FernUniversität in Hagen

A generalized computational model for modeling and simulation of complex systems

Jochen Kerdels & Gabriele Peters

A generalized computational model for modeling and simulation of complex systems

Jochen Kerdels   Gabriele Peters

November 29, 2012

Abstract

The use of computer models and simulation is a widely adopted approach to study complex systems. To this end a diverse set of computational models like Cellular Automata, Artificial Neural Networks, or Agent-based simulation is being used. As a common denominator virtually all of these approaches favor different variations of complex systems and are tailored to support the description of systems that fit the corresponding variation well. Although this form of specialization has its benefits like ease of modeling with respect to the particular subset of complex systems, the drawbacks of this specialization are a lack of comparability between structurally different systems and a diminished expressiveness with respect to systems that do not fit any particular subset of complex systems favored by existing, specialized models. In this paper a generalized computational model for complex systems is proposed which allows for the description of most types of systems with a single model. Furthermore, the proposed model provides a high degree of encapsulation and reduces the amount of shared knowledge needed among the constituents of the system. The paper closes with a set of example applications of the proposed model to further illustrate the involved concepts and to provide an intuition on how this model may be used.

1   Introduction

In recent years the interest in modeling and simulation of complex systems has increased considerably among a broad variety of domains and scientific disciplines. For instance, Niazi and Hussain describe the increasing use of Agent-based modeling over the last two decades in [NH11]. Although lacking a precise technical definition the term *complex system* is generally applied to systems consisting of many interacting constituents that collectively exhibit some traits which cannot easily be deduced from specific properties of the individual constituents. Instead, these collective traits emerge from the interactions occurring within the complex system. Typical examples of such complex systems are ant colonies, economic systems, nervous systems or biological evolution [New11].

In order to study such systems the use of exploratory models is a widely adopted approach. To this end a set of diverse computational models like Cellular
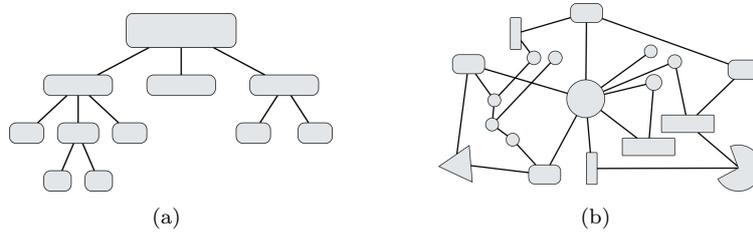
Figure 1: Complicated (a) vs. complex (b) systems.

Automata, Artificial Neural Networks, or Agent-based simulation is being used. The models focus on different variations of complex systems and are tailored to support the description of systems that fit the corresponding variation well. For example, Cellular Automata aim at the description of complex systems that consist of vast numbers of *cells* which are arranged in a regular topology and which act according to very few, simple rules. In contrast to this, Agent-based models focus on fewer, more elaborate constituents, i.e. agents, that can roam freely in a typically two- or three-dimensional environment.

On the one hand, the use of specialized models allows for the intuitive description of a complex system if this system fits the particular characteristics favored by the model. Additionally, the intuitive mapping of a complex system to such a model can ease the interpretation of results gained through simulation with respect to the original complex system. On the other hand, the results obtained by a specialized computational model may not be readily generalized and applied to other complex systems that do not fit the characteristics of the particular model. Furthermore, the use of specialized models that favor or even "demand" certain characteristics of a complex system bears the risk of *Maslow's hammer*[1], i.e. it might be tempting to fit the complex system onto the characteristics of the model instead the other way around. In order to lessen these drawbacks, it is desirable to have a generalized computational model for complex systems that minimizes the requirements on specific characteristics of the particular complex system and thus maximizes the diversity and number of systems that can be analyzed within one model.

This paper proposes a computational model for complex systems that is designed to be as general and reduced as possible while maintaining the ability to allow for an intuitive modeling of the particular complex system at hand.

2 The difficulty of modeling complex systems

The analysis, modeling and simulation of complex systems is generally acknowledged as a hard problem. The main argument in this regard is based upon the distinction between *complicated* and *complex* systems. While both types of systems can consist of many interacting constituents, the former type has a

---

[1]"I suppose it is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail." – Maslow, 1966, [MW66]

fully known and well-formed structure, e.g., in the form of a hierarchy (Fig. 1a), whereas the latter type is characterized as a collection of heterogeneous, highly interconnected constituents that lack such a structure (Fig. 1b) and thus – according to common reasoning – can not be analyzed with otherwise successful and widespread reductionistic methods. Although this argument hints at the right direction as to why it is difficult to analyze complex systems, the conclusion drawn appears premature. The sole fact that a complex system consists of heterogeneous, highly interconnected entities does not suffice to make this system hard to analyze. According to the given description a complex system would be nothing more than some form of *complicated* graph – a structure for which a rich set of tools exists in computer science and mathematics.

In order to shed some light on the root causes as to why complex systems are difficult to analyze and model it is worthwhile to investigate two central characteristics of complex systems. The first one regards the change of relations in time between the constituents of a complex system. The second one regards the superposition of constituents based on the particular aspects under which a complex system is analyzed.

