the formation of patterns or structures without human intervention [1] [2].

This mechanism of autonomous organization has been investigated in various fields, beginning with the study of molecules and advancing to artificial applications like robotic systems [2]. This reflects a scientific interest in exploring this concept, which is considered fascinating due to the appearance of order from disorder [1]. In that sense, we hypothesise that the self-assembly mechanism can produce dynamic and emergent behaviour that can be used for generative music. As such, the reason why a self-assembly approach is suitable for this task is due to how music can be composed in sequential or simultaneous arrangements (i.e. melodies and chords) over time through the "assembly" of several simple units (i.e. notes and duration) in a coherent way. This reinforces the idea of transitioning *from chaos to order* for music composition, which is a process that requires multiple perspectives that have to do with essential components of music, such as *harmony*, *melody*, and *rhythm*.

Our goal is to explore *self-assembly* for music generation, considering the *rhythm* component as a foundation for including harmonic and melodic elements; as such, we propose a self-assembly strategy for agents to execute regular events in time and interact in a virtual environment. These agents have internal pulsed oscillators to trigger these events; and, as they need to be represented as shapes that can participate in the self-assembly process, we refer to them as "embodied oscillators". The synchronization of these oscillators is achieved when agents join each other.

The events triggered by the agents are musical notes assigned randomly from pre-defined musical arrangements, such as musical scales. The sound dynamics of these notes change when the agents are part of bigger structures, which leads to an emergent formation of harmonies and melodies.

In this paper, we describe in detail this novel self-assembly algorithm and present a study regarding synchronization aspects of this solution, as well as emergent properties related to structure formation and the resultant sound output. To perform this study, we implemented the algorithm in a virtual 3D environment using the *Unity3D*[1] game engine and produce a video of a particular system configuration[2]. The objective of this study is to conduct an initial exploration of the overall system supporting the proposed music-making process. This exploration is relevant for understanding the complexity that leads to the sonic result.

The contributions of this work are: a novel self-assembly algorithm operating in a virtual environment, a strategy for synchronization and sound mapping for music generation using this algorithm, the study mentioned above, and a public GitHub repository[3] that contains all the necessary assets to replicate this work. These assets include the Unity project and Python scripts used for data analysis.

This paper is organized as follows: Section II lists works related to this research; Section III describes the self-assembly algorithm, starting with the rationale behind this strategy, the anatomy of the agent, the description of the pulsed oscillator used in this work, and sound and music aspects; Section IV shows the results for the performed study; Section V discusses our findings and future work regarding this proposal; and finally Section VI presents conclusions.

[1]https://unity.com/
[2]https://youtu.be/GplsQD09y7Q
[3]https://github.com/pedro-lucas-bravo/self_assembly_sync_music

II. Related Work

The scientific literature contains numerous works related to the term "self-assembly", particularly in the fields of chemistry and biology. This indicates that self-assembly is a widespread process in nature [2], suggesting that the most important justification for studying self-assembly is its central role in life [1]. However, our focus lies in exploring artificial solutions that draw inspiration from swarm intelligence mechanisms, specifically in the field of swarm robotics. Self-assembly solutions in this field attempt to solve various problems related to hardware and software. For instance, Nagy *et al.* [3] explored a magnetic solution for swallowable modular robots to navigate the gastrointestinal tract, which proved to be highly adaptable to irregular paths. In terms of algorithms, the recent work of Zhang *et al.* [4] describes the creation of docking mechanism layouts for assembling autonomous vehicles, proposing a reliable and efficient *Self-Assembly Planning (SAP)* algorithm and referring relevant research in this area. Additionally to a self-assembly algorithm, the quality of the structures it forms is important. The common approach to achieve this involves proposing an algorithm and performing heuristic evaluations of the structures, typically using evolutionary algorithms [5]. Several works in robotics have explored the effectiveness of self-assembly algorithmic solutions, including *s-bots* [6] [7], as well as the evolutionary approach [8].

In the field of music, there are some prior examples of the use of self-assembly. One of these is the installation *FATHOM: Self-Assembly Music*[4] that artistically associates sounds and their properties with predefined geometric shapes. When visitors interact with these shapes through a physical interface in a virtual space, it creates new musical material. Self-assembly has also been used for music analysis in the scientific literature. Abstract spaces representing well-known musical objects (e.g. sequence of chords, interval series, etc.) have been built using self-assembly in [9] and [10]. These spaces can be valuable in the field of musicology for analysis and also have the potential for music composition, with the latter being the focus of this paper.

We did not find any additional references to self-assembly related to music, suggesting that our work is one of the first to explore the self-assembly concept in this field from a more formal and scientific perspective.

Furthermore, we incorporate rhythm as a foundational element, taking music synchronization into account. As such, we refer to studies from Nymoen *et al.* [11] [12], which utilize pulse-coupled oscillators for *self-synchronization*, a process that aligns phases based on an update rule whenever an oscillator detects pulses from others. In our work, we use this type of oscillator, but in our case, synchronization is performed through our self-assembly method. This integrated approach offers a novel strategy for addressing music composition.

[4]https://shanemyrbeck.com/portfolio/fathom-self-assembling-music-2/

III. Self-Assembly Strategy

In this section, we describe the rationale for this strategy related to music, the agent anatomy, the self-assembly algorithm, and the music-making mechanism used to achieve the collective behaviour showcased in a simulation presented in an audio-visual recording[2]. Fig. 1 shows a real-time session where 50 agents interact with each other forming structures within a spherical environment.

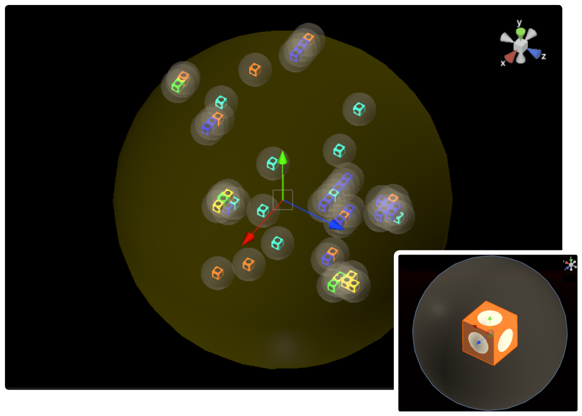


Fig. 1. A group of 50 agents interacting within a yellow semitransparent sphere with radius R_E. Some are alone, while others form structures. The colour of each individual represents its current state. At the bottom right, we have the graphical representation of an agent A_i. The faces are the `slots` where other agents can join. The convex circular sides allow us to visualize its internal oscillator's pulse.

A. Rationale

In principle, we establish a mechanism for assembling and disassembling agents that allow structures of different sizes to be produced over time. This mechanism can control these sizes according to the variation of local parameters and local interactions. The agents have an internal pulse oscillator that is synchronized every time agents join each other. The criteria for choosing the sound mapping defined in Section III-D take this self-assembly mechanism into consideration.

We justify the design of the self-assembly algorithm in combination with pulsed oscillators based on its support for rich rhythmic music. This music includes melodic and harmonic elements as a result of the agents' dynamics, with components that have been analyzed as musical objects in previous works. These works are mentioned in Section II in [9] and [10] using self-assembly; however, our focus is creating music. Particularly, applying a sound mapping such as the one described in section III-D gives us rhythms through the playback of musical notes using the oscillators when agents are part of a structure. Melodies can be identified for small structures, while harmonies are created through the overlapping of notes, which are held longer for larger structures due to the mapping with the amplitude envelope applied to the notes. The combination of these events generates music emergently, which varies as agents assemble and disassemble over time.

