Adaptive Agent Abstractions to Speed Up Spatial Agent-Based Simulations

Abbas Sarraf Shirazi[a,*], Timothy Davison[a], Sebastian von Mammen[b], Jörg Denzinger[a], Christian Jacob[a,c]

[a] *Dept. of Computer Science, Faculty of Science, University of Calgary, Canada*
[b] *Institut für Informatik, University of Augsburg, Germany*
[c] *Dept. of Biochemistry & Molecular Biology, Faculty of Medicine, University of Calgary, Canada*

Abstract

Simulating fine-grained agent-based models requires extensive computational resources. In this article, we present an approach that reduces the number of agents by adaptively abstracting groups of spatial agents into meta-agents that subsume individual behaviours and physical forms. Particularly, groups of agents that have been clustering together for a sufficiently long period of time are detected by *observer* agents and then abstracted into a single meta-agent. *Observers* periodically test meta-agents to ensure their validity, as the dynamics of the simulation may change to a point where the individual agents do not form a cluster any more. An invalid meta-agent is removed from the simulation and subsequently, its subsumed individual agents will be put back in the simulation. The same mechanism can be applied on meta-agents thus creating adaptive abstraction hierarchies during the course of a simulation. Experimental results on the simulation of the blood coagulation process show that the proposed abstraction mechanism results in the same system behaviour while speeding up the simulation.

Keywords: Agent-based simulation, Abstraction, Optimization, Online learning

1. Introduction

Agent Based Models (ABM) provide a natural means to describe complex systems, as agents and their properties have a convenient mapping from the entities in real world systems. The interaction of agents in ABM gives rise to

*Corresponding author
Email addresses: asarrafs@ucalgary.ca (Abbas Sarraf Shirazi), tbdaviso@ucalgary.ca (Timothy Davison), sebastian.von.mammen@informatik.uni-augsburg.de (Sebastian von Mammen), denzinge@ucalgary.ca (Jörg Denzinger), cjacob@ucalgary.ca (Christian Jacob)

an interesting concept in the study of complex systems: emergent phenomena, higher-level properties or behaviours that are not easily traceable in the lower-level entities [1]. Moreover, ABM capture discontinuity in individual behaviours, which is difficult when modelling with an alternative like differential equations [2].

The flexibility of ABM comes at a computational cost. As the granularity of a model increases, so do the computational resources needed to simulate all of the interactions among the agents, which directly translates into longer simulation times. Some researchers have restricted agent interactions to be only among neighbouring agents in a two or three-dimensional lattice [3, 4]. However, changing the interaction topography among agents is a necessary feature in some models, e.g. developmental processes [5]. Others have utilized parallel computing to meet the computational demands of ABM [6, 7]. Finally, many researchers have proposed super-individuals [8]: agents that encompass other agents, e.g. a single super red blood cell agent that subsumes and represents thousands of individual red blood cell agents.

In this work, we propose a framework to adaptively abstract groups of spatial agents into super-individuals, a.k.a meta-agents during the course of a simulation. To this end, *observer* agents are immersed in the simulation to monitor groups of agents. The *observers* try to detect a cluster of agents that have adhered to one another for a sufficiently long duration of time. Once an *observer* finds such a cluster, it abstracts the agents into a single meta-agent that subsumes both the behaviour and the structure of the individual agents in that cluster. As the dynamics of the simulation change, groups of agents may no longer stick together and therefore the *observer* needs to break down those meta-agents into their constituent individual agents. An unsupervised validation mechanism ensures the validity of meta-agents by periodically monitoring whether they should continue to subsume their agents. Since meta-agents have the same basic definition as the individual agents, the same abstraction process is applied on them, thus making adaptive abstraction hierarchies during the course of the simulation.

The remainder of this paper is organized as follows. Section 2 reviews related works both in solving the problem of scalability and in dealing with higher-order patterns in agent-based simulations. Section 3 gives a formal definition, along with a computational timing analysis of our component-based agent framework – *LINDSAY Composer*. Section 4 presents our abstraction framework with a detailed description of the involved steps and algorithms. We conclude this section with a computational timing analysis of our abstraction. In order to demonstrate the effectiveness of this approach, we apply it to an agent-based blood coagulation simulation and report the results in Section 5. Finally, concluding remarks are presented in Section 6.

2. Related Work

Agent based models operate at the individual level and describe potentially numerous behaviours for all of their constituent units. Simulating all of the

individual behaviours is therefore considered to be extremely computationally intensive [2, 9, 10, 11, 12]. It has been suggested that abstracting higher-order patterns could reduce the computational complexity of ABM without introducing much overhead [13, 12, 14]. In this section, we briefly describe the attempts made to address the problem of scalability and performance in ABM, then we review the works that motivated this research.

2.1. Scalability and Performance in ABM

Bonabeau points out that despite increasing computational power, simulating all the individual behaviours in ABM still remains a problem when it comes to modelling large-scale systems [2]. Research in improving the scalability of ABM is roughly categorized into two groups: (1) parallel computing, and (2) grouping similar agents into a single agent.

The first category, parallel computing, tries to concurrently simulate clusters of agents that interact primarily with one another without much intra-cluster communication. Efficiency is improved as long as the time spent on synchronization is much less than the time spent on computation [6]. Scheutz and Schermerhorn developed a framework with two algorithms for the automatic parallelization of ABM [6]. Particularly, they developed a separate algorithm for spatial agents, as their location data can efficiently determine in what cluster they should be simulated.

Along the same line, Lysenko and D'Souza propose a framework to use Graphics Processing Units (GPU) to parallelize an agent-based simulation [7]. They utilize a technique in General Purpose Computing on GPUs called state textures [15] to map each agent to a pixel. A pixel is defined by its colour components: Red, Green, Blue, and Alpha (RGBA). Each numerical property of an agent is thus mapped to a colour component. If an agent cannot be squeezed into four floating point values, then extra colour buffers should be used, which in turn adds to the complexity of the problem.

The second category of grouping similar agents deals with the granularity of an agent. For example, super-individuals can represent groups of agents. Scheffer et al. suggest assigning an extra variable to each agent to denote how many agents it represents [8]. More advanced algorithms have been proposed to find super-individuals during the course of a simulation. Stage et al. propose an algorithm called COMPRESS to aggregate a cluster of agents into one agent [16]. They divide their algorithm into two stages to avoid applying a time-consuming clustering algorithm on the space of all the attributes in all the agents. In the first stage, they calculate a linear combination of attributes l_i for each agent i by applying principal component analysis (PCA) [17]. Then this list is sorted to find n clusters of agents with the largest gaps in l_i. In the next stage the clusters are further subdivided based upon their variance until the variance is within a given range. The first stage maintains overall system variations while the second stage reduces the intra-cluster variations.

COMPRESS is a static algorithm, in that once a cluster of agents is replaced by one agent, the original agents will not be released back into the simulation. Wendel and Dibble extend the static COMPRESS algorithm with the

Dynamic Agent Compression (DAC) algorithm in which higher-order agents are created and destroyed based on the heterogeneity of agents in the system [18]. They define two special agents in their system: (1) *container agents* which are the higher-order agents, and (2) a *compression manager* which handles all the queries to individual agents thus making the container agents invisible to the model. It also creates and destroys other agents. For example, upon receiving a create request from the model, the compression manager decides if it has to create a new individual or whether the create request can be ignored, as there already exists an agent with the same attributes. In DAC a container agent monitors its encompassed agents and upon detecting a difference in behaviour gives the changed agents to the compression manager as newly instantiated individuals.

Sarraf Shirazi et al. present an algorithm to abstract agent interaction processes during the course of a simulation [14]. They replace the actions of many agents with a single group meta-action thus speeding up the simulation. Specifically, they introduce a special observer agent that logs the simulation state and learns the parameters of individual actions. Similar to [18], they also propose a validation mechanism capable of destroying meta-actions to ensure that they execute the learned meta-actions as long as they do not deviate significantly from the individual actions.

2.2. Higher-Order Patterns in ABM

Emergence is the appearance of macro level patterns in a system that are not described by the properties of its parts [19]. According to Müller [20], an emergent phenomenon is either observed by an external observer (weak emergence), or by the agents themselves (strong emergence), provided that they have the knowledge to describe it. ABM provide a basis to observe emergent phenomena, as the behaviour of the modelled system can be traced during the execution. In this sub-section, we describe a few examples of higher-order, emergent phenomena modelled with ABM.