Change of relations

In a complex system it is likely that the relations between its constituents change considerably over time. In contrast to complicated systems, where such changes occur only rarely or not at all, it is an important and distinctive feature of complex systems. A good example for this kind of change can be observed in neural networks where the relations between different neurons change constantly in response to the activation of the involved neurons. In Artificial Neural Networks these changing relations between neurons are commonly modeled as weighted connections whose weights vary in course of a particular learning process. As the actual weights of the connections are only determined during runtime based on the input data, it is impossible to predict the connection weights in advance, i.e. during design-time, and thus it is not possible to exactly specifiy which neuron will contribute to which output of the system. As a consequence, it is very difficult among other things to determine how many neurons arranged in which configuration are sufficient to produce the desired behavior of the network.

A slightly different kind of changing relations between constituents of a complex system can be described by Agent-based models. In this case the relations between the constituents, i.e. agents, are induced by a shared environment. As the agents move, their relations, e.g. their relative positions to each other, change over time. In contrast to Artificial Neural Networks the relations between the constituents of an Agent-based model are not explicitly modeled. Instead, they are implicitly defined by the particular environment of the model. On the one hand this implicit definition of relations enables the intuitiv modeling of a complex system by translating the perceived structure of the system almost one to one into the structure of the Agent-based model. On the other hand this approach may obfuscate important relations which should be stated explicitly in order to isolate and precisely characterize them.

These examples illustrate that it is not the sheer amount of connections between the entities of a complex system which cause the system to be difficult to analyze

and model. It is the fact, that those connections are only *potential* connections that can vary over time. Computational models like Artificial Neural Networks or Agent-based models focus on exactly this characteristic of complex systems by providing mechanisms and structural elements which allow to model these potentially varying relations.

Superposition of constituents

The second characteristic of complex systems regarding the difficulty of analyzing and modeling is more subtle and it relates to the way we humans perceive and think. Our main mental strategy to cope with arbitrarily complex matters is the use of *categorisation*. Typically we begin by taking (perceived) continua and breaking them down into manageable categories. For example, if we want to address the wavelength emitted by an object we see, we break the continuous visible spectrum of wavelengths down into a small number of categories, i.e. colors. When matters get more complex, we aggregate categories into more abstract super-categories. As this process continues we build up ever more abstract categories. If such a higher-level category is contemplated, the constituting lower-level sub-categories become increasingly less present in working memory, thus effectively hiding the full complexity of the higher-level category and keeping the *perceived* complexity approximately constant. This strategy enables us to think about arbitrarily complex matters without the need to actually address the full complexity of the particular matter at once.

In fact, even if we would want to address the full complexity of some matter at once, we would not be able to do so. As soon as we start to explore some aspect of a higher-level category by recursively breaking down the category into its lower-level sub-categories we are bound to a limited number of items that we can store in our working memory at once. Thus, while dealing with some specific detail of a higher-order category we temporarily lose sight of the "big picture". When dealing with complicated systems like the one depicted in figure 1a, this limitation poses no serious problem to the analysis and modeling since the different sub-parts of a complicated system are cleanly separated from each other. For each sub-part of the system it is sufficient to define a proper interface that hides the internal structure and allows for "black boxing" the particular sub-part.

In contrast, when dealing with complex systems the approach of piece by piece analysis and modeling of the system is not without problems as the assumption of cleanly separated sub-parts is usually not true for a complex system. As a result, the previously decribed strategy of "black boxing" which corresponds well to the way we mentally deal with complex matters does not fit the structure of complex systems. Instead, the categories involved in complex systems frequently share common sub-categories that give rise to potentially unanticipated, mutual influences. These potential side-effects can be difficult to notice and track since the involved sub-categories may not be readily present in the working memory of the person that is performing the analysis. As described before, our mental strategy for dealing with complex matters hides the full complexity of higher-level categories by keeping the lower-level sub-categories out of working memory. Hence, lower-level sub-categories that are shared between different parts of the
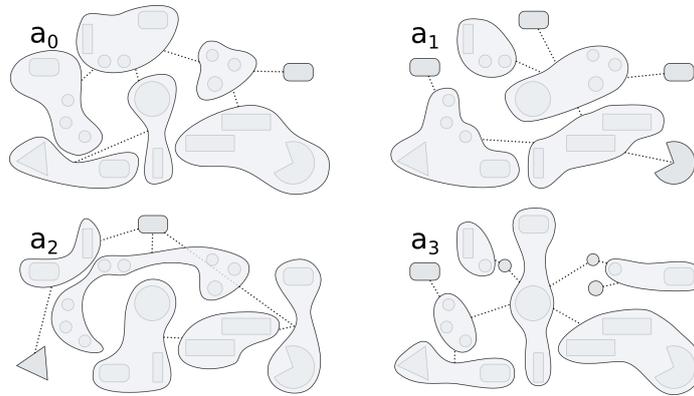
Figure 2: Constituents of a hypothetical complex system with respect to four different aspects of that system.
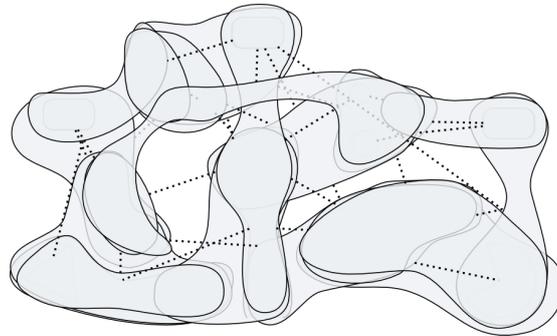

Figure 3: The resulting superposition of constituents that correspond to different aspects of the system when viewed as a single model.

complex system can easily be overlooked. For example, it is a practical necessity to stop the analysis and modeling of some aspect of a system at a level of abstraction which appears to approximate the particular aspect reasonably well. If it then happens that a shared sub-category "hides" some levels below that chosen level of abstraction, a crucial part of mutual influence in the system may be missed in the resulting model. The mere possibility of shared sub-categories between different parts of a complex system makes it much more difficult – compared to a complicated system – to determine if the analysis and modeling of each part has reached a sufficiently detailed level.