The set of possible design choices to achieve the goals presented above is not limited to the proposal described in this work; other possibilities can be considered.

B. Agent Anatomy

From a set of self-assembly agents $\mathcal{AG} = \{A_1, A_2, A_3, ...\}$, we represent an agent A_i as a virtual cube in 3D space, as shown in the bottom right of Fig. 1. Each face is a `slot` that can host another agent, allowing one agent to have up to six more agents "assembled" to it, one per face. When agents are connected together, they can exchange data directly through these slots. The set of slots for A_i is denoted by $\mathfrak{F}_i$, where $\mathfrak{F}_i \subseteq (\mathcal{AG} \cup \{\emptyset\}) - \{A_i\}$ and $|\mathfrak{F}_i| = 6$; here, $\emptyset$ represents the `null` agent (empty slot).

A_i has a communication radius R_C to detect other agents in a nearby region. This area is depicted in the bottom right of Fig. 1 as a semitransparent sphere surrounding the cube. When another agent enters this region, the algorithm for self-assembly described in the next Section III-C takes place.

Additionally, A_i includes an internal oscillator to generate regular pulses which trigger sound events for music-making as described in Section III-D. This pulsed oscillator is represented by its phase ϕ_i and has been previously used in studies related to music synchronization [11] [12]. In these studies, the update rule for a pulsed oscillator i is described as follows: The phase is initialised randomly between 0 an 1, and evolves over time (t) toward 1 at a rate of $\omega_i(t) = \frac{d\phi_i}{dt}$, this rate is the *frequency* of the oscillator. When the phase of oscillator i reaches maximum, the node "fires", and resets back to 0 before it continues to evolve toward 1.

The oscillator behaviour can be visualized as a colour pulse lighted over a convex circle on each side of the cubic agent, as shown in the bottom right of Fig. 1. The development of this oscillator in time is illustrated in Fig. 2 together with a representation of the visual feedback provided by the virtual agent.

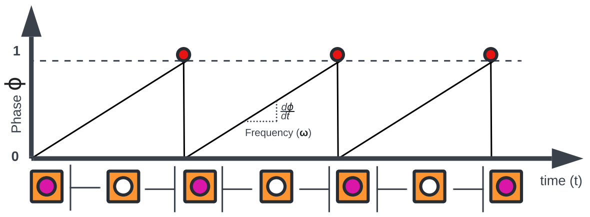


Fig. 2. Internal pulsed oscillator ϕ_i. A 2D representation of the agent under the x axis shows that the sides change colour (from white to purple) briefly to indicate the pulse that triggers a sound event.

C. Agent Behaviour: Self-Assembly Algorithm

An overall description of the agent's behaviour in a 3D space is as follows: An agent A_i enters a spherical environment E of radius R_E and moves in space in a straight line with a constant speed v. Initially, A_i adopts a random position and direction in this environment; however, when it faces the boundaries of E, it changes to a random reflected direction in the hemisphere defined by the tangent plane of E at the collision point to continue its constant movement through a different path. Each agent has a communication range of radius R_C that allows it to detect other agents within that range; thus, when A_i detects another agent A_j, they exchange data to ensure that both can join together and form a structure; that is, both must have a `slot` (cube face) available to perform the `join` operation; otherwise, both agents continue their normal paths. Moreover, the agent A_i may be able to detach from a structure considering the following mechanism: A_i is running an internal timer t_{join_i} which is restarted every time a successful `join` operation takes place; nevertheless, while this timer is running, A_i will not detach from its current structure to join a detected agent A_j, which means that two structures (i.e., the structure that contains A_i and the one that contains A_j, if that is the case) can join through this mechanism to form a bigger one; however, if the time t_{join_i} reaches a value T_{jmax}, then A_i will be able to detach from its current structure upon detecting A_j and then joining alone to A_j; in that moment, A_i is selected as the leader responsible for directing the movement of the newly formed structure. This process allows the creation and dissolution of emergent structures of different sizes over time, as the example shown in Fig. 3, which helps with the music-making dynamics explained in Section III-D.

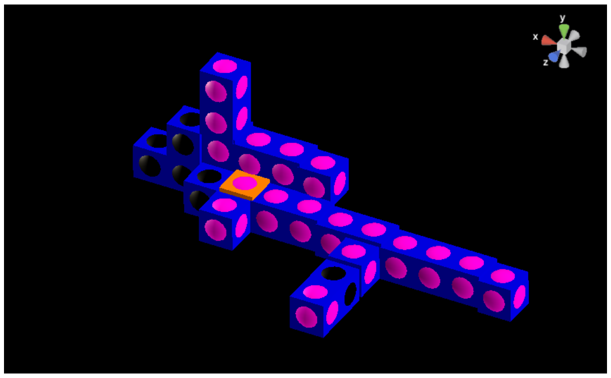


Fig. 3. Example of a structure involving several agents built through the self-assembly algorithm. At the moment of this capture, most of the agents were triggering a sound event (purple pulse), and others were off. The orange agent leads the movement of the whole structure while the others follow it directly or indirectly through a chain of connections.

The self-assembly algorithm for an agent A_i is described by the *Finite State Machine (FSM)* shown in Fig. 5. The FSM outlines the processes that occur in each state as sequential actions in the format of `do / Action1(), Action2(),...`. Additionally, the state transitions are shown as a pair of **`event`**`/actions`, where **`event`** is the trigger for the transition and `actions` is the set of sequential actions executed when the transition occurs, in the format of `Action1()`, `Action2(),...`. Moreover, an agent reflects visually its state through the colours shown in Fig. 4.

The specific procedure considering the FSM is explained below:

1) The environment E is instantiated with a radius R_E in the origin $(0, 0, 0)$ of the virtual 3D space. Then, we instantiate

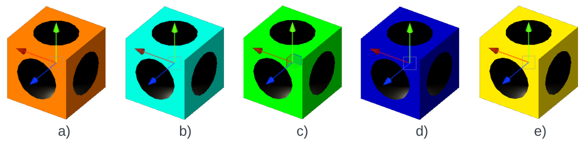


Fig. 4. Colours corresponding to every state for an agent A_1, which are: a) `Wandering`, b) `Wandering and Detecting`, c) `To Join` A_j, d) `Joined`, d) `Indirect to Join`.

TABLE I
CONFIGURABLE PARAMETERS SHARED BY ALL AGENTS IN A SELF-ASSEMBLY SESSION

Parameter	Description
$R_E \in \mathbb{R}$	Spherical environment radius
$\lambda \in \mathbb{R}$	Length of the cube side.
$v \in \mathbb{R}$	Movement speed
$T_{wmax} \in \mathbb{R}$	Maximum `wander` time
$T_{jmax} \in \mathbb{R}$	Maximum `join` time
$R_c \in \mathbb{R}$	Communication and detection radius

N agents which share the same values for the *configurable parameters* listed in Table I. To describe the algorithm, we will refer to agent A_i from the FSM in Fig. 5, where i could be any agent where $i \in \mathbb{N}$ and $1 \leq i \leq N$. A_i has a position $\vec{P_i}$ in space and moves with a constant speed v towards a direction $\hat{d_i}$. This $\vec{P_i}$ and $\hat{d_i}$ are part of the internal variables managed independently in each agent. These variables are presented in Table II. Additionally, A_i encapsulates abstract objects to refer to their connected agents, which are shown in Table III and will be explained accordingly as we continue the description of this algorithm.