Cellular automata are one of the most widely used tools to study macroscopic patterns that emerge from the discrete, microscopic interactions in a 2D or 3D lattice [21]. While continuous models (e.g. differential equations) fail to capture the essentials of certain problems like self-reproduction of cells, such phenomena can be studied when modelled as cellular automata [22]. Although each agent is restricted to interact with its local neighbours, numerous higher-order patterns have been studied using cellular automata, such as self-organization in Conway's Game of Life [23], pattern formation in biological systems [24], engineering applications [25], and medical simulations [26].

ABM can mimic the behaviour of mathematical models and at the same time they give more power to the modeller. Bonabeau states that there is not much experimental work in the field of pattern formation in spite of a large body of theoretical analysis [27]. He claims that ABM can bridge this gap, as they are amenable to experimental observations. Subsequently, he shows how to derive the equivalent agent-based representation of a reaction-diffusion model. Agents in his model perform a random walk and interact with the environment

by depositing or removing bricks at a rate calculated from the mathematical formula of a reaction-diffusion system. He takes the agent-based model beyond its original reaction-diffusion system by replacing the state-dependent variables with short and long term memories in the agents.

Generally, higher-order patterns emerge when spatial agents act as a group. For example, in crowd modelling, groups of people tend to walk together while keeping their distance from other groups [28, 29]. Social segregation [30] is another example in which different social (ethnical, racial, or religious) groups tend to avoid other groups. Studying these systems promotes tolerance and social integration [31]. In an example from biological systems – blood coagulation – the adhesion of platelet and fibrinogen molecules leads to the formation of a blood clot within a damaged blood vessel wall [32]. One may observe a clot as an emergent entity formed as the result of interactions among several other smaller entities [33].

3. Agent Formalism

In this section, we formally describe our concept of agents. A formal definition of agents helps us clarify our component-based agent architecture, e.g. how we define an agent with regards to its constituent components, the interdependency among components, etc. It further provides a basis to analyze the computational complexity of our simulations. The timing analysis is used in the next section to study the benefit of the abstraction mechanism in improving the run-time of a simulation.

We use a generic definition of agents and show how our component-based composition of an agent fits into this definition. In our framework, an agent is defined by a 4-tuple:

$$agent = (\mathbf{Sit}, \mathbf{Act}, \mathbf{Dat}, f) \tag{1}$$

where **Sit** is the set of situations the agent can be in, **Act** is the set of actions that it can perform, **Dat** is the set of value combinations for its internal data areas, and f is a decision function [34]. At any point in time an agent decides what actions to perform based on its current situation and its internal data. This decision is captured by the decision function $f : \mathbf{Sit} \times \mathbf{Dat} \rightarrow \mathbf{Act}$.

We employ the component-based approach introduced in *LINDSAY Composer* [35, 36] to construct the agents in our simulations. A component, $comp^i$, is a mini-agent that can be combined with other components to create agents with aggregate functionalities:

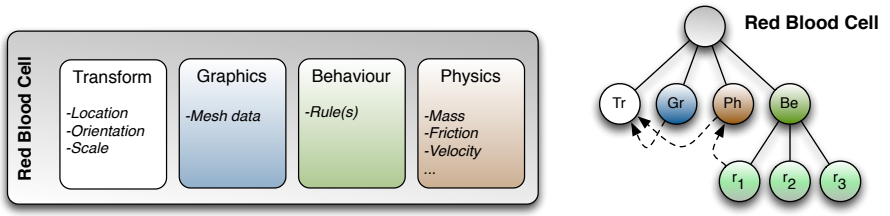
$$comp^i = (\mathbf{Sit}^i, \mathbf{Act}^i, \mathbf{Dat}^i, f^i) \tag{2}$$

With this in mind, an agent is re-defined as the composition of several components:

$$agent = \langle comp^1, comp^2, \cdots \rangle \tag{3}$$

$$\mathbf{Sit}^{agent} \subseteq \mathbf{Sit}^1 \times \mathbf{Sit}^2 \times \cdots \tag{4}$$

$$\mathbf{Act}^{agent} \subseteq \mathbf{Act}^1 \times \mathbf{Act}^2 \times \cdots \tag{5}$$



(a) (b)

Figure 1: An example of a red blood cell agent which is composed of transform, graphics, physics, and behaviour components. (a) The internal data in each component, (b) The interdependency of the sibling components in the hierarchy is denoted by dashed lines. Tr, Gr, Ph, and Be stand for the transform, graphics, physics, and behaviour components, respectively.

$$\mathbf{Dat}^{agent} \subseteq \mathbf{Dat}^1 \times \mathbf{Dat}^2 \times \cdots \tag{6}$$

$$f^{agent} = \langle f^1, f^2, \cdots \rangle = \langle f^1 : \mathbf{Sit}^1 \times \mathbf{Dat}^1 \rightarrow act^1, f^2 : \mathbf{Sit}^2 \times \mathbf{Dat}^2 \rightarrow act^2, \cdots \rangle \tag{7}$$

where $\mathbf{Sit}^{agent}$ is a subset of all the combinations of $\mathbf{Sit}^i$ in the components of an agent. While $\mathbf{Act}^{agent}$ and $\mathbf{Dat}^{agent}$ are defined similar to $\mathbf{Sit}^{agent}$, $\mathbf{f}^{agent}$ is defined as a vector of all the decision functions. In other words, all the actions chosen by the individual decision functions will be executed by the agent. It should be noted that there is no internal conflict resolution between conflicting actions. It is up to the system builder to avoid composing conflicting actions.

Behaviour components use a rule-based architecture in which **Dat** is re-written as **Intvar** $\times$ **RS**, where **Intvar** is a set of values for internal variables and **RS** is a set of interaction rules, as defined in Equation (8).

$$\mathbf{RS} = \{(r_1, ..., r_k) \mid r_i : \textbf{if } \texttt{condition}_i \textbf{ then } \texttt{execute } act_i\} \tag{8}$$

where $act_i \in \mathbf{Act}$, and condition_i is a statement about the situation the agent is in and the actual values of the variables in **Intvar**.

For example, the red blood cell agent in Figure 1(a) is defined as a combination of four sibling components. (1) A transform component containing all the information necessary to represent an agent in a three-dimensional space. (2) A graphics component with the data about how to render this agent, e.g. the mesh data. (3) A physics component containing all the physical properties like the mass and friction, which enables this agent to undergo physical interactions with other physical agents. (4) A behaviour component which includes the set of interaction rules **RS**.

A component may depend on the situation or the data of another component. The dependency of a component to other components is encoded in its **Sit** and **Dat**, i.e. there might be $\mathbf{Sit}^i$, $\mathbf{Dat}^i$, or subsequently f^i in a component that

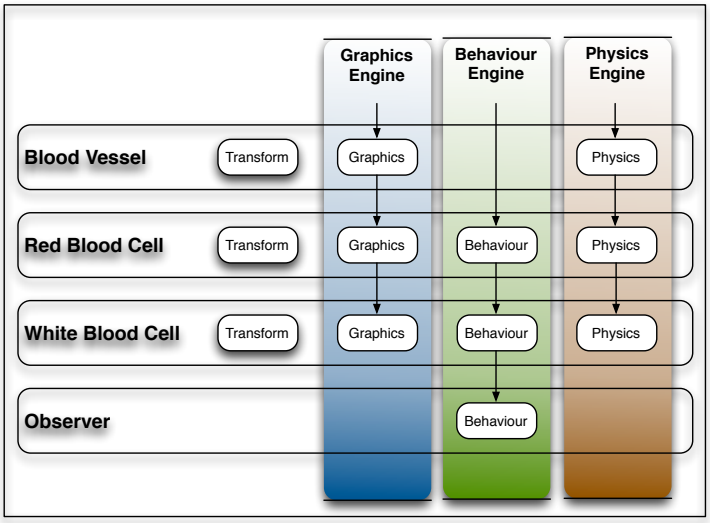


Figure 2: Three default engines in *LINDSAY Composer* drive the execution of components within agents.

looks up areas in $\mathbf{Sit}^j$ and $\mathbf{Dat}^j$ of another component. In Figure 1(b), the graphics component depends on the transform component to render a mesh while the physics component updates the same transform component once a physical force is applied to this agent. The physics component might also be used to trigger the execution of a custom rule, e.g. an action by the agent when it collides with another agent.

Components in *LINDSAY Composer* may delegate their decision functions to an engine which in turn drives their execution at each frame of the simulation. Specifically, a component may decide to share any of its $\mathbf{Sit}^i$, $\mathbf{Dat}^i$, $\mathbf{Act}^i$, or f^i with an engine. The link between a component to an engine is also encoded in $\mathbf{Dat}^i$, which specifies what engine to delegate to, along with the parameters for that engine. This design explicitly formalizes the link between components and engines. It also gives the components the freedom to share an engine or to instantiate new engines based on their needs.