Furthermore, the intricate relations of higher- and lower-level categories attributed to a complex system affect the constituents of the corresponding model since the constituents are predominantly deduced from these categories. As a result, constituents of the model that reflect certain aspects of the complex system inherit some of the intricate relations from the underlying categories, i.e. they overlap to a certain extent where the underlying categories share common lower-level subcategories. Figures 2 and 3 illustrate this by showing the constituents of a hypothetical complex system with respect to four different aspects and their resulting superposition when the constituents corresponding to

the different aspects are merged into a single model of the complex system. The mutual influences of shared lower-level categories hint at potential side-effects between overlapping constituents that have to be considered in order to model the paricular complex system correctly.

The foregoing considerations as to why complex systems are difficult to analyze and model can be summarized as follows:

- The relations between the constituents of a complex system can vary over time in non-trivial ways and thus cannot be specified in advance. The particular computational model which is being used has to provide structures and mechanisms that allow for a determination of these relations at runtime.

- Mutual influences between constituents of a complex system are not only represented by higher-level categories as in complicated systems but are also indicated by shared, lower-level categories. As the mental analysis of a complex matter typically progresses from higher-level categories towards lower-level categories while simultaneously only a limited number of items can be kept in working memory at once, it is difficult to reliably identify the mutual influences occurring between the constituents of a complex system.

- If a complex system is analyzed with respect to different aspects, the resulting constituents for each aspect can "conceptually overlap" when they are merged into a single model of the system. These overlaps hint at potential side-effects between the involved constituents and suggest that a further fragmentation of the corresponding constituents may be necessary in order to transform the implicit side effects into explicit relations.

Existing computational models for complex systems focus predominantly on the first point by providing structures and mechanisms to deal with dynamic relations between constituents. The main aspect involving the latter two points, i.e. the granularity and identification of suitable constituents, is usually not handled explicitly. Instead, it is implicitly predefined by the characteristics of the specific variation of complex systems that is favored by the particular model. In order to treat this aspect explicitly, a more general computational model may be beneficial.

3 A generalized model for complex systems

Virtually all present-day computational models for complex systems focus on different variations of complex systems and are tailored to support the description of systems that fit the corresponding variation well. While such a restriction can be quite useful regarding the intuitive mapping of the complex system to the model, it bears also a number of drawbacks. As described in the previous section the identification of suitable constituents that reduce the amount of implicit side effects can be difficult. If a given model favors a certain kind of constituents, this preference can interfere with the already difficult selection process and lead

to a poor choice of constituents, i.e. the complex system is fittet onto the model instead of the opposite. Furthermore, the results obtained by a specialized model may not be readily generalized and applied to other complex systems that do not fit the characteristics of the particular model.

A generalized computational model could offer the flexibility needed to address the aforementioned drawbacks. The main challenge for such a generalized model would be to provide a *common theme* that allows for an intuitive interpretation of the model and the corresponding system while being general enough to encompass all or at least most types of complex systems. As it turns out, an appropriate common theme can be derived from a similar modeling problem in physics. There, one of the most general ways to characterize a system is to describe how the energy contained in the system is distributed across and transformed by its constituents. For example, if a stone is lifted up, the energy required to do the lifting is transfered from the lifter to the stone and stored as *potential energy.* Although this example is as simple as it gets, it illustrates an important insight: The seemingly passive stone is an active constituent of the system. It can "store" and "release" energy by means of its location, mass and momentum.

Analogous to this approach a complex system can be characterized by describing how information is distributed and transformed inside the system. To this end a complex system can be described as a set of information processing *nodes* which send and receive information using messages. In contrast to the constituents of existing models, e.g. cells or agents, these nodes represent also parts of the system that would be commonly considered as passive elements. For example, if an agent in an Agent-based model can place some resource onto a patch of the model's environment, this patch – like the stone in the example above – is an active part of the complex system. In that sense, the patch can be regarded as a very simple information processing node.

The addressing problem

Based on this common theme of communicating nodes, the issue of varying relations between constituents becomes an *addressing problem*:

> How can a set of nodes exchange messages in a constantly changing system with a minimal amount of shared[2] knowledge?

A solution to this addressing problem can be infered from the following observation. Many computational models for complex systems use some form of environment to impose a set of comparable attributes on all constituents, thereby creating a means to relate the individual constituents to each other. For example, the environment in a Cellular Automaton places the cells in a grid topology resulting in inter-cell relations like 4- and 8-neighborhoods. The definition of a

---

[2]The addendum requiring that shared knowledge should be avoided if possible emphasizes that solutions to the addressing problem like *yellow pages* which are based on global knowledge are ill-suited to model complex systems as such global mechanisms do not scale well.

cell's behavior can then refer to these neighborhoods, regardless where in the grid the cell is actually placed[3].

This functional role of the environment, i.e. establishing comparable attributes among the constituents, can be generalized to serve as a key element in a solution to the addressing problem. Instead of using a globally defined environment, each node of the proposed model can exhibit a custom and potentially changing set of arbitrary properties on its own. These properties can then be used in conjunction with a property based addressing scheme. This scheme determines the recipient nodes of a message by evaluating a set expression composed of sets of nodes, common set operators and special set operators called *property filters*.