2) After A_i is instantiated, the `Initialize()` action is run. The initialization process assigns a random position $\vec{P_i}$ within the boundaries of E (inside the spherical limits) and a random direction $\hat{d_i}$. Moreover, the timers t_{w_i} and t_{join_i} (through the `Restart_`t_{join_i}`()` action), explained later, are set to zero. In parallel, the internal oscillator for A_i is initialized through `Initialize_`ϕ_i`()`, which assigns randomly a starting phase ϕ. The oscillator mechanism is later explained in Section III-D.

3) Following the initialization, A_i enters to the `Wandering` state in which the position $\vec{P_i}$ is updated towards the direction $\hat{d_i}$ with a speed v in a discrete time step δt as defined by (1).

$$\vec{P_i}(t+\delta t) = \vec{P_i}(t) + v\delta t\hat{d_i} \tag{1}$$

Within the `Wandering` state, the timer t_{w_i} is being updated by accumulating the current time step δt as $t_{w_i}(t+\delta t) = t_{w_i}(t) + \delta t$. This timer is reset ($t_{w_i} = 0$) when A_i faces the boundaries of E; that is, when the event **`Boundary_Detected`** is triggered. The value of this event is determined based on the magnitude of the vector $\vec{P_i}$. Specifically, **`Boundary_Detected`** is set to `True` if $\|\vec{P_i}\| \geq R_E$, and `False` otherwise; assuming that E is placed in the origin of the virtual 3D space.
Additionally, when the boundary of E is detected, A_i moves away from the edge by changing its current direction $\hat{d_i}$ to a new direction that allows a "bouncing effect" from the edge. For this new direction, we consider a random point on the surface of a sphere with radius 1 as $\hat{h}$; then, as E is placed in the centre, the direction of the collision between A_i and the edge as $\hat{n}_i = \frac{\vec{P_i}}{\|\vec{P_i}\|}$, which is the normal vector to the collision point; next, we calculate the angle θ_i between $\hat{h}$ and $\hat{n}_i$; thus, we can define the new direction $\hat{d_i}$ in the next time step δt as $\hat{d_i}(t+\delta t) = \hat{h}$ if $\theta_i \geq \frac{\pi}{2}$, and $\hat{d_i}(t+\delta t) = -\hat{h}$ otherwise. This action can be identified in the FSM as `change_`d_i`()` in the self-transition happening in the `Wandering` state.
When the timer t_{w_i} expires ($t_{w_i} = T_{wmax}$), A_i transitions to the `Wandering and Detecting` state.

4) In the `Wandering and Detecting` state, A_i also updates $\vec{P_i}$ considering (1), and similarly as in `Wandering`, when the event **`Boundary_Detected`** takes place, the timer t_{w_i} is reset, and the direction $\hat{d_i}$ changes; besides, A_i comes back to the `Wandering` state as shown in the FSM.
While A_i is in the `Wandering and Detecting` state, it is also sensing nearby agents within a radius R_C through the `Sensing_within_`R_C`()` action, which evaluates the distance between A_i and other agents to be within the range of $[0, R_C]$.
If several agents are detected, only the first which triggers the **`Detect_joinable_agent_`**A_j event will be the target agent A_j to which A_i will join to form a structure. This event happens when the following conditions are fulfilled:

a) A_i is in `Wandering and Detecting` or in `Joined` state.

TABLE II
INTERNAL VARIABLES FOR A SELF-ASSEMBLY AGENT A_i

Variable	Description
$\vec{P_i} \in \mathbb{R}^3$	Position in space
$\hat{d_i} \in \mathbb{R}^3$	Movement direction
$t_{w_i} \in \mathbb{R}$	Timer when A_i is only `wandering`
$can\_detach \in \{True, False\}$	Boolean variable to allow A_i detach from its current structure
$t_{join_i} \in \mathbb{R}$	Timer to set $can\_detach = True$ when it expires

TABLE III
INTERNAL OBJECTS FOR A SELF-ASSEMBLY AGENT A_i

Object	Description
$\mathfrak{F}_i$	An indexed list of agents attached to A_i. Empty slots are represented by $\emptyset$ (null).
$\mathfrak{S}_{(i,f)}$	This a tuple $(slotIdx_i, A_f, slotIdx_f)$ attached to A_i, where A_f is an agent that belongs to the set $\mathfrak{F}_i$ and is followed by A_i. Both are connected through the slots with index $slotIdx_i$ (owned by A_i) and $slotIdx_f$ (owned by A_f).

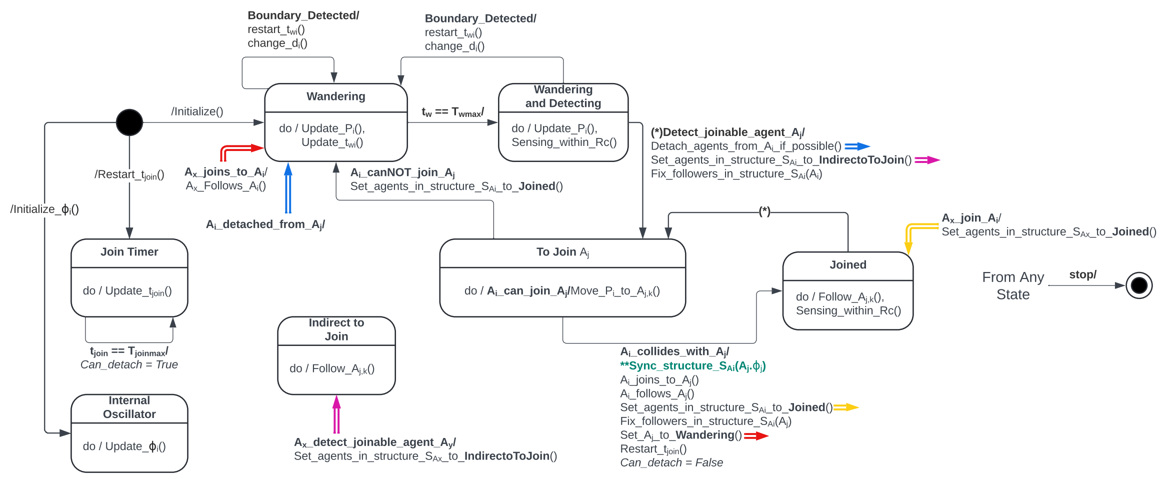

Boundary_Detected/
restart_t_{wi}()
change_d_i()
Boundary_Detected/
restart_t_{wi}()
change_d_i()
Wandering
do / Update_P_i(), Update_t_{wi}()
Wandering and Detecting
do / Update_P_i(), Sensing_within_Rc()
/Initialize()
t_w == T_{wmax}/
(*)Detect_joinable_agent_A_j/
Detach_agents_from_A_i_if_possible()
Set_agents_in_structure_S_{Ai}_to_IndirectoToJoin()
Fix_followers_in_structure_S_{Ai}(A_i)
/Restart_t_{join}()
A_x_joins_to_A_i/
A_x_Follows_A_i()
A_i_canNOT_join_A_j
Set_agents_in_structure_S_{Ai}_to_Joined()
/Initialize_ϕ_i()
A_i_detached_from_A_j/
(*)
A_x_join_A_i/
Set_agents_in_structure_S_{Ax}_to_Joined()
Join Timer
do / Update_t_{join}()
To Join A_j
do / A_i_can_join_A_j/Move_P_i_to_$A_{j,k}$()
Joined
do / Follow_$A_{j,k}$(), Sensing_within_Rc()
From Any State
stop/
t_{join} == $T_{joinmax}$/
Can_detach = True
Indirect to Join
do / Follow_$A_{j,k}$()
Internal Oscillator
do / Update_ϕ_i()
A_x_detect_joinable_agent_A_y/
Set_agents_in_structure_S_{Ax}_to_IndirectoToJoin()
A_i_collides_with_A_j/
**Sync_structure_S_{Ai}(A_j.ϕ_j)
A_i_joins_to_A_j()
A_i_follows_A_j()
Set_agents_in_structure_S_{Ai}_to_Joined()
Fix_followers_in_structure_S_{Ai}(A_j)
Set_A_j_to_Wandering()
Restart_t_{join}()
Can_detach = False