LINDSAY Composer includes the following default core engines which are instantiated once the simulation starts: (1) the graphics engine which renders all the graphics components onto the screen, (2) the physics engine which handles all the physics components, and (3) the behaviour engine which executes the rules in the behaviour components. In the simple scenario of a single scale simulation, each engine iterates through its components and updates them at each frame. Figure 2 shows how each engine drives the execution of the delegating components.

The explicit formalism of components, agents, and engines in *LINDSAY Com-*

poser makes it easy to analyze the performance of the agent-based simulations. The time required to simulate an agent-based model in *LINDSAY Composer* with a single processor is defined as follows:

$$TotalTime = T_{init} + \sum_{t=1}^{T} Step^t \tag{9}$$

$$Step^t = \sum_{i} step^t(eng_i) \tag{10}$$

The initial time, T_{init} is the time spent only at the beginning of the simulation – for example to instantiate the default engines. The graphics and physics engines are usually well-optimized, as their functionality is limited to rendering and calculating the physical forces acting upon objects. On the other hand, the behaviour engine is where every custom behaviour of an agent is executed, and is therefore the bottleneck of the simulation. As a result, we only focus on the performance of the behaviour engine:

$$Step^t = step^t(BehaviourEngine) = \sum_{i=1}^{N} BComp^{t,i} = \sum_{i=1}^{N} \sum_{j=1}^{R^i} r_j^{t,i} \tag{11}$$

where N is the number of behaviour components, $BComp^{t,i}$ is the i^{th} behaviour component and $r_j^{t,i}$ is the j^{th} interaction rule in $BComp^{t,i}$ at time step t.

Without loss of generality, one can assume that all behaviour components have the same number of interaction rules, i.e. $R = \max(R^i)$. The asymptotic complexity of simulating the behaviour engine at each time step is calculated as follows:

$$O(Step^t) = \sum_{i=1}^{N} O(BComp^{t,i}) = \sum_{i=1}^{N} \sum_{j=1}^{R} O(r_j^{t,i}) = \sum_{i=1}^{N} \sum_{j=1}^{R} O(1) = N * R \tag{12}$$

where R is the number of interaction rules in each agent.

Equation (12) is a lower bound of $O(Step^t)$ since it assumes that the computational complexity of executing each interaction rule is $O(1)$. This assumption is correct when it takes $O(1)$ for an agent to check the condition of a rule. In other cases, agents might need to iterate over every other agent in the simulation to check whether the condition part of a rule holds or not, hence Equation (12) changes to $O(Step^t) = N^2 * R$. This formula clearly shows that the run-time of a simulation mainly depends on (a) the number of agents in the simulation, and (b) the number of interaction rules for each agent. It can also be inferred that simulating all the behaviours for all the agents requires a great deal of computational resources. We propose an abstraction mechanism to address this issue, which is discussed in the next section.

4. Adaptive Abstraction of Spatial Agents

The goal of the proposed abstraction is to adaptively reduce the number of agents – N in Equation (12) – during the course of the simulation. We immerse

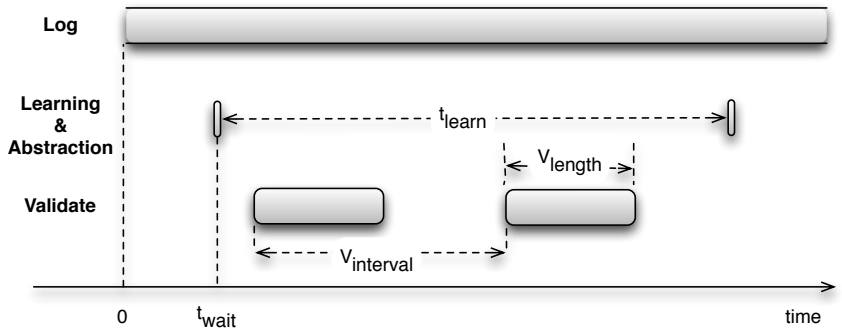


Figure 3: The three rules – **Log**, **Learning & Abstraction** and **Validate** – inside an *observer* are executed at specific time steps.

observer agents, or *observers*, in the simulation to monitor the agents and learn their adhesion patterns. *Observers* are defined the same way any other agent in the system is defined, i.e. through Equation (3) with only one behaviour component. The behaviour engine executes the rules in the *observers* at each time step during the course of the simulation.

Each *observer* has three rules in its behaviour component (Figure 3). An *observer* constantly monitors the simulation space and logs certain information about agents and their interactions. Once enough information is logged ($t > t_{wait}$), the *observer* tries to detect and learn an adhesion pattern that describes which agents have been sticking together. If this pattern is detected, the *observer* creates a meta-agent that subsumes other individuals or meta-agents. In order to validate the behaviour of their meta-agents, *observers* periodically check whether the deployment of the subsumed individual agents yields an outcome different from the predictions of the learned pattern. If the discrepancy between these two outcomes exceeds a given threshold τ_{conf}, the *observer* destroys the meta-agent and restores the subsumed individual agents. The process of learning and validating happens periodically resulting in an abstraction hierarchy that is adaptive over the course of a simulation.

Table 1 summarizes the conditions and actions for each rule. We describe each of the rules in Table 1 in the following subsections, and then we present a computation analysis of our abstraction mechanism.

4.1. The First Rule: Log

Observers are configured to maintain an adhesion graph $G_{Adhesion} = (V, E)$ whose nodes are the agents in the simulation. An edge e_{ij} between two agents denotes the strength of their adhesion, i.e. the longer two agents stick together, the larger the weight of their edge w_{ij} is. At every time step, an *observer* loops through the agents it is monitoring and updates the adhesion graph based on Algorithm 1.

Table 1: Three rules of each *observer* along with their condition and action.

Name	Condition	Action
Log	*always*	UpdateGraph: logs information about agents and their interactions
Learning & Abstraction	$(t > t_{wait})$ && $(t \mod t_{learn})$	Abstract: detect patterns and create meta-agents
Validate	$\lvert t - \frac{V_{interval}+V_{length}}{2} \rvert < V_{length}/2$	Validate: validate meta-agents

Algorithm 1 The Action: UpdateGraph($G = (V, E)$)

```
1:  for all Agent ag^i do
2:     comp^i_physics = ag^i.getDependency(PhysicsComponent);
3:     partners = comp^i_physics.collisionPartners();
4:
5:     for all Agent j in partners do
6:        w_ij ← w_ij + Δ_inc;
7:     end for
8:
9:     for all Agent j in V_i do
10:       {V_i is the set of neighbours in G for ag^i}
11:       if (w_ij > 0) && (j ∉ partners) then
12:          w_ij ← w_ij − Δ_dec;
13:       end if
14:    end for
15: end for
```

UpdateGraph is the action of this rule that updates the adhesion graph. For each monitored agent ag^i, its sibling physics component $comp^i_{physics}$ is fetched (line 2) to get the *partners* it is colliding with. Then for each collision partner j, the value of the edge between the two agents is incremented by some value Δ_{inc} (line 6). There might be other agents that were colliding with ag^i in previous time steps which are not colliding any more at this time step. Therefore, their corresponding edge should be decremented by a larger number ($\Delta_{dec} > \Delta_{inc}$) to ensure that once two agents stop colliding, their corresponding edge will be quickly set to zero. This number could also depend on the current value of an edge, but for simplicity we set it to be a constant number.

4.2. The Second Rule: Learning & Abstraction

An *observer* maintains an adhesion graph of agents. At certain intervals (t_{learn}), the *observer* finds clusters of agents that have adhered to one another for a sufficiently long duration of time, and subsequently, creates a meta-agent

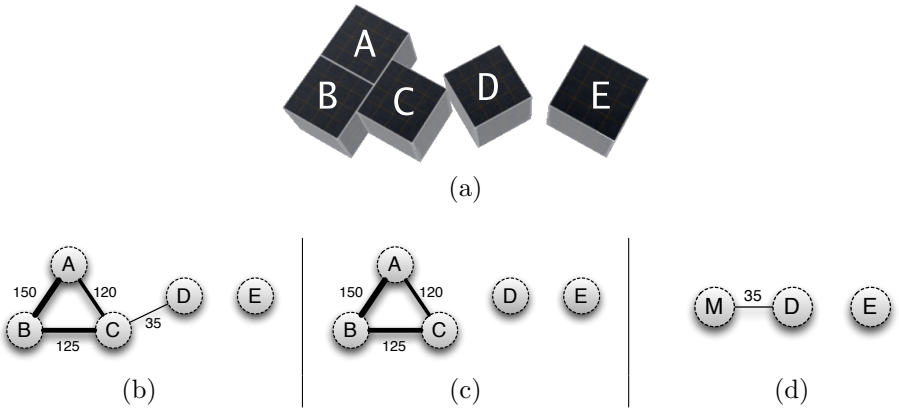


Figure 4: (a) Five agents in the simulation space. (b) The adhesion graph G maintained by the observer. (c) The modified graph G' in which weaker links ($w_{ij} < 100$) are removed. (d) The new adhesion graph G is constructed by replacing agents A, B, and C with the new abstract agent M and restoring the previous connections in the old adhesion graph.

that subsumes the individual agents in each of these clusters. Since the structure of the meta-agents is the same as that of the individual agents, the same process can be applied to meta-agents, thus creating abstraction hierarchies during the course of the simulation.