A property filter

$$\Phi : C_{in} \rightarrow C_{out} \quad with \quad C_{out} \subseteq C_{in}$$

is essentially a function that takes a set of nodes C_{in} as argument and returns a subset C_{out}. Which node of the input set remains in the output set is determined by the property filter using only information provided by the properties of the nodes in the input set. For example, let K be the set of all nodes of a model. Further, let K contain nodes with properties of a type $\langle position \rangle$ and a type $\langle color \rangle$. Then, a set of all *red* nodes in the neighborhood of the position (x_s, y_s) can be expressed as the set expression

$$\left(K \circ \Phi_{\langle neighborhoodOf \rangle}(x_s, y_s) \right) \cap \left(K \circ \Phi_{\langle hasColor \rangle}(red) \right)$$

where $\circ$ denotes the application of a property filter to the left hand side of the operator. Based on this addressing scheme, a node that wants to transmit information can indirectly address a set of nodes as recipients by using a set expression as previously described.

Model component overview

The four component types *Node*, *Property*, *PropertyFilter*, and *Message* introduced so far are the basic building blocks of the proposed, generalized model. Figure 4 illustrates these four component types and their relation among each other. The functional roles of the four components can be summarized as follows:

- *Nodes* encapsulate local processing of information. They "consume" and "produce" packets of information, i.e. messages.

- *Properties* encapsulate common attributes shared among a subset of nodes. They allow to relate different nodes to each other and thus facilitate the indirect addressing of nodes as recipients of information.

- *Property filters* encapsulate semi-global operations on properties. They allow to select specific nodes as recipients of information based on an arbitrarily complex evaluation and comparison of properties.

- *Messages* encapsulate packets of information that can be send from one node to a set of recipient nodes. The contents of a message can be arbitrary.

---

[3]A popular instance of a Cellular Automaton whose cells make use of such a neighborhood relation is John Conway's *Game of Life*[Gar70].
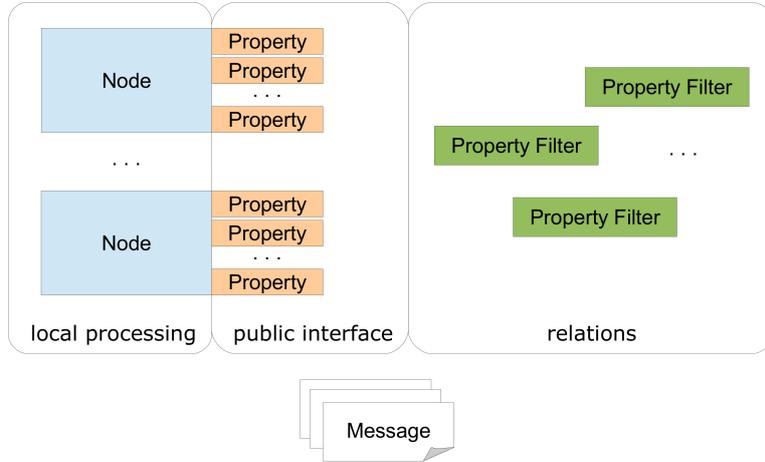
Figure 4: Overview of all components of the proposed model.

The proposed model minimizes and isolates shared knowledge. For example, if the nodes of a complex system should be related to each other by their position, only the fact that each node has a position is shared knowledge that has to be present in each node – especially if the nodes can change their position. The actual implementation of this attribute is isolated as a separate element, i.e. the property of type $\langle position \rangle$. Additionally, the way how relations are build upon this attribute is isolated in corresponding property filters, like $\Phi \langle neighborhoodOf \rangle$. As a result, the nodes themselves do not need to know how high-level concepts like $neighborhoodOf$ are implemented.

Time discretization

In the model described so far, a node may change the properties it exhibits and may send messages at any time. This approach is not practically feasible and therefore the time domain has to be discretized. Accordingly, the proposed model operates on a discrete time scale with steps $t \in \mathbb{N}$. For each time step t the model performs the following sequence of actions:

1. Determine the set of recipients for every message by evaluating the property set expression of the particular message.

2. Determine the set of incoming messages for every node.

3. Let every recipient node process its incoming messages. Each node may create new outgoing messages, create or delete other nodes and create, delete or modify its properties.

Due to this three stage approach the model effectively simulates the parallel execution of the information processing performed by the individual nodes in the third stage. Changes to the properties of a node, or the creation and deletion of nodes take effect only in the following time step.

9

Formal definition

As the proposed model has only four basic components and processes only three stages per time step, it is reasonable to provide a formal definition of the model in addition to the informal description given so far.

Basically, the model consists of three types of sets per time step t: A set of nodes

$$K^t = \{k_1, k_2, \ldots\},$$

a set of exhibited properties per node

$$P_k^t = \{p_1, p_2, \ldots\}, \quad k \in K^t,$$

and a set of messages

$$M^t = \{m_1, m_2, \ldots\}.$$

A message $m \in M^t$ is a tuple (ϕ_m, c) with a property set expression ϕ_m and the message content c.