Fig. 5. Finite State Machine (FSM) for a self-assembly agent A_i. Coloured double arrows refer to changes produced by agents external to A_i; in that sense, note that A_i enters the `Indirect to Join` state only through an external event, thus A_i does not have an internal transition to it. Moreover, the event (*) and its actions are identical for both transitions. The action (**) serves as a synchronization point for the oscillators of the agents in the structure associated with A_i. This synchronization ensures that the current phase on the oscillator ϕ_j is matched.

b) The boundary of E is not detected.
c) A_j is within A_i's detection range, meaning a distance less or equal to R_C.
d) Both, A_i and A_j have a `slot` available; that is, as they are geometrically a cube, both have at least one of their six faces free to join a new agent. We define an available slot as k, where $k \in \mathbb{N} \cap [1, 6]$. Additionally, k_i belongs to A_i and k_j to A_j.
e) A_j is not yet included in the structure to which A_i belongs. Algorithm 1 illustrates the procedure implementing this condition.

Once these conditions are satisfied, we execute consecutively the next actions that prepare A_i to join A_j, as depicted in the FSM:

- `Detach_agents_from_`A_i`_if_possible()`: The agents associated with A_i detach from it only if A_i is not alone (there is at least one element in the slots' set $\mathfrak{F}_i$) and the flag $can\_detach = True$. This flag is disabled when the join process happens and is enabled when the timer t_{join_i} expires, which will be further explained in more detail. The detaching procedure is presented in Algorithm 2.
- `Set_agents_in_structure_`S_{A_i}`_to _IndirectoToJoin()`: All agents in the structure associated to A_i, know as S_{A_i}, are changed to `Indirect To Join` state using Algorithm 3. Note that in this algorithm, as well as in others described later, we use an auxiliary global variable called $visitedAgents$. This variable helps us keep track of the processed agents in the structure associated with the starting agent, thus preventing infinite loops during recursion.
- `Fix_followers_in_structure_`S_{A_i}`(`A_i`)`: When there are multiple agents in a structure, only one of them leads the movement. The leader is chosen by electing A_i as the one whose motion will be followed. Other agents may be indirectly connected to A_i, as shown in Fig. 3. The structure is reconfigured using Algorithm 4 to determine who follows whom, allowing A_i to be the leader.

Then, A_i transitions to the `To Join` A_j state.

Algorithm 1 Check if another agent A_j is in the same structure to which A_i belongs

```
procedure OTHERINSTRUCTURE($A_j$)
    agentsStack ← new stack of SelfAssemblyAgent
    visitedAgents ← new set of SelfAssemblyAgent
    agentsStack.push($A_i$)
    visitedAgents.add($A_i$)
    while agentsStack is not empty do
        $A_x$ ← agentsStack.pop()
        if $A_x = A_j$ then
            return true
        end if
        for all $A_{joined_x}$ in $\mathfrak{F}_x$ do
            if $A_{joined_x}$ is not ø then
                if $A_{joined_x}$ not in visitedAgents then
                    agentsStack.push($A_{joined_x}$)
                    visitedAgents.add($A_{joined_x}$)
                end if
            end if
        end for
    end while
    return false
end procedure
```

5) While A_i is in the `To Join` A_j state, A_i is constantly evaluating if it can join to A_j through the event A_i`_can_join_`A_j, which is `True` while the boundary

Algorithm 2 Detach an agent from its structure

```
1: procedure DetachFromStructure
2:     for $k \leftarrow 0$ to $|\mathfrak{F}_i| - 1$ do
3:         $A_k \leftarrow \mathfrak{F}_i[k]$
4:         if $A_k$ is not ø then
5:             $idx_k \leftarrow$ index of $A_i$ in $\mathfrak{F}_k$
6:             if $idx_k \geq 0$ then
7:                 set $\mathfrak{F}_k[idx_k]$ to ø
8:                 if $A_k$.IsAlone then
9:                     set $\mathfrak{F}_k[idx_k]$.CurrentState to Wandering
10:                end if
11:            end if
12:        end if
13:    end for
14:    Fill $\mathfrak{F}_i$ with ø
15: end procedure
```

Algorithm 3 Set a $state$ to the structure associated with agent A_i

```
1: procedure SetToState($A_x$, $state$, $visitedAgents$)
2:     Add $A_x$ to $visitedAgents$
3:     Set $A_x$.CurrentState to $state$
4:     for $k \leftarrow 0$ to $|\mathfrak{F}_x| - 1$ do
5:         $A_k \leftarrow \mathfrak{F}_x[k]$
6:         if $A_k$ is not ø then
7:             if $A_k$ not in visitedAgents then
8:                 SetToState($A_k$, $state$, $visitedAgents$)
9:             end if
10:        end if
11:    end for
12: end procedure
13: $visitedAgents \leftarrow$ new set of SelfAssemblyAgent
14: SetToState($A_i$, $state$, $visitedAgents$)
```

Algorithm 4 Fix followers for the structure S_{A_i} having A_i as the leader

```
1: procedure FixFollowers($A_x$, $visitedAgents$)
2:     Add $A_x$ to $visitedAgents$
3:     for $k \leftarrow 0$ to $|\mathfrak{F}_x| - 1$ do
4:         $A_k \leftarrow \mathfrak{F}_x[k]$
5:         if $A_k$ is not ø then
6:             if $A_k$ not in visitedAgents then
7:                 $slotIdx_k \leftarrow$ index of $A_x$ in $\mathfrak{F}_k$
8:                 $A_f \leftarrow A_x$
9:                 $slotIdx_f \leftarrow k$
10:                $\mathfrak{S}_{(k,f)} \leftarrow (slotIdx_k, A_f, slotIdx_f)$
11:                FixFollowers($A_k$, $visitedAgents$)
12:            end if
13:        end if
14:    end for
15: end procedure
16: $visitedAgents \leftarrow$ new set of SelfAssemblyAgent
17: FixFollowers($A_i$, $visitedAgents$)
```

of E is not detected and both agents have a slot available to join. Consequently, A_i is updating its position $\vec{P_i}$ as in (1), but towards the available slot of A_j; thus, the direction $\hat{d_i}$ should be replaced by $\hat{d}_{(i,j)}(k_j, t)$, which needs the following components to be calculated:

a) For any agent A_i, the normalized position of a slot k_i relative to the agent's position $\vec{P_i}$, is denoted as $\hat{ds_i}$ and given by (2), where $\hat{x}$, $\hat{y}$, and $\hat{z}$ are the unit vectors in the directions of the x-axis, y-axis, and z-axis respectively in 3D.

$$\hat{ds}_i(k_i) = \begin{cases} \hat{x}, & \text{if } k_i = 1 \\ -\hat{x}, & \text{if } k_i = 2 \\ \hat{y}, & \text{if } k_i = 3 \\ -\hat{y}, & \text{if } k_i = 4 \\ \hat{z}, & \text{if } k_i = 5 \\ -\hat{z}, & \text{if } k_i = 6 \end{cases} \tag{2}$$

b) The distance between the centre of the cube, which is the position $\vec{P_i}$, and the centre of a face. Thus, half of the length of the cube side; that is, $\frac{\lambda}{2}$.