Abstract is the action of the **Learning & Abstraction** rule which is described in Algorithm 2. It first creates another graph G' by removing all the edges in the adhesion graph G whose weight is less than some threshold θ (line 1). In this new graph, an edge between two agents means that they have been sticking together for an adequately long time. In the next step, we find all the connected components in this graph. Each cluster of agents in a connected component will be subsumed by a meta-agent.

Figure 4(a) illustrates an example of learning in which there are five agents in the simulation space. Agents A, B, C, and D are colliding while agent E is detached. Figure 4(b) shows the adhesion graph of an *observer* that is monitoring this simulation. Assuming that θ is 100, the new graph G' is constructed by removing the edge between Agents C and D whose value is 35. In the next step (line 2), the connected component algorithm finds a cluster that contains more than one agent (Figure 4(c)). Subsequently, a meta-agent M is created and the adhesion graph G is updated to reflect the subsuming meta-agent (Figure 4(d)).

The components of the new meta-agent are configured as follows:

1. The behaviour component is the aggregation of all the unique rules in the subsumed agents.

2. A physics composite component encompasses individual physics components. From now on, the physics engine calculates the forces on this

Algorithm 2 The Action: Abstract(G, θ)

```
1: G' = (V, {e_ij : ∀e_ij ∈ E s.t. e_ij > θ});
2: clusters = connectedComponents(G');
3: {each cluster contains a set of agents}
4: for all Set<Agent> aCluster in clusters do
5:   if aCluster.size() ≤ 1 then
6:     continue;
7:   end if
8:   Agent meta_agent = new Agent();
9:   {composing the hierarchy of the meta_agent}
10:  TransformComponent meta_transform = new TransformComponent();
11:  PhysicsCompositeComp meta_body = new PhysicsCompositeComp();
12:  BehaviourComponent meta_behaviour = new BehaviourComponent();
13:  meta_agent.add(meta_transform);
14:  meta_agent.add(meta_body);
15:  meta_agent.add(meta_behaviour);
16:  {adding each agent in aCluster to the meta_agent}
17:  for all Agent anAgent in aCluster do
18:    meta_agent.add(anAgent);
19:    for all Component comp in anAgent do
20:      if isGraphicsComponent(comp) then
21:        continue; {nothing happens to the graphics component}
22:      end if
23:      if isPhysicsComponent(comp) then
24:        comp.active = false; {the physics component is disabled}
25:        {and attached to the composite meta_body}
26:        meta_body.attach(comp);
27:      end if
28:      if isTransformComponent(comp) then
29:        meta_transform.origin += comp.origin;
30:        comp.makeRelativeTo(meta_transform);
31:      end if
32:      if isBehaviourComponent(comp) then
33:        comp.active = false;
34:        for all Rule r in comp do
35:          if meta_behaviour.contains(r) == false then
36:            meta_behaviour.add(r);
37:          end if
38:        end for
39:      end if
40:    end for
41:  end for
42:  {meta_transform.origin is the average of all the individual origins}
43:  meta_transform.origin /= aCluster.size();
44:  {the addition of meta_agent will be reflected in G, c.r. Fig. 4(d)}
45:  update(G, meta_agent);
46:  {setting up the validation mechanism}
47:  state = waitToValidate;
48:  conf_initial = 50%;
49: end for
```

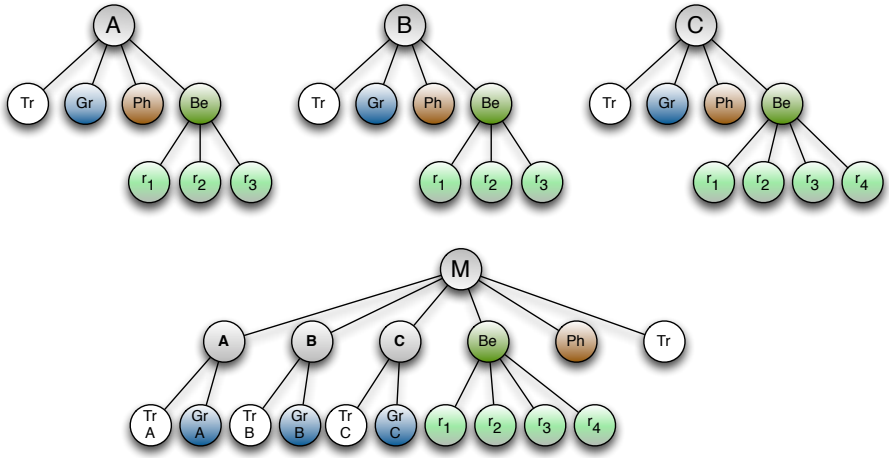


Figure 5: Three agents A, B, and C are subsumed by a meta-agent M. The meta-agent aggregates unique rules in its behaviour component resulting in a reduction of 10 individual rules to 4 rules in M.

composite structure instead of the individual structures.

3. The origin of the transform component is the average origin of all the subsumed agents. Since the graphics component depends on a single transform component as its sibling, we do not disable individual transform components. In addition, all the transform components in the subsumed agents will become relative to the meta-agent's transform component to ensure proper movement of all the sub-parts when the meta-agent moves.

Figure 5 shows the representative data structures of the three subsumed agents in Figure 4, along with the structure of the meta-agent. Since the graphics engine requires that only one transform component be present as the sibling of each graphics component, the meta-agent also maintains each parent individual agent along with its transform and graphics components. The efficiency gains are a result of aggregating the rules in the behaviour component of each individual agent into the meta-agent such that only the unique rules are added to the meta-agent.

The newly created meta-agents have a behaviour and a physics component, which enable them to undergo physical interactions as a whole, and also to execute their rules at each time step. This scale-free representation of meta-agents allows for further abstractions, as meta-agents are not any different from individual agents, and therefore they can be abstracted in the same way. For example, Figure 6 shows the next learning cycle in which agents D and M form the next meta-agent N.

An important consideration is to make sure that meta-agents show valid behaviours, as the dynamics of the system might change and individual agents

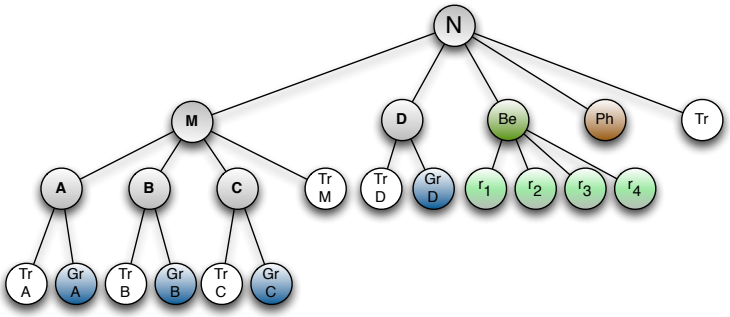


Figure 6: Assuming that agent D has the same structure as that of agent A, agents D and M form the new meta-agent N in the next learning cycle.

might not stick together any longer. In this case, a validation mechanism should be in place to ensure that the meta-agent is destroyed and the individual agents are returned to the simulation. To this end, the *observer* sets up the starting state for the validation phase at the end of Algorithm 2. Also, it assigns an unbiased confidence value ($conf_{\text{initial}} = 50\%$) to the learned hierarchy (line 48). The validation mechanism is discussed in the next section.

4.3. The Third Rule: Validate

After some time, a learned hierarchy might not be valid any more. The *observer* needs to ensure that a learned hierarchy is valid to be simulated. As the abstraction is an online process taking place as the simulation proceeds, there is no future expected data to conduct a supervised validation algorithm. As a result, the *observer* has to rely on unsupervised measures to validate a learned pattern. One such measure is the discrepancy between the outcome of the expected behaviour and the deployed behaviour of a learned hierarchy.