Each node $k \in K^t$ has a message processing function

$$\delta_k : D_k \mapsto \left(\Delta^+ K_k^{t+1}, \Delta^- K_k^{t+1}, \Delta^+ P_k^{t+1}, \Delta^- P_k^{t+1}, M_k^{t+1}\right)$$

which processes all messages $m \in D_k$ and returns a tuple of five sets that describe the changes to the set of nodes, the changes to the set of exhibited properties and the outgoing messages of that node for the next time step. The returned tuple of five sets reflects the node's ability to create and delete nodes, to create, modify (*implicitly*) and delete its properties, and to create new messages.

In this formal description of the model the previously described three stages performed per time step t are extended to four stages:

- For every message $m \in M^t$ determine the set C_m of recipient nodes by evaluating the property set expression ϕ_m.

- For every node $k \in K^t$ determine the set $D_k = \{m | k \in C_m\}$ of messages that are addressed to node k.

- For every node $k \in K^t$ where $D_k \neq \emptyset$ evaluate the message processing function δ_k.

- Update the sets of nodes, properties and messages:

$$K^{t+1} = \left(K^t \cup \bigcup_{k \in K^t} \Delta^+ K_k^{t+1}\right) \setminus \bigcup_{k \in K^t} \Delta^- K_k^{t+1},$$

$$P_k^{t+1} = \left(P_k^t \cup \Delta^+ P_k^{t+1}\right) \setminus \Delta^- P_k^{t+1}, \quad k \in K,$$

$$M^{t+1} = \bigcup_{k \in K^t} M_k^{t+1}.$$

This formal definition concludes the description of the proposed computational model.

Node	Property	PropertyFilter	Message
process_messages()	get_id()	init()	address
----------------------	----------------------	filter()	----------------------
add_node()		reiterate()	
rm_node()		----------------------	
add_property()			
rm_property()			
send_message()			
ensure_call()			

Figure 5: Overview of all components of the proposed model as pseudocode classes.

4   Examples

The following sections provide examples of different complex systems modeled with the computational model introduced above. These examples are intended to illustrate the broad applicability of the proposed model and to convey some ideas on how to utilize the model.

Pseudocode introduction

The examples use a pseudo-code notation that is similar to an existing object-oriented implementation of the model. The main base classes are `Property`, `PropertyFilter`, `Node`, and `Message`. Figure 5 illustrates these four base classes. It is noteworthy how minimal the actual code base for the proposed model is.

The class `PropertyFilter` is the base class for all property filters. It provides three virtual methods which are used to describe the behavior of any property filter. The methods are

- `virtual void init()`
- `virtual bool filter(Property properties_of_node[])`
- `virtual bool reiterate()`

The method `init()` is called once every time the filter is applied onto a node set. The method `filter()` is called for every node in a node set and should return `true` if the node should remain in the set and `false` otherwise. As some filters will need to iterate more than once over the set of nodes, e.g. a maximum filter, the method `reiterate()` is called after each iteration loop. If it returns `true` another iteration will be performed. The default implementation of `reiterate()` returns `false`.

The class `Node` is the base class for all nodes. It provides one virtual method to process incoming messages and several support methods:

- `virtual void process_messages(Message messages[])`
- `void add_property(Property p)`
- `void remove_property(Property p)`
- `void add_node(Node n)`
- `void remove_node(Node n)`

11

- `void send_message(Message m)`
- `void ensure_call()`

While most of the methods are self-explanatory, the method `ensure_call()` may need an explanation. If called, it ensures that `process_messages()` is called in the next time step even if there are no incoming messages. This could be useful for example, if the node wants to perform some computation in every time step regardless of the messages it receives. The method itself is rather a convenience method than an addition to the proposed computational model. Instead of using this method the node could send an empty message addressed to itself by means of some unique property, e.g. an unique ID.

The class `Message` is the base class for all messages. It has a member `address` which can hold an arbitrary property set expression to specifiy the recipient nodes of the particular message.

The constant `K` is used in the following examples to refer to the set of all nodes.

4.1 Conway's *Game Of Life*

John Horton Conway's *Game Of Life* [Gar70] is a widely known cellular automaton which can generate amazingly complex patterns based on a very simple set of rules. The cells are organized in a regular grid pattern where each cell has 8 neighboring cells. Each cell can either be active or inactive. The activation or deactivation of a cell is guided by the following two rules:

- If a cell is active and two or three neighboring cells are active too, the cell stays active. Otherwise, it deactivates.

- If a cell is inactive and exactly three neighboring cells are active, the cell activates itself.

The definition of this system in terms of the pseudocode introduced above is straight forward. It needs a property *Position*, a property filter *NeighborhoodOf*, a node *ConwayCell*, and a message *ActiveMsg*:

```
Position : Property {

    int x;
    int y;

}
```

The property `Position` is just a container for the two grid coordinates of the `ConwayCell` (see below).

```
NeighborhoodOf : PropertyFilter {

    Position refPos;

    NeighborhoodOf(Position rp) :
        refPos(rp)
    {}
```

```
        bool filter(Property properties_of_node[]) {
            Position p = properties_of_node["position"];
            if p is valid {
                int dx = abs(refPos.x - p.x);
                int dy = abs(refPos.y - p.y);
                return (dx == 1) || (dy == 1);
            }
            return false;
        }

    }
```