c) With the previous two items, we can calculate the absolute position $\vec{P_i}(k_i, t)$ for a slot k_i that belongs to A_i as in (3).

$$\vec{P_i}(k_i, t) = \vec{P_i}(t) + \frac{\lambda}{2}\hat{ds}_i(k_i) \tag{3}$$

Having these components, $\hat{d}_{(i,j)}(k_j, t)$ is obtained from (4). Note that the absolute slot position defined in (3) is calculated over A_j.

$$\hat{d}_{(i,j)}(k_j, t) = \frac{\vec{P_j}(k_j, t) - \vec{P_i}(t)}{\|\vec{P_j}(k_j, t) - \vec{P_i}(t)\|} \tag{4}$$

On the contrary, A_i`_cannot_join_`A_j could be triggered if either agent does not have an available slot, which might occur if a third agent joins one of the two before. Moreover, A_i`_canNOT_join_`A_j can happen when the boundary of E is detected, allowing to change the direction $\hat{d_i}$. In either case, when A_i`_canNOT_join_`A_j is triggered, A_i transitions back to the `Wandering` state, and additionally, sets all agents in the structure that contains A_i to the `Joined` state using Algorithm 3.

If A_i successfully collides with A_j; that is, the A_i**_collides_with_**A_j event is triggered, then the following actions are performed for the join mechanism:

- `Sync_structure_`S_{A_i}`(`$A_j.\phi_j$`)`: Before A_i joins A_j, the oscillators in the structure associated with A_i are synchronized with the phase of the oscillator ϕ_j that belongs to A_j as described in Section III-D.
- A_i`_joins_to_`A_j`()`: This action allows A_j to be included in the index k_i of $\mathfrak{F}_i$; and likewise, A_i being included in the index k_j of $\mathfrak{F}_j$. This is how two agents join each other.

- `$A_i$_follows_$A_j$()`: A_i starts to follow the movement of A_j through the assignation $\mathfrak{S}_{(k_j,i)} \leftarrow (k_j, A_i, k_i)$. (See Table III for the description of this tuple).
- `Set_agents_in_structure_$S_{A_i}$_to _Joined()`: All agents are set to the `Joined` state using Algorithm 3.
- `Fix_followers_in_structure_$S_{A_i}$($A_j$)`: It uses Algorithm 4 to choose A_j as the leader for the collective motion and adjust all other agents in the structure S_{A_i} accordingly.
- `Set_$A_j$_to_Wandering()`: A_i can directly influence A_j, so A_i requests that A_j set its state to `Wandering`. This makes A_j the one that moves the entire structure.
- `Restart_$t_{join_i}$()` The timer t_{join_i} from A_i is reset. As we can see in the FSM, the timer t_{join_i} is running in parallel with the rest of the behaviour of agent A_i and, when it expires, the flag *can_deatch* is set to $True$. As mentioned previously, this flag allows A_i to detach or not from its current structure when it detects a joinable agent.
- `Can_detach = False`: Apart from resetting the timer t_{join_i}, A_i sets the flag *can_deatch* to $False$, allowing A_i to be connected to its current structure at least during T_{jmax}.

After these actions, A_i transitions to the `Joined` state.

6) In the `Joined` state, A_i is following an agent A_f; that is, A_f is dragging A_i and both are connected through the slots index k_i and k_f respectively. Geometrically, the faces k_i and k_f should be stuck together, so A_i should be rotated accordingly; then, $\vec{P_i}$ is updated using (5). Here, we use the absolute position of the connected slot k_f from A_f given by $\vec{P_f}(k_f, t)$ denoted previously by (3), and the normalized position $\hat{ds}_f$ of slot k_f relative to the agent's position $\vec{P_f}$, presented before in (2).

$$\vec{P_i}(t) = \vec{P_f}(k_f, t) + \frac{\lambda}{2}\hat{ds}_f(k_f) \tag{5}$$

Additionally, A_i is also detecting nearby agents similar to the `Wandering and Detecting` state, and if A_i detects one, it performs the same protocol described previously if possible as indicated in the FSM with (*).

7) The `Indirect to Join` state is set only by an external agent to A_i as described previously. Note that in the FSM, it only transitions due to an external event. When A_i is in this state, it means that it is following one of its attached agents using (5). The leader of the structure that A_i is part of is in the `To Join` A_j state, so A_i is being dragged and must not be in a state that is performing other activities.

In the FSM depicted in Fig. 5, the coloured double arrows represent the influence of an external agent on A_i. As previously explained, A_i can alter the state of other agents within its current structure in various scenarios, and this is depicted by these arrows.

Agents have the option to receive a **stop** event, which will signal the agent to terminate its function, regardless of its current state.

D. Sound Mapping and Music-Making

The algorithm described above can be used for any application that wants to take advantage of the emergent dynamics produced when a collection of these agents interact with each other. In this work, we use these dynamics for music-making; specifically, we explore rhythmic compositions with emergent melodic and harmonic patterns, as well as sound dynamics. The following subsections explain in detail how we map sound and music properties to the proposed self-assembly algorithm.

Synchronization for Musical Rhythm: Every agent has an internal pulse oscillator, as explained previously in Section III-B, that generates regular pulses to trigger sound events. The starting phase ϕ of every oscillator is assigned randomly when they are initialized and is updated in parallel with the agent's behaviour, as shown in the FSM in Fig. 5. For an agent A_i, its phase ϕ_i changes according to the update rule described in Section III-B. All agents use the same frequency ω for their oscillators, which is a parameter that can be configurable to have a faster or slower train of pulses.

As every agent starts at a different phase, they are not synchronised when a self-assembly session starts; however, through the assembly and disassembly mechanism, they synchronize each other when phases are matched in a structure at the moment of transition between the `To Join` A_j and the `Joined` state when the event A_i**_collides_with_**A_j is triggered, as depicted in the FSM. When this happens, the action `Sync_structure_$S_{A_i}$($A_j.\phi_j$)` is executed according to Algorithm 5. This process allows all oscillators in the structure associated with A_i to synchronize their phases with the phase ϕ_j that belongs to agent A_j, being ϕ_j the phase ϕ_{sync} that allows the synchronization.

Algorithm 5 Synchronize all the oscillators in the structure associated with Agent A_i by setting their phases to ϕ_{sync}

```
1: procedure SyncStruct($A_x$, $\phi_{sync}$, $visitedAgents$)
2:     Add $A_x$ to $visitedAgents$
3:     $A_x.\phi_x \leftarrow \phi_{sync}$
4:     for $k \leftarrow 0$ to $|\mathfrak{F}_x| - 1$ do
5:         $A_k \leftarrow \mathfrak{F}_x[k]$
6:         if $A_k$ is not $\emptyset$ then
7:             if $A_k$ not in visitedAgents then
8:                 SyncStruct($A_k$, $\phi_{sync}$, $visitedAgents$)
9:             end if
10:        end if
11:    end for
12: end procedure
13: $visitedAgents \leftarrow$ new set of SelfAssemblyAgent
14: SyncStruct($A_i$, $\phi_{sync}$, $visitedAgents$)
```

Additionally, in order to increase the diversity for emergent rhythms, the actual sound event would be played according to a simple division rule that considers the expression $c : b$

where $c, b \in \mathbb{N}^+$. It means that b *evenly distributed events are triggered within c cycles of the oscillator*; for instance, for $1:4$ we trigger 4 events within each oscillator cycle; that is, if $\omega = 1Hz$ then an event occurs every 0.25 seconds; likewise, for $4:2$, we trigger 2 events every 4 cycles, which for $\omega = 1Hz$ means to fire an event every 2 seconds. At initialization, every agent is assigned a random division rule from a predefined set of rules as part of the configurable parameters of the agents.