Algorithm 3 shows the proposed validation mechanism. At regular time intervals V_{length}, the *observer* releases a subset of the abstracted agents back into the simulation (line 6). After the validation period V_{length}, the *observer* re-abstracts those test agents (line 13) and regulates the confidence value (line 16) based on the difference between the behaviour of the test agents compared to the behaviour expected by the *observer*. In particular, if the individual test agents stick together in the validation period, the confidence value will be increased. A confidence measure below a given threshold indicates that a learned hierarchy is not valid any longer and that the *observer* has to break the learned abstraction by releasing its subsumed agents into the simulation (line 20).

4.4. Computational Analysis of the Proposed Abstraction Mechainsm

The introduction of *observers* adds an overhead to the run-time of the simulation. On the other hand, a successful abstraction should reduce the run-time shown in Equation (12). Therefore, an analysis to identify the parameters of

Algorithm 3 The validation algorithm

1: {t is the simulation time step}
2: **if** ($state == waitToValidate$) && ($t \mod V_{interval} == 0$) **then**
3: $state = validating$;
4:
5: {undo the abstraction for a subset of abstracted agents}
6: $testAgents$ = undoAbstraction(V_{ratio});
7: **end if**
8:
9: **if** ($state == validating$) && ($t \mod (V_{interval} + V_{length}) == 0$) **then**
10: $state = waitToValidate$;
11:
12: {re-abstract test individuals}
13: reAbstract($testAgents$);
14:
15: {regulate the confidence value based on the performance of individuals against what was expected}
16: $conf$ = regulate();
17:
18: {if the confidence is less than a threshold, break down the learned abstraction}
19: **if** $conf < \tau_{conf}$ **then**
20: breakAbstraction();
21: $state = noValidation$;
22: **end if**
23:
24: **end if**

the abstraction is necessary. To this end, we define the run-time of a simulation in the presence of an *observer* as follows:

$$Step^t = N' * R + Step^t(log) + Step^t(learn) + Step^t(validate) \tag{13}$$

$$N' = \alpha N + M \tag{14}$$

where N' is the number of agents in the simulation, α is the percentage of the unsubsumed agents and M is the number of meta-agents. Ideally, we want to abstract as many agents as possible ($\alpha \to 0\%$) into a single meta-agent ($M = 1$).

Calculating the timing for the first rule is straight-forward. Since each individual agent has a bounded number of collision partners, the inner loops in Algorithm 1 are executed in constant time and therefore, $Step^t(log)$ is equal to the number of agents in the system, i.e. $Step^t(log) = N'$.

The performance of the second rule depends on how the adhesion graph is implemented. Generally, finding connected components in a graph $G = (V, E)$ requires $O(|V|+|E|)$ where $|V|$ is the number of nodes in the graph, i.e. N' in the adhesion graph. Assuming that the addition of the unique rules in the behaviour

component of a meta agent requires $O(R)$, the time required to execute the second rule is calculated as follows:

$$Step^t(learn) = \begin{cases} N' + |E| + N' * R & \text{if } (t \mod t_{learn}) == 0 \\ 0 & otherwise \end{cases} \quad (15)$$

where $|E|$ is the number of the edges in the adhesion graph and R is the maximum number of interaction rules in a behaviour component.

The last rule – **validate** – simply involves monitoring a subset of subsumed agents in meta-agents and requires the following time:

$$Step^t(validate) = \begin{cases} V_{ratio}(1-\alpha)N & \text{if } |t - \frac{V_{interval}+V_{length}}{2}| < V_{length}/2 \\ 0 & otherwise \end{cases} \quad (16)$$

where V_{ratio} is the percentage of the subsumed agents whose original structure is restored in the validation cycle.

5. Experiments

The proposed self-organized learning and abstraction method can be employed in any agent-based simulation in which individual agents form groups of agents by sticking together spatially. Biological simulations are particularly suitable applications as biological entities are formed from the aggregation of smaller entities. We applied our proposed method to an agent-based simulation of blood coagulation described in the next subsection.

5.1. Model Setup

Blood coagulates at wound sites because of the interplay of various bio-agents such as platelets, fibrinogens, and serotonins. If a collagen protein around the wound site collides with a platelet, the platelet becomes activated. In case that an activated platelet collides with the wound site, it secretes several chemicals which in turn activate more platelets in the blood vessel. Gradually, a network of fibrinogens together with a platelet plug form a clot around the wound site (Fig. 7).

We identified eleven agents for this simulation, as listed in Table 2. Figure 8 shows the initial setup of the agents that exist at $t = 0$. The emitter agent produces platelets and fibrinogens and randomly positions them in a small volume at the right side of the blood vessel. A horizontal flow field moves all the platelets and fibrinogens along the blood vessel. There is a vertical flow field that pushes the agents to exit through the wound hole. Consequently, the agents exit the blood vessel either through the wound or once they reach the end of the blood vessel. Once the agents exit the blood vessel, they are no longer needed in the simulation and removed by the two destructors at both exits.

Most of the agents in Table 2 have a behaviour component consisting of a set of rules. Agents can share some rules while at the same time having their own unique rules. Although thrombin and serotonin agents, and also red and white

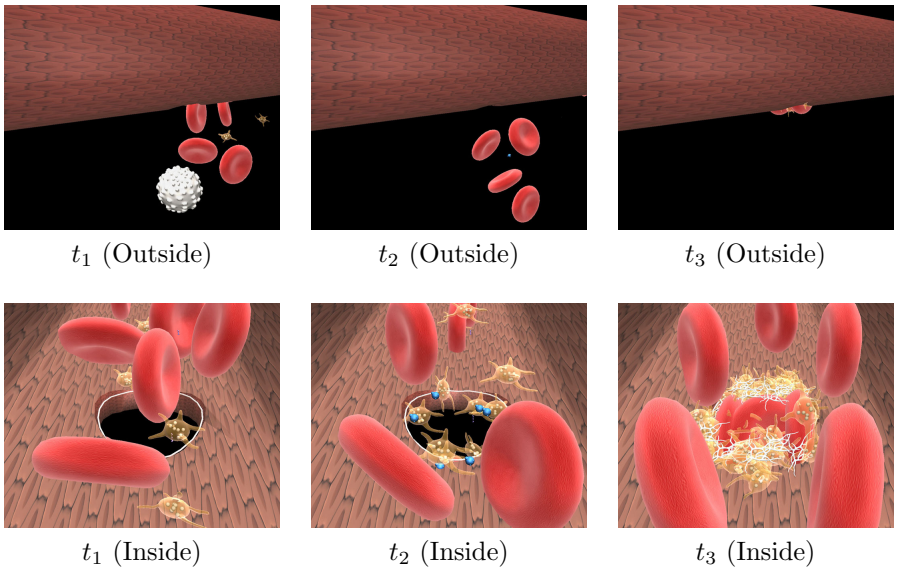


Figure 7: The blood coagulation simulation at different time steps ($t_1 < t_2 < t_3$). The process is observed from two different perspectives: inside and outside of the vessel.

Table 2: Agent description

Agent Types	**Description**
Platelet, Fibrinogen, Serotonin, and Thrombin	Their interaction results in the formation of the clot.
Red and White blood cells	They participate in the formation of the clot by getting stuck in the wound site.
Destructor	Removes the agents it is colliding with from the simulation.
Emitter	Adds new agents into the simulation space and positions them randomly in a pre-defined volume.
Flow field	Applies a fluid flow force onto the agents thus moving them along a given direction.
Blood Vessel	Defines a volume in which the flow fields move other agents.
Wound	The wound site that interacts with platelets.