The property filter `NeighborhoodOf` provides a constructor to set a reference position. It selects via its `filter()` method all nodes which exhibit a property *position* that has a manhattan distance of one to the reference coordinates.

```
ConwayCell : Node {

    Position ownPos;
    bool     active;

    int survivalBegin = 2;
    int survivalEnd   = 3;
    int birthBegin    = 3;
    int birthEnd      = 3;

    ConwayCell(Position p, bool act) :
        ownPos(p),
        active(act)
    {
        add_property( ownPos );
    }

    void process_messages(Message messages[]) {
        if (active) {
            active = (messages.count >= survivalBegin) &&
                     (messages.count <= survivalEnd);
        } else {
            active = (messages.count >= birthBegin) &&
                     (messages.count <= birthEnd);
        }

        if (active) {
            ActiveMsg am;
            am.address = K * NeighborhoodOf( ownPos );
            send_msg( am );
            ensure_call();
        }
    }

}
```

The node `ConwayCell` provides a constructor to set the position and the inital state of the cell. It also makes the position an exhibited property by calling `add_property()`. The `process_messages()` method implements the two basic rules for activation and deactivation of the cell. If the cell turns out to be active after the rules are applied, it sends an `ActiveMsg` to its neighboring cells. In addition, the call to `ensure_call()` guarantees that the `process_messages()`

method is executed in the next time step. This is necessary if no surrounding cells are active in the current time step and therefore no `ActiveMsg` is sent to the cell. As only one type of message, i.e. the `ActiveMsg`, is used in this example, the `process_messages()` method does not have to distinguish between different message types. To apply the rules for activation and deactivation it can simply rely on the pure number of messages received.

Albeit the given implementation outline is aimed at Conway's *Game Of Life*, variations like Kellie Evans *Larger than Life* (LtL) [Eva01] can be implemented with only minor modifications. In case of LtL just the property filter defining the neighborhood as well as the survival and birth intervals of the cells would have to be updated.

4.2 Boids

In [Rey87] Craig Reynolds presents a computer model of "coordinated animal motion". He calls his simulated flocking creatures *boids*. In contrast to the cells of the previously described cellular automaton the boids can roam freely in a three-dimensional space. However, they too are guided by only a small number of local rules of which the three essential ones are:

- **Separation:** steer to avoid collisions with nearby flockmates.

- **Alignment:** steer to match the heading and speed of the flockmates in proximity.

- **Cohesion:** steer towards the average location of nearby flockmates.

In their simplest form boids have an internal state consisting of a position and a speed. In addition to this, boids have a limited perception. They can only sense flockmates within a certain distance and under a certain viewing angle. At first glance the general structure of the model given for the previous example seems to fit here too. The neighborhood of a boid could be modeled by a single property filter that uses a general assumption on how the boids' limited perception affects what each boid can sense. Although this approach is valid, it does not seem to fit the characteristics of the underlying system very well. The limited perception described above is an individual trait of each boid and should not be modeled separately as it is easy to imagine having several different types of boids with very different perceptual abilities within one simulation. Therefore, the model should express the shared concept of *boids seeing each other* on the one hand as well as the *individual aibility* of each boid *to see* on the other hand. The following model outlines a solution to this requirement by combining a so called *active* property with the object-oriented concept of an abstract class.

```
Position {
    float x;
    float y;
    float z;
}
```

```
Speed {
    float dx;
    float dy;
    float dz;
}
```

The first two classes of the model are `Position` and `Speed`. Both are simple containers for the respective values and as they are not used as properties in this example, they do not need to inherit from `Property`.

```
VisionSense : Property {

    virtual bool canSee(Position pos) = 0;

}
```

The property `VisionSense` is the first part of the aforementioned solution to combine a shared concept with individual and localized implementations. The property is defined as an abstract class that provides an interface to the subsequently defined property filter. The method `canSee()` should return true, if the particular boid which is exhibiting a property of type *VisionSense* can see the position *pos*. The property is dubbed an *active* property as its values are not read passively, but are queried actively by calling a member function.

```
HaveInSight : PropertyFilter {

    Position refPos;

    InSight(Position rp) :
        refPos(rp)
    {}

    bool filter(Property properties_of_node[]) {
        VisionSense vs = properties_of_node["vision"];
        if vs is valid {
            return vs.canSee( refPos );
        }
        return false;
    }

}
```

The property filter `HaveInSight` uses the interface defined by the property `VisionSense` to decide, if a message should be send to the particular biod which is exhibiting the property. Thus, it relates a group of boids to each other based on the individual capabilities of the involved boids by means of a shared, abstract property.

```
BoidInfoMsg : Message {

    Position boidPos;
    Speed    boidSpeed;

}
```

The message `BoidInfoMsg` encapsulates the actual information that is transmitted between a pair of boids. It contains the position and the speed of the boid which sends the message.

```
Boid : Node {

    Position ownPos;
    Speed    ownSpeed;

    BoidVision : VisionSense {
        Position ownPos;
        Speed    ownSpeed;

        void update(Position p, Speed s) {
            ownPos   = p;
            ownSpeed = s;
        }

        bool canSee(Position pos) {
            // return true if boid can see pos
        }
    }

    BoidVision ownVision;

    Boid(Position p, Speed s) :
        ownPos(p),
        ownSpeed(s)
    {
        ownVision.update( p,s );
        add_property( ownVision );
    }

    void process_messages(Message messages[]) {

        Speed d1,d2,d3;

        foreach BoidInfoMsg b in messages {
            d1 += calc_seperation_influence( b );
            d2 += calc_alignment_influence( b );
            d3 += calc_cohesion_influence( b );
        }

        normalize( d1, d2, d3 );

        ownSpeed = ownSpeed + d1 + d2 + d3;
        ownPos   = ownPos + ownSpeed;

        ownVision.update( ownPos, ownSpeed );

        BoidInfoMsg bim;
        bim.boidPos   = ownPos;
        bim.boidSpeed = ownSpeed;

        bim.address = K * HaveInSight( ownPos );

        send_msg( bim );
        ensure_call();
    }

}
```

Finally, the node `Boid` describes the abilities and behavior of a boid. The inner class `BoidVision` defines the local implementation for the abstract interface `VisionSense` and thus enables the definition of individual vision capabilities for every class of boids. If parameterized, it even allows the definition of individual vision capabilities for every single boid instance.