Musical Notes Assignation and Playback: At initialization, each of the six slots in an agent is assigned a musical note, which is chosen randomly from a set of defined musical arrangements, such as musical scales. To provide more diversity, the octaves of the notes are also randomly increased or decreased. We represent musical notes using standard MIDI values. In our implementation, we randomly choose one scale from the *major*, *major blues*, and *major pentatonic* scales; then, the chosen scale is used for the assignation described before.

When an agent A_i is not isolated, meaning that at least one agent is attached to one of its slots, a note is played back if its oscillator ϕ_i triggers an event according to its division $c:b$.

The synthesis of the note is given by the process described below, which takes into account the conversion from a MIDI value to an audible tone.

Sound Synthesis and Structure Size: In our implementation, we use a sine wave tone with a pitch obtained from converting the MIDI note to the corresponding sound frequency in hertz (Hz) of a musical note. We use an amplitude envelope ADSR (attack, decay, sustain, release) to play the note for a short period of time.

We assign longer attack and release times for agents attached to larger structures based on the number of agents that compose the structure. Thus, to calculate this number, we traverse the structure recursively, similarly to Algorithms 3, 4, 5, to count the total elements. These dynamic changes in the amplitude envelope allow sounds for bigger structures to act as a base for shorter sounds, as can be noticed in the video[2]. Additionally, the size of a structure is used to adjust the gain of the sine tones, preventing the sounds from becoming overwhelming when played simultaneously.

IV. Results

In this section, we analyze the configurable parameters presented in Table I. The following subsections describe the experiments that utilize the same values for parameters specified in Table IV. For each experiment, we specify the additional parameters that vary in the study apart from the ones in Table IV. Note that, considering the relatively high amount of parameters and possible combinations, we performed an initial exploration of arbitrary values to obtain a subjectively "appealing" musical result. We focus on this configuration to demonstrate the capabilities of the strategy proposed in this paper; as such, our goal is to study the synchronization behaviour and the resulting emergent structures over time, as well as their reflection in the sound output. As an additional note, we set trials to 30 to have a balance between statistical power and execution time.

TABLE IV
Values for shared parameters on every experiment

R_E	λ	v	T_{wmax}	R_c
2 m	0.1 m	1 m/s	1 s	0.4 m

A. Synchronization Time

We conducted a scalability experiment focusing on various numbers of agents, denoted by N. In particular, we ran 30 sessions for each value of N in the set $\{5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100\}$, for a total of 330 runs. Each run lasted for 5 minutes. All agents shared the same parameter values shown in Table IV, T_{jmax} = 5 s, and a frequency $\omega = 1Hz$ for their internal oscillators.

We recorded the timestamp of each oscillator cycle for all agents and calculated their *desynchronization degree* of one agent compared to the others. We defined the desynchronization degree for an agent A_i in two steps:

1) Given a timestamp ts for agent A_i as $A_i.ts_p$, where $p \in [1, m_i]$ and m_i is the number of collected samples in a run; we calculate the time difference $\Delta ts_{(i,j)}(p)$ for a sample p, which represents the time between the timestamp $A_i.ts_p$ and the closest timestamp from another agent A_j, given by (6).
$$\Delta ts_{(i,j)}(p) = \min_{1 \leq q \leq m_j} |A_j.ts_q - A_i.ts_p| \qquad (6)$$
2) Then, we obtain the desynchronization degree for A_i regarding the rest of agents for a sample p, denoted as $\Delta Ts_i(p)$, by averaging $\Delta ts_{(i,j)}(p)$ for all agents $j = 1, \ldots, N; j \neq i$ as in (7).
$$\Delta Ts_i(p) = \frac{1}{N-1} \sum_{j=1; j \neq i}^{N} \Delta ts_{(i,j)}(p) \qquad (7)$$

As an example, we plotted the desynchronization degree over time for a run with 50 agents in Fig. 6. Note that, as all agents are assigned random phases at the beginning, the desynchronization degree is close to 0.5 seconds for oscillators where $\omega = 1Hz$. In this example, we see that all agents are synchronized when this metric is close to zero, which happens around 70 seconds after the session starts.

Considering this, we processed all runs for the different number of agents from the previously defined set and obtained the results for the synchronization time displayed in Fig. 7. Furthermore, we perform a linear regression *(Intercept: 144.93, Slope: -0.54, both with p-value $<$ 0.05)*, also plotted in Fig. 7, with an $R^2 = 0.084$, indicating that approximately 8.4% of the variance in synchronization time can be explained by the number of agents N, which is considered a low value, and suggest that the linear model does not explain a large portion of the variability. These results indicate a slight

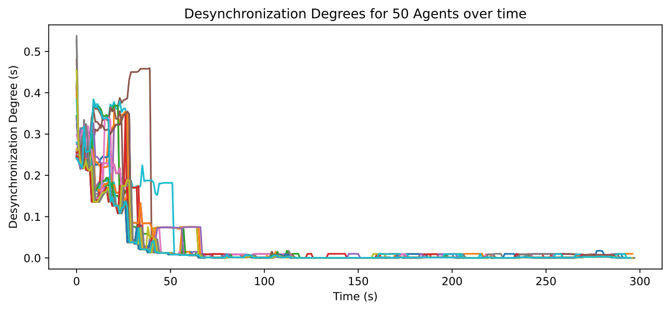


Fig. 6. Desynchronization degrees for 50 agents. Each agent plots its degree with a different colour. The synchronization point in this run is close to 70 seconds when this metric is close to zero.

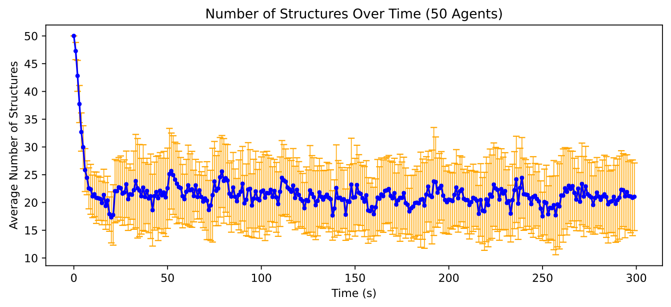


Fig. 8. Average number of structures over time for 50 agents. The number of structures reduces after approximately 10 seconds and oscillates between 10 and 30 structures during the session.

decrease in synchronization time, although, due to the high variability per group, we might not have lower times as N increases.

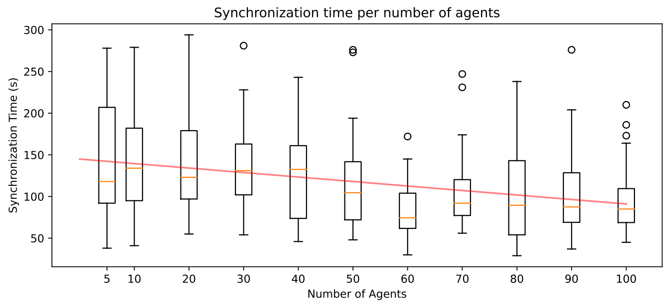


Fig. 7. Synchronization time per number of agents. Some groups show high variability in synchronization time, which is slightly decreasing as N increases. We have faster synchronization when $N = 60$.