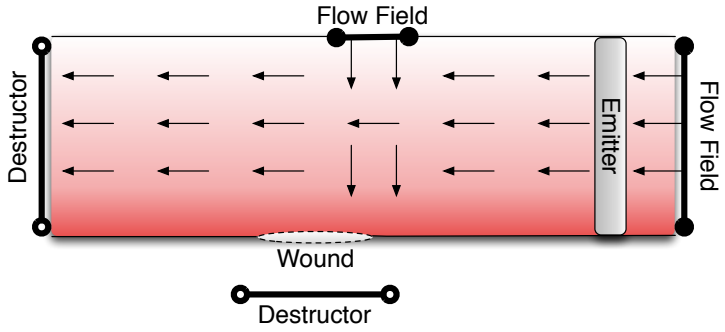


Figure 8: The simulation state at $t = 0$, the emitter agent produces platelets and fibrinogens which are moved by the flow fields inside the blood vessel. There is a hole in the wound site through which some agents exit the blood vessel. Two destructors remove agents that are not needed any longer.

blood cell agents share the same behaviour rules, they collide with other agents in different ways, since their physical structures are different. In the following, we describe the rules for each agent:

Platelet

r_1: **Self Activation**
if (agent is deactivated) **AND** (agent is colliding with either an activated platelet or the wound)
then activate the agent

r_2: **Fibrinogen Activation**
if (agent is activated) **AND** (agent is colliding with a deactivated fibrinogen)
then activate the colliding fibrinogen

r_3: **Adhesion-1**
if ($mass > 0$) **AND** (agent is activated) **AND** (agent is colliding the wound)
then set $mass$ to 0

r_4: **Adhesion-2**
if ($mass > 0$) **AND** (agent is activated) **AND** (agent is colliding with an activated platelet or an activated fibrinogen)
then set $mass$ to 0

r_5: **Secretion**
if (agent is activated) **AND** ($rand() > 3\%$)
then secrete randomly a new thrombin or serotonin

r_6: **Random Walk**
if (TRUE) **then** random walk in the space

Fibrinogen

r_1: **Self Activation**: same as r_1 in Platelet

r_2: **Adhesion-1**: same as r_3 in Platelet

r_3: **Adhesion-2**: same as r_4 in Platelet

r_4: **Random Walk**: same as r_6 in Platelet

Thrombin and Serotonin

r_1: **Chase**
if `(TRUE)` **then** `accelerate toward a randomly selected, deactivated platelet`

r_2: **Random Walk** same as r_6 in Platelet

Destructor

r_1: **Destruct**
if `(agent is colliding with another agent)` **then** `remove the colliding agent from the simulation`

Emitter

r_1: **Generate**
if $(t \bmod 15)$ **then** `generate 2 platelet and 2 fibrinogen agents with random positions`

Flow field

r_1: **Move**
if `(TRUE)` **then** `apply a physical force on all the agents inside the given volume`

Red and White Blood Cell

r_1: **Random Walk**: same as r_6 in Platelet

We ran the simulation for 1500 time steps 10 times. Each simulation started with 3 agents and ended with nearly 180 agents. Figure 9 shows the run-time of the three engines in the simulation. It confirms our previous claim that the behaviour engine is the bottleneck of the simulation. The physics and the graphics engine have a constant run-time independent of the number of the agents while the run-time of the behaviour engine grows approximately linearly with the number of agents. The linear growth of the run-time of the behaviour engine stems from the fact that none of the rules actually search in the list of the agents, hence it follows the asymptotic complexity of $O(N * R)$, as explained in Section 3.

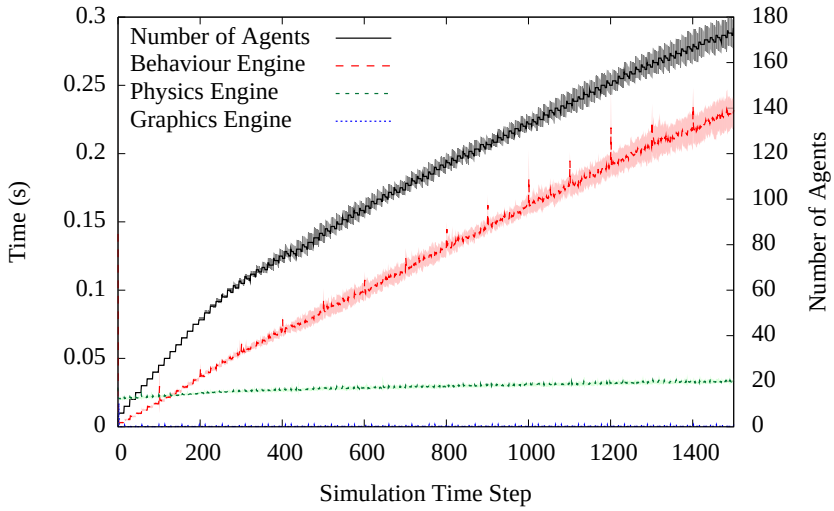


Figure 9: Run-times of the three engines along with the number of agents per simulation time step. The run-time of the graphics engine is almost at zero.

5.2. Observer Setup

In addition to the individual agents, we add one *observer* to the simulation. Table 3 lists all the important parameters in our system. The *observer* monitors the simulation space and updates the adhesion graph based on Algorithm 1. After the *observer* monitors the simulation long enough (t_{wait}), at specific intervals (t_{learn}) it finds the connected components and subsequently, creates the meta-agents. The meta-agents subsume the individual agents according to Algorithm 2. In predefined intervals, $V_{interval}$, the *observer* randomly chooses a subset of the subsumed individuals in every meta-agent and restores their original hierarchy. The size of this subset is determined by V_{ratio}. After some time, V_{length}, the *observer* puts the individual agents back in the subsuming meta-agent and validates its abstractions based on the resulting interactions compared to the expected result. The confidence of the learned pattern is regulated accordingly. If the confidence of a pattern is less than some threshold τ_{conf}, the according meta-agent will be removed and its subsumed agents will be put back in the simulation.

5.3. Results

Figure 10(a) shows the run-time of the behaviour engine in the presence of the abstraction mechanism. Compared to the normal run of the simulation (Figure 9) in which there are almost 180 agents at the end of the simulation, the abstraction mechanism reduces this number to 120. Speeding up the simulation is the immediate result of this abstraction. The larger peaks in Figure 10(a) denote the learning intervals (t_{learn}) while the smaller peaks happen at the

Table 3: System parameters

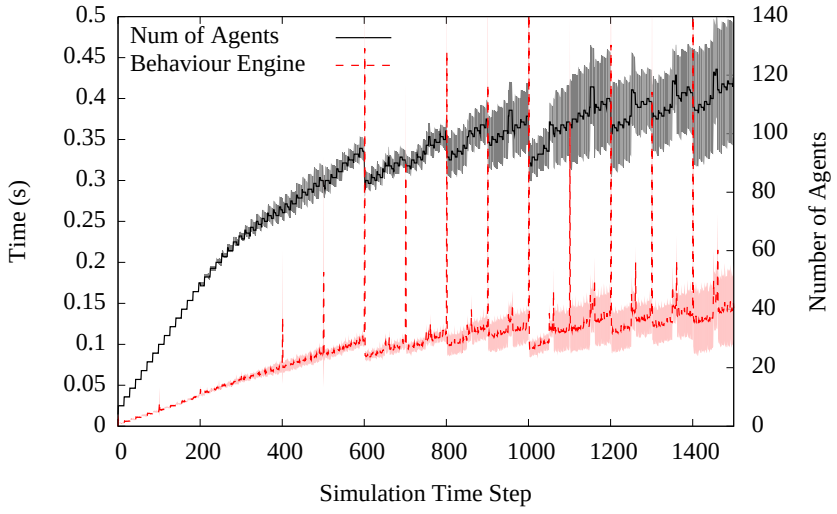
Parameter Name	Symbol	Value
Adhesion incremental weight	Δ_{inc}	1
Adhesion decremental weight	Δ_{dec}	5
Delay before learning	t_{wait}	400
Learning interval	t_{learn}	100
Cutoff threshold for the adhesion graph	θ	200
Validation interval	$V_{interval}$	50
Validation length	V_{length}	10
Validation ratio	V_{ratio}	10%
Confidence threshold	τ_{conf}	0.4

validation phase (V_{length}). Figure 10(b) depicts the cumulative run-time of the simulation comparing a normal run against a run with the *observer*. Adding the *observer* introduces no measured overhead while at the same time reducing the total run-time of the simulation from 180 seconds to 140 seconds resulting in a 20% reduction of the run-time.