At last, the method `process_messages()` applies the three steering rules based on the received positions and speeds of the surrounding boids, it updates the boid's speed and position accordingly and it sends out the information about the updated position and speed to all other boids that have this particular boid in sight. As the boid should update its position and speed also if there are no boids in its perception range, the invocation of `ensure_call()` makes sure that `process_messages()` is called in every time step.

The presented solution to combine a shared concept with individual and localized implementations acts on two levels. The property `VisionSense` and the property filter `HaveInSight` encapsulate the abstract notions of *seeing* and *being seen* that induce the corresponding relation among the involved boids. The inner class `BoidVision` encapsulates, how the individual perceptual abilities of each boid contribute to this relation.

4.3 An Ubiquitous Computing Scenario

The third example illustrates the use of the proposed model to describe a basic ubiquitous computing scenario. The scenario has the following setup:

- A person is standing in a room and can point at several objects in the room.

- The pointing gestures are recognized by a 3D camera, e.g. a Kinect sensor.

- Some of the objects, e.g. lights, have special capabilities like being switchable.

- A voice recognition system captures commands from the person in the room.

The following sketch of the model describes how a voice command "*toggle that*" is processed such that the light the person is pointing at is toggled. Additionally, the model tries to capture some particular characteristics of the given scenario. One of these characteristics is the way, how the meaning of commonly passive objects changes in different contexts. For example, a common object which is not specified further has not much more properties than a position and some basic shape. But as soon as something interacts with that object the object acquires a property that reflects this interaction, e.g., if a person is pointing at that object, it acquires the property of being pointed at. In general, this observation can be stated as:

Objects are contextualized by interaction.

17

This notion is captured within the subsequent model by the common base node `Contextualizable`. This node is the base class for all objects in the scenario. It possesses a property `Position` and the special ability to exhibit properties that are externally impressed on the node. The node processes `ExhibitPropertyMsg` messages which contain the property to exhibit and a duration how long the property should persist.

```
Position : Property {
    float x;
    float y;
    float z;
}


ExhibitPropertyMsg : Message {
    Property propertyToExhibit;
    int      duration;
}


Contextualizable : Node {

    Position ownPos;
    Property extProperty;
    int      extDuration;

    Contextualizable( Position p ) :
        ownPos(p),
        extProperty(null),
        extDuration(0)
    {
        add_property(ownPos);
    }

    void process_messages(Message messages[]) {

        ExhibitPropertyMsg epm = messages["exhibitProperty"];

        if epm is valid and extProperty is null {
            extProperty = epm.propertyToExhibit;
            extDuration = epm.duration;
            add_property(extProperty);
        }

        if extDuration == 0 {
            return;
        }

        extDuration--;

        if extDuration > 0 {
            ensure_call();
            return;
        }

        remove_property(extProperty);
        extProperty = null;
    }

}
```

In this example the only descendant of `Contextualizable` will be the node `Light` which represents some form of switchable light source, e.g., a lamp standing in the corner of the room.

```
Switchable : Property {}

ToggleMsg : Message {}

Light : Contextualizable {

    Switchable sw;

    Light(Position p) :
        Contextualizable(p)
    {
        add_property(sw);
    }

    void process_messages(Message messages[]) {

        Contextualizable::process_messages(messages);

        ToggleMsg tm = messages["toggle"];

        if tm is valid {
            performToggle();
        }
    }

}
```

The node `Light` calls the method `process_messages()` of its parent class to ensure the proper handling of the *ExhibitPropertyMsg* messages. The light itself exhibits the property `Switchable` and processes `ToggleMsg` messages.

The two sensors of the scenario – the 3D camera and the voice recognition – are modeled by the nodes `Kinect` and `VoiceRecognition`. The node `Kinect` tries to detect a pointing gesture in every time step. If it is successful, it sends out a `ExhibitPropertyMsg` to impress a property `Selected` on the node that is being pointed at.

```
Selected : Property {

    float intensity;

    Selected(float i) :
        intensity(i)
    {}

}
```

To account for inaccuracies of the pointing gesture, the `Selected` property has a parameter *intensity* which correlates to the distance from the selection center. In order to different impress `Selected` properties with variable intensities on potential target objects, the `Kinect` node utilizes the subsequently defined property filter `BeingPointedAt`. With this propertery filter the `Kinect` node can create a set of ring shaped selections of potential recipients for the `ExhibitPropertyMsg` messages.

```
BeingPointedAt : PropertyFilter {

    PointingGesture refGesture;
    float           refRadius;

    BeingPointedAt(PointingGesture rg, float r) :
        refGesture(rg),
        refRadius(r)
    {}

    bool filter(Property properties_of_node[]) {
        Position p = properties_of_node["position"];
        if p is valid {
            //return true if p is in the direction of
            //refGesture and within a radius of refRadius
        }
        return false;
    }

}


PointingGesture {
    float baseX, baseY, baseZ;
    float dirX,  dirY,  dirZ;
}

Kinect : Node {

    void process_messages(Message messages[]) {

        PointingGesture p = detect_pointing_gesture();

        if p is valid {
            ExhibitPropertyMsg epm;
            epm.propertyToExhibit = Selected(1.0);
            epm.duration          = 10;
            epm.address           = K * BeingPointedAt( p, 1.0 );
            send_msg(epm);

            float radius;
            for (radius = 2.0; radius < 10.0; radius += 1.0) {
                epm.propertyToExhibit = Selected( (10.0 - radius) / 10.0 );

                epm.address = (K * BeingPointedAt( p, radius )) -
                              (K * BeingPointedAt( p, radius - 1.0 ));

                send_msg(epm);
            }
        }

        ensure_call();
    }

}
```