B. Emergent Structures

We studied how the *number* and *size* of emergent structures evolve during a self-assembly session, being the *size* the number of agents that compose a structure.

For this purpose, we ran 30 trials consisting of 50 agents each, with the values from Table IV and $T_{jmax} = 5s$. During each run, we recorded the number and size of structures every second for 5 minutes. The results for the number of structures over time, along with the variability within the 30 runs, are presented in Fig. 8. During the session, the number of structures decreases as more agents join to form bigger structures. At the start of the session, there are 50 structures since a single agent can be considered a structure of size 1. However, for this particular number of agents, the number of structures reduces after approximately 10 seconds and oscillates between 10 and 30 structures during the session.

With the same experimental setting, we varied the time T_{jmax} that controls the waiting period for an agent before allowing it to detach from its current structure, according to the self-assembly strategy. We plotted 3 runs with different values of T_{jmax} (15, 35, and 50 seconds) in Fig. 9. These plots depict two types of values: the number of structures in a blue continuous line, and the structures per se represented as small red dots drawn in a vertical line on a specific time denoting their size; both values are normalized between 0 and 1. For instance, in Fig. 9c, we start with all structures of size 1, and we finish in around 50 seconds with only one structure at full size.

We observed that, with a T_{jmax} of approximately 15 seconds, we start to see a repetitive pattern in which we have fewer but bigger structures, and then agents separate from each other to form more and smaller structures. However, as T_{jmax} increases, this repetitive process takes more time, and if T_{jmax} is large enough, we could end up with a single large structure sooner. Moreover, although in Fig. 9a we can identify this pattern for the number of structures (blue line), the sizes (red dots amount) are slightly more diverse.

C. Sound Analysis

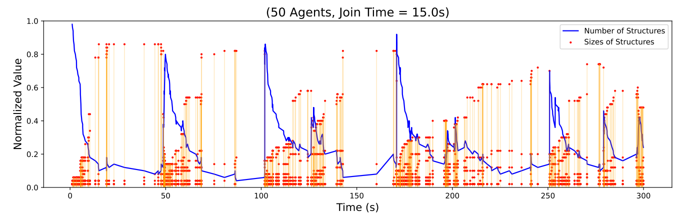
In Fig. 10, we plot the waveform and spectrogram for the third minute of audio corresponding to our video[2], which uses the settings from Table IV, $T_{jmax} = 5s$, $\omega = 0.25$, and a set of division rules $\{1:1, 1:2, 1:4, 1:8\}$ to choose from. We can observe that the waveform has peaks in various sections, which indicates rhythmic variations in the sound energy over time. Additionally, we can identify the impact of the size of structures in the amplitude envelope for sound events in the spectrogram, where we can find longer durations and fade tails as indicated in Fig. 10b, meaning that the amplitude envelope has higher attack and release times. Furthermore, we can identify sounds that are regularly separated over long periods, which demonstrates the periodicity of the oscillators.

V. Discussion

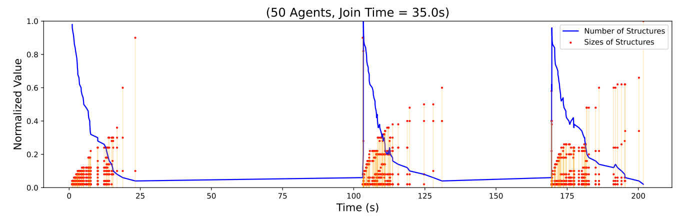
A. Algorithm Design

The self-assembly algorithm we propose is less constrained compared to the works referenced in Section II. This is due to its operation in a virtual environment, which allows for instant connections and disconnections, and the omission of disturbances. However, we share some elements, such as recursivity, as used in [4], and the ability to shape relatively large structures, as shown in simulations referenced in [5].

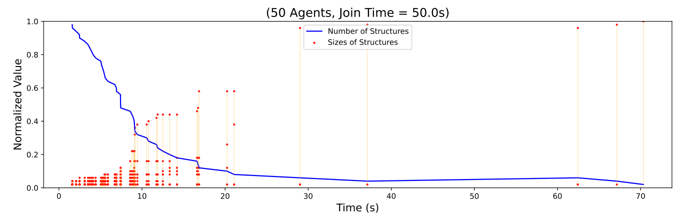
Furthermore, our algorithm is designed to work in a decentralized environment, using local radius communication to



(a)

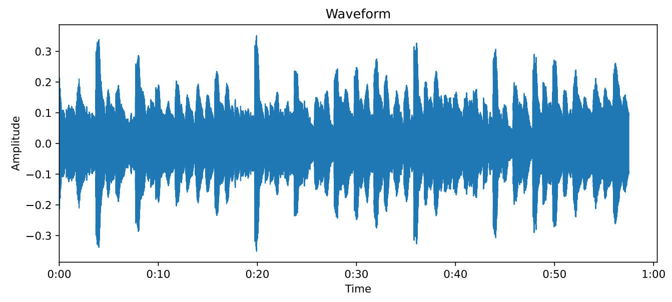


(b)

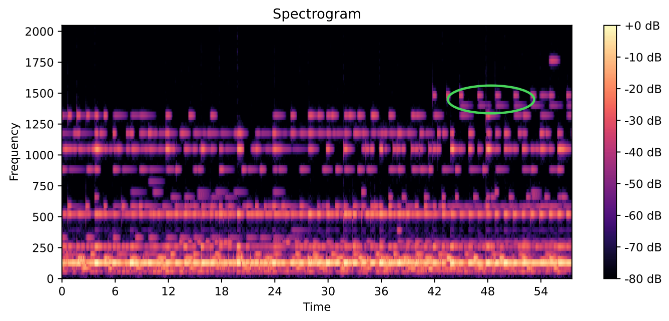


(c)

Fig. 9. Normalized number and sizes of structures over time. When T_{jmax} is around 15 seconds (a), a periodic pattern for the number of structures over time appears, and disappears as T_{jmax} increases to 35 (b) and 50 (c) seconds.



(a)



(b)

Fig. 10. (a) Waveform and (b) Spectrogram for the third minute of a sound recording for 50 agents with the settings from Table IV and $T_{jmax} = 5s$. We illustrate, using a green oval, an example of how a structure affects the length and amplitude envelope of sound events in the spectrogram.

detect other agents and allow data to flow through several connected agents, but not with external ones. However, depending on the implementation, certain mechanisms may be centralized, especially in a virtual environment. For example, the detection of nearby agents requires knowing the distance from one agent to all the others, which is the case in our implementation. Moreover, we develop our system in Unity using the standard method of running all agents in Unity's main thread. As a result, the self-assembly algorithm is executed synchronously according to the order of instantiation of the agents. To improve performance, we can leverage Unity's multi-threading capabilities.

Additionally, it is important to consider potential performance issues with certain routines. For example, we can use either iterative (as in Algorithm 1) or recursive (as in Algorithms 3, 4, and 5) procedures. Although recursive procedures seem to be a natural choice for this strategy, if we face performance issues, we can change to an iterative one. However, the choice of traversing a structure would depend on the platform. For potential physical systems, it could be challenging to establish communication in a structure, and problems such as communication failures or latency might affect the algorithm's functioning.