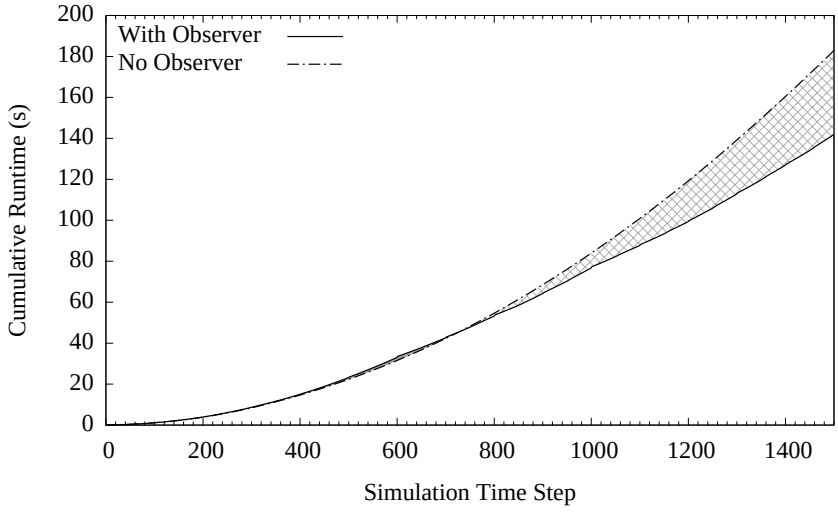
Figure 11 shows the agent adhesion graph in a sample run at $t = 900$, in which four connected components are distinctive by their colours. There are 37 agents in the biggest cluster consisting of 16 platelets, 9 fibrinogens, 9 red blood cells, and 3 meta agents. Together, they have $16 * 6 + 9 * 4 + 9 * 1 + 3 * 6 = 159$ rules. On the other hand, the resulting meta-agent will only have 6 rules, as fibrinogens and red blood cells share the same rules defined in a platelet. Therefore, creating a new meta-agent will reduce the number of rules to be checked by 153. This reduction in the number of rules is mainly responsible for speeding up the simulation.

To verify that the abstraction mechanism produces the same or a similar behaviour as that of a normal simulation, we studied how the clot is formed during the course of the simulation. The clot concentration simply measures how many platelets, fibrinogens, or red blood cells are attached to the wound. We compare the result of ten normal runs of the simulation against ten runs of the simulation with the abstraction and report the result in Figure 12. This result suggests that the choice of values for the parameters resulted in the same system behaviour while at the same time speeding up the simulation.

To further study the validation mechanism, we introduced an abrupt change in the behaviour of the simulation at $t = 1000$, when we dissolve the clot by detaching the agents from the wound. As a result, there will be almost no platelet, fibrinogen, or red blood cell attached to the wound at $t = 1200$. We undo this new change at $t = 1500$ to let the clot form again. Figure 13 compares the behaviour of our proposed abstraction mechanism with that of the original simulation. The validation mechanism ensures that the system behaviour will adapt to the changes in the simulation – turning the validation off would result



(a)



(b)

Figure 10: Run-time with and without the *observer*, (a) Run-time per simulation time step, (b) Cumulative run-time.

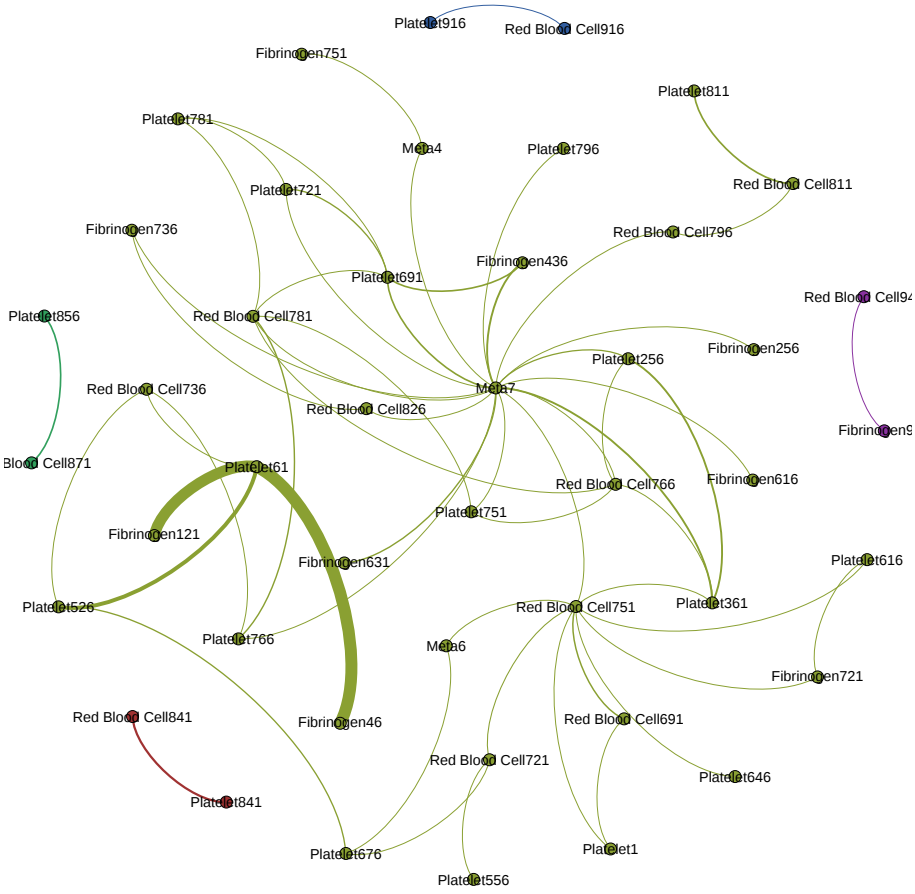


Figure 11: The agent adhesion graph in a sample run at $t = 900$ in which there are 3 clusters of connected components.

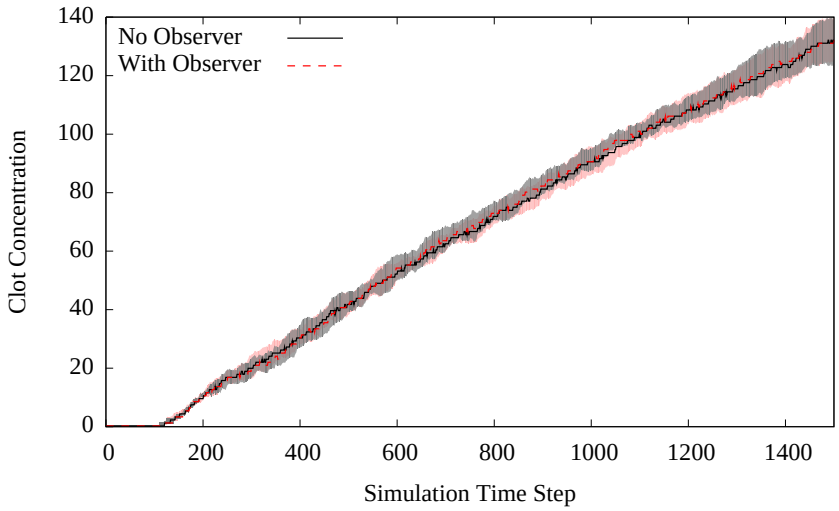


Figure 12: System behaviour in terms of the clot concentration, i.e. the number of platelets, fibrinogens, and red blood cells around the wound, reported over ten runs of the simulation with and without the proposed abstraction mechanism.

in an inaccurate system behaviour.

6. Conclusion and Future Work

We introduced the concept of abstraction to boost the speed of agent-based simulations by means of a light-weight *observer* agent that monitors the simulation space and abstracts groups of individual agents to higher-order meta-agents which in turn are subject to further abstractions. We analyzed the time required to run such agent-based models and compared it to a normal run with no abstraction.

Our proposed abstraction mechanism was applied to an agent-based simulation of blood coagulation, in which bio-agents stick together to form a clot around the wound site thus preventing further bleeding. We showed that the adaptive abstraction results in the same system behaviour but with a 20% faster run-time. We emphasized the role of our unsupervised validation algorithm to ensure the validity of meta-agents.

The abstraction mechanism creates self-organized, dynamical hierarchies during the course of a simulation. Studying the emerging patterns in such hierarchies is of great interest, particularly in the case of biological simulations in which new entities at higher levels are formed as the result of interactions among lower level entities. For multi-scale modelling, a stable, higher order entity can be used in other time or spatial scales to manage the computational burden of the simulation. This could eliminate the need for exponential increases in computation power to model such systems.

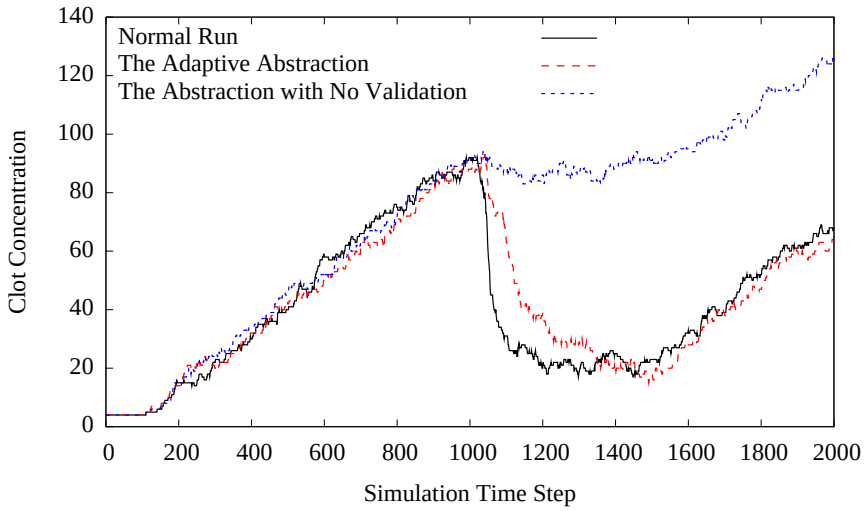


Figure 13: System behaviour in terms of the clot concentration. At $1000 < t < 1500$, the clot is forced to dissolve. An adaptive abstraction successfully follows the behaviour of the original run while an abstraction without validation results in an inaccurate system behaviour.