As described above, the Kinect node uses the property filter BeingPointedAt in combination with a common relative set complement to address potential recipient nodes in order to impress Selected properties with decreasing intensity.

Finally, the node VoiceRecognition checks in every time step if a new voice command was issued. If this is the case, a ToggleMsg message is send to that

node which

- exhibits a `Switchable` property,

- exhibits a `Selected` property,

- and has the highest *intensity* among those nodes exhibiting a `Selected` property.

To select the right node, i.e. the light the person points at, the `VoiceRecognition` node uses two property filters. The first property filter just checks, if a node exhibits a certain property. It is used to select all nodes exhibiting the `Switchable` property.

```
HaveProperty : PropertyFilter {

    Property refProp;

    bool filter(Property properties_of_node[]) {
        if properties_of_node contains property of type refProp {
            return true;
        }
        return false;
    }
}
```

The second property filter is an example for a filter that needs to reiterate over the set of nodes, as the filter selects the node with the highest *intensity* among those nodes exhibiting the `Selected` property.

```
MaxSelected : PropertyFilter {

    float maxInt;
    int   round;

    void init() {
        maxInt = 0.0;
        round  = 0;
    }

    bool filter(Property properties_of_node[]) {
        Selected s = properties_of_node["selected"];
        if s is valid {
            if round == 0 {
                maxInt = max( if s.intensity, maxInt );
                return true;
            } else {
                return s.intensity == maxInt;
            }
        }
        return false;
    }

    bool reiterate() {
        return round++ == 0;
    }
}
```

The definition of the node `VoiceRecognition` as already described above:

```
VoiceRecognition : Node {

    void process_messages(Message messages[]) {

        VoiceCommand vc = detect_new_voice_command();

        if vc is command "toggle that" {

            ToggleMsg tm;

            tm.address = (K * HaveProperty(Switchable)) * MaxSelected();

            send_msg(tm);

            }
        }

        ensure_call();
    }

}
```

The model of this third example is more extensive than it would have to be in order to illustrate some more advanced uses of the proposed computational model, e.g., the use of more complex property set expressions for addressing.

One important pattern that was demonstrated is the use of the *ExhibitPropertyMsg* message to manipulate the properties a node is exhibiting. By doing so a node, like the *Kinect* node in this example, can implicitly channel information to another node. It is worth noting, that the message which conveys the manipulation, itself is send indirectly by means of the *BeingPointedAt* property filter.

5 Conclusion

The proposed computational model offers a way to model and simulate a broad variety of complex systems. The model promotes and supports a strict encapsulation of all constituents involved in the behavior of the complex system. It facilitates the use of established object-oriented programming concepts like polymorphism and combines it with a simple, yet powerful mode of addressing as basis for the exchange of information inside the system.

The use of a generalized model offers not only the ability to describe a wider range of complex systems with a single approach but also allows for a more accurate modeling of systems that do not fit any of the complex system types favored by more specialized computational models.

References

[Eva01] Evans, Kellie M.: Larger than Life: Digital Creatures in a Family of Two-Dimensional Cellular Automata. In: Cori, Robert (Hrsg.) ; Mazoyer, Jacques (Hrsg.) ; Morvan, Michel (Hrsg.) ; Mosseri, Rémy (Hrsg.): *Discrete Models: Combinatorics, Computation, and Geometry, DM-CCG 2001* Bd. AA, Discrete Mathematics and Theoretical Computer Science, 2001 (DMTCS Proceedings), 177-192

[Gar70] Gardner, M.: Mathematical Games: The Fantastic Combinations of John Conway's New Solitaire Game 'Life'. In: *j-SCI-AMER* 223 (1970), oct, Nr. 4, S. 120–123. – ISSN 0036–8733 (print), 1946–7087 (electronic)

[MW66] Maslow, Abraham H. ; Wirth, Arthur G.: *The John Dewey Society lectureship series*. Bd. 8: *The psychology of science: a reconnaissance*. Harper & Row, 1966

[New11] Newman, M. E. J.: Resource Letter CS-1: Complex Systems. In: *American Journal of Physics* 79 (2011), aug, S. 800–810. `http://dx.doi.org/10.1119/1.3590372`. – DOI 10.1119/1.3590372

[NH11] Niazi, Muaz ; Hussain, Amir: Agent-based computing from multi-agent systems to agent-based models: a visual survey. In: *Scientometrics* 89 (2011), nov, Nr. 2, 479–499. `http://dx.doi.org/10.1007/s11192-011-0468-9`. – DOI 10.1007/s11192–011–0468–9. – ISSN 0138–9130

[Rey87] Reynolds, Craig W.: Flocks, herds and schools: A distributed behavioral model. In: *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*. New York, NY, USA : ACM, 1987 (SIGGRAPH '87). – ISBN 0–89791–227–6, 25–34