B. Synchronization

In Section IV-A, we present the results of several self-assembly sessions, focusing on desynchronization degrees and synchronization time for a specific set of parameters. Our findings show that it takes several seconds to achieve synchronicity, and this process is highly variable. Although these results are not directly comparable with the works on synchronization mentioned in Section II, we offer a different approach that contributes to the dynamics of music composition while synchronization is being achieved.

Moreover, synchronization appears to speed up slightly with an increase in the number of agents. This is expected as more agents occupy the same area and therefore have more encounters. However, we have observed that this is not always the case. For example, we found that the fastest synchronization occurred when $N = 60$, compared to larger groups. It is important to note that this finding is limited to the specific parameters we used, and changing other parameters, such as T_{jmax}, could impact synchronization behaviour.

These results are useful for estimating when agents would achieve synchronization, especially when additional actions need to be performed around that time, such as synchronized accompaniment for sound and music.

C. Emergent Behaviour

We examined the number and size of structures over time in Section IV-B. We discovered periodic patterns with certain settings for a T_{jmax} of approximately 15 seconds. Additionally, we observed that we could achieve a complete structure at a specific time by increasing T_{jmax} continuously. These results indicate that we can adjust this parameter to modify assembly and disassembly behaviours as per our requirements, which is useful to influence the development of emergent music performance over time.

Additionally, note that there is a sudden shift from a low number of structures to a high number. This is also observed in the video[2], where structures suddenly dissolve. This is an unexpected behaviour that emerges when agents detach from each other. This detachment triggers a chain reaction, causing other agents to detach and enter the `Wandering` state while they are alone. This behaviour creates opportunities for additional sound and music mappings that could potentially enhance the composition.

D. Music-Making

The video[2] and the sound analysis in Section IV-C demonstrate how rhythmic music unfolds over time. We can argue that initial desynchronization might contribute to the expressive quality of the musical piece, considering that *syncopation*[5] can help to develop the generative composition at initialization.

Furthermore, the sound dynamics are influenced by the number and size of the structures in play, enabling the creation of harmonic foundations and melodic patterns over the rhythm. While the sound and music mapping proposed in Section III-D can be extended to incorporate other aspects of the movement dynamics from the agents, we should be careful not to overload the sound environment depending on our musical intentions.

E. Future Work

To improve the music generation process, we can extend our study by exploring various parameters and their combinations. This will help us discover new emergent behaviours and properties of the algorithm. We plan to continue our exploration by adding a feedback mechanism that influences the movement dynamics through the music outcome.

The embodiment of these agents raises the opportunity of incorporating human interaction into the process, making it part of a *human-swarm interactive music system* [13]. Technologies like Extended Reality (XR) can enable us to manipulate these agents directly, giving us the ability to involve humans in the movement dynamics.

Implementing physical systems using this algorithm might require a significant effort in finding a particular mechanical solution, a suitable communication strategy, and a proper modification of the algorithm. However, exploring this approach for robotic applications is worth considering.

Furthermore, the self-assembly strategy presented can be applied to other applications besides music, and we encourage its utilization in other fields where it can be beneficial.

VI. Conclusions

We present an algorithm for self-assembly and synchronization of agents that have oscillators embodied in 3D cube shapes for generating rhythmical behaviour. We observed the synchronization time of a group of agents as we increased their number under certain parameters. We found that this time slightly decreases as we have more agents, but it varies highly across different groups. Additionally, we noticed that emergent structures formed in real-time exhibited quasi-regular patterns of increments and decrements in both number and size over time.

These particular behaviours help to produce emergent rhythmical harmonies and melodies when the embodied oscillators play musical notes from predefined music structures. The results of this study contribute to the development of immersive virtual environments for sound and music, offering potential for creative human interaction.

[5]https://www.oxfordreference.com/display/10.1093/acref/9780199203833.001.0001/acref-9780199203833-e-8887

References

[1] G. M. Whitesides and B. Grzybowski, "Self-Assembly at All Scales," *Science*, vol. 295, no. 5564, pp. 2418–2421, Mar. 2002. [Online]. Available: https://www.science.org/doi/10.1126/science.1070821

[2] A. S. Iyer and K. Paul, "Self-assembly: a review of scope and applications," *IET Nanobiotechnology*, vol. 9, no. 3, pp. 122–135, Jun. 2015. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1049/iet-nbt.2014.0020

[3] Z. Nagy, R. Oung, J. Abbott, and B. Nelson, "Experimental investigation of magnetic self-assembly for swallowable modular robots," in *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*. Nice: IEEE, Sep. 2008, pp. 1915–1920. [Online]. Available: http://ieeexplore.ieee.org/document/4650662/

[4] L. Zhang, Y. Jiao, Y. Huang, Z. Wang, and H. Qian, "Parallel Self-assembly for Modular USVs with Diverse Docking Mechanism Layouts," Jan. 2024, arXiv:2401.15399 [cs]. [Online]. Available: http://arxiv.org/abs/2401.15399

[5] E. Bonabeau, M. Dorigo, and G. Theraulaz, *Swarm Intelligence: From Natural to Artificial Systems*. Oxford University Press, Oct. 1999. [Online]. Available: https://academic.oup.com/book/40811

[6] R. Gro, M. Bonani, F. Mondada, and M. Dorigo, "Autonomous Self-Assembly in Swarm-Bots," *IEEE Transactions on Robotics*, vol. 22, no. 6, pp. 1115–1130, Dec. 2006. [Online]. Available: http://ieeexplore.ieee.org/document/4020359/

[7] R. O'Grady, R. Gross, A. L. Christensen, F. Mondada, M. Bonani, and M. Dorigo, "Performance benefits of self-assembly in a swarm-bot," in *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*. San Diego, CA, USA: IEEE, Oct. 2007, pp. 2381–2387. [Online]. Available: http://ieeexplore.ieee.org/document/4399424/

[8] E. Tuci, V. Trianni, C. Amatzis, A. Christensen, and M. Dorigo, "Self-Assembly in Physical Autonomous Robots: the Evolutionary Robotics Approach," in *Artificial Life XI*, S. Bullock, J. Noble, R. Watson, and M. A. Bedau, Eds. United States of America: MIT Press, Aug. 2008, pp. 616–623.

[9] L. Bigo, A. Spicher, and O. Michel, "Spatial Programming for Music Representation and Analysis," in *2010 Fourth IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshop*. Budapest, Hungary: IEEE, Sep. 2010, pp. 98–103. [Online]. Available: http://ieeexplore.ieee.org/document/5729604/

[10] L. Bigo and A. Spicher, "Self-Assembly of Musical Representations in MGS," *International Journal of Unconventional Computing*, vol. 10, no. 3, pp. 219–236, 2014.

[11] K. Nymoen, A. Chandra, and J. Torresen, "The challenge of decentralised synchronisation in interactive music systems," *Proceedings - IEEE 7th International Conference on Self-Adaptation and Self-Organizing Systems Workshops, SASOW 2013*, pp. 95–100, 2013, publisher: IEEE ISBN: 9781479950867.

[12] K. Nymoen, A. Chandra, K. Glette, and J. Torresen, "Decentralized harmonic synchronization in mobile music systems," *2014 IEEE 6th International Conference on Awareness Science and Technology, iCAST 2014*, vol. 257906, no. 257906, 2014, publisher: IEEE ISBN: 9781479973736.

[13] P. Lucas and K. Glette, "Human-Swarm Interactive Music Systems: Design, Algorithms, Technologies, and Evaluation," *Proceedings of the 16th International Symposium on Computer Music Multidisciplinary Research*, Nov. 2023, publisher: Zenodo. [Online]. Available: https://zenodo.org/doi/10.5281/zenodo.10113080