In this work, the *observer* agents look for a particular, fixed spatial pattern to abstract groups of individual agents, i.e. agents that stick together for a sufficiently long period of time. However, since the proposed framework is generic, other types of patterns could be explored to cover a wider range of abstraction mechanisms.

7. Acknowledgement

This work has been partially supported by the Natural Sciences and Engineering Research Council of Canada.

References

[1] D. Noble, The music of life, Oxford University Press, 2006.

[2] E. Bonabeau, Agent-based modeling: methods and techniques for simulating human systems, Proceedings of the National Academy of Sciences of the United States of America 99 Suppl 3 (2002) 7280–7287.

[3] Z. Wang, C. M. Birch, J. Sagotsky, T. S. Deisboeck, Cross-scale, cross-pathway evaluation using an agent-based non-small cell lung cancer model., Bioinformatics (Oxford, England) 25 (2009) 2389–96.

[4] S. Bandini, S. Manzoni, G. Vizzari, Multi-agent modeling of the immune system: The situated cellular agents approach, Multiagent and Grid Systems – An International Journal 3 (2007) 173–182.

[5] I. Salazar-Ciudad, Tooth Morphogenesis in vivo, in vitro, and in silico, Current Topics in Developmental Biology 81 (2008) 342.

[6] M. Scheutz, P. Schermerhorn, Adaptive algorithms for the dynamic distribution and parallel execution of agent-based models, Journal of Parallel and Distributed Computing 66 (2006) 1037–1051.

[7] M. Lysenko, R. D'Souza, A framework for megascale agent based model simulations on graphics processing units, Journal of Artificial Societies and Social Simulation 11 (2008).

[8] M. Scheffer, J. M. Baveco, D. L. DeAngelis, K. A. Rose, E. H. van Nes, Super-individuals a simple solution for modelling large populations on an individual basis, Ecological modelling 80 (1995) 161–170.

[9] D. Helbing, S. Balietti, How to Do Agent-Based Simulations in the Future: From Modeling Social Mechanisms to Emergent Phenomena and Interactive Systems Design.

[10] A. Crooks, C. Castle, M. Batty, Key challenges in agent-based modelling for geo-spatial simulation, Computers, Environment and Urban Systems 32 (2008) 417–430.

[11] D. Pawlaszczyk, S. Strassburger, Scalability in distributed simulations of agent-based models, in: Proceedings of the 2009 Winter Simulation Conference (WSC), section 3, IEEE, 2009, pp. 1189–1200.

[12] S. von Mammen, A. Sarraf Shirazi, V. Sarpe, C. Jacob, Optimization of Swarm-Based Simulations, ISRN Artificial Intelligence 2012 (2012) 1–13.

[13] A. Sarraf Shirazi, S. von Mammen, C. Jacob, Hierarchical self-organized learning in agent-based modeling of the MAPK signaling pathway, in: Evolutionary Computation (CEC), 2011 IEEE Congress on, pp. 2245–2251.

[14] A. Sarraf Shirazi, S. von Mammen, C. Jacob, Abstraction of Agent Interaction Processes: Towards Large-Scale Multi-agent Models, Simulation: Transactions of the Society for Modeling and Simulation International (in press).

[15] D. Luebke, M. Harris, N. Govindaraju, A. Lefohn, M. Houston, J. Owens, M. Segal, M. Papakipos, I. Buck, GPGPU: general-purpose computation on graphics hardware, in: SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing, ACM, New York, NY, USA, 2006, p. 208.

[16] A. R. Stage, N. L. Crookston, R. A. Monserud, An aggregation algorithm for increasing the efficiency of population models, Ecological modelling 68 (1993) 257–271.

[17] H. Abdi, L. J. Williams, Principal component analysis, Wiley Interdisciplinary Reviews: Computational Statistics 2 (2010) 433–459.

[18] S. Wendel, C. Dibble, Dynamic agent compression, Journal of Artificial Societies and Social Simulation 10 (2007).

[19] J.-L. Dessalles, D. Phan, Emergence in multi-agent systems: cognitive hierarchy, detection, and complexity reduction part I: methodological issues, Artificial Economics 564 (2006) 147–159.

[20] J.-P. Müller, Emergence of Collective Behaviour and Problem Solving, in: A. Omicini, P. Petta, J. Pitt (Eds.), Engineering Societies in the Agents World IV, volume 3071 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, 2004.

[21] S. Wolfram, Statistical mechanics of cellular automata, Reviews of Modern Physics 55 (1983) 601.

[22] A. Deutsch, S. Dormann, Introduction and Outline, in: Cellular Automaton Modeling of Biological Pattern Formation, Modeling and Simulation in Science, Engineering and Technology, Birkhäuser Boston, 2005, pp. 3–11.

[23] M. Gardner, Mathematical Games: The fantastic combinations of John Conway's new solitaire game "life", Scientific American (1970) 120–123.

[24] A. Deutsch, S. Dormann, Cellular Automaton Modeling of Biological Pattern Formation, Birkhäuser Boston, 2005.

[25] X.-S. Yang, Y. Young, Cellular Automata, PDEs, and Pattern Formation, Handbook of Bioinspired Algorithms and Applications (2010) 12.

[26] R. M. Amorim, R. S. Campos, M. Lobosco, C. Jacob, R. W. dos Santos, An Electro-Mechanical Cardiac Simulator Based on Cellular Automata and Mass-Spring Models., in: G. C. Sirakoulis, S. Bandini (Eds.), ACRI, volume 7495 of *Lecture Notes in Computer Science*, Springer, 2012, pp. 434–443.

[27] E. Bonabeau, From classical models of morphogenesis to agent-based models of pattern formation., Artificial life 3 (1997) 191–211.

[28] G. Vizzari, L. Manenti, An agent-based model for pedestrian and group dynamics: experimental and real-world scenarios, in: Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems - Volume 3, AAMAS '12, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 2012, pp. 1341–1342.

[29] M. Batty, Agent-based pedestrian modelling, Advanced spatial analysis: the CASA book of GIS (2003) 81.

[30] T. C. Schelling, Dynamic Models of Segregation, Journal of Mathematical Sociology 1 (1971) 143–186.

[31] M. Abdou, N. Gilbert, K. Tyler, Agent-Based Simulation Model for Social and Workplace Segregation, in: Proceedings of the 8 Annual Conference of European Social Simulation Assoc., Brescia, pp. 1–12.

[32] B. Dahlbäck, Blood coagulation, Lancet 355 (2000) 1627–32.

[33] A. S. Shirazi, S. von Mammen, I. Yazdanbod, C. Jacob, Self-Organized Learning of Collective Behaviours in Agent-Based Simulations, in: H. Sayama, A. A. Minai, D. Braha, Y. Bar-Yam (Eds.), ICCS 2011 - 8th International Conference on Complex Systems, NECSI Knowledge Press, NECSI Knowledge Press, Boston, MA, USA, 2011.

[34] J. Denzinger, J. Hamdan, Improving observation-based modeling of other agents using tentative stereotyping and compactification through kd-tree structuring, Web Intelligence and Agent Systems 4 (2006) 255–270.

[35] S. von Mammen, T. Davison, H. Baghi, C. Jacob, Component-based networking for simulations in medical education, in: Computers and Communications (ISCC), 2010 IEEE Symposium on, IEEE, 2010, pp. 975–979.

[36] C. Jacob, S. von Mammen, T. Davison, A. Sarraf Shirazi, V. Sarpe, A. Esmaeili, D. Phillips, I. Yazdanbod, S. Novakowski, S. Steil, C. Gingras, H. Jamniczky, B. Hallgrimsson, B. Wright, LINDSAY Virtual Human: Multi-scale, Agent-based, and Interactive, in: J. Koodziej, S. U. Khan, T. Burczynski (Eds.), Advances in Intelligent Modelling and Simulation, volume 422 of *Studies in Computational Intelligence*, Springer Berlin Heidelberg, 2012, pp. 327–349.