System Metamodeling of Open-Ended Evolution Implemented with Self-Modifying Code

Patrik Christen
FHNW, Institute for Information Systems, Switzerland
ETH Zurich, Chair for Philosophy, Switzerland

Having a model and being able to implement open-ended evolutionary systems are important for advancing our understanding of open-endedness. Complex systems science and the newest generation high-level programming languages provide intriguing possibilities to do so. Here, some recent advances in modeling and implementing open-ended evolutionary systems are reviewed (an earlier and shorter version was presented at [1]). Then, the so-called allagmatic method is introduced as a computational framework that describes, models, implements and allows interpreting complex systems using system metamodeling. Based on recent advances, the model building blocks evolving entities, entity lifetime parameter, co-evolutionary operations of entities and environment and combinatorial interactions are identified to characterize open-ended evolutionary systems. They are formalized within the system metamodel, providing a formal description of an open-ended evolutionary system. The study further provides a self-modifying code prototype in C# and guidance to create code blocks for an intrinsic implementation of open-ended evolutionary systems. This is achieved by controlling the self-modification of program code within the abstractly defined building blocks of the system metamodel. It is concluded that the identified model building blocks and the proposed self-modifying code provide a promising starting point to model and implement open-endedness in a computational system that potentially allows us to interpret novelties at runtime.

Keywords: Open-Ended Evolutionary Systems; Metamodeling; Self-Modifying Code; Combinatorial Evolution; Allagmatic Method

1. Introduction

The diversity and complexity of organisms created by biological evolution over the last billions of years is staggering. It seems to really be a never-ending story of inventions [2, 3]. Engineered physical systems, evolutionary and genetic algorithms, artificial intelligence, deep learning and other computational methods are far from simulating and explaining the diversity, creativity and open-endedness exhibited by

https://doi.org/10.25088/ComplexSystems.32.4.353

biological evolution. The main deficiency is that all these computational systems reach an equilibrium state and cease to generate further adaptive novelty—they are essentially cul-de-sacs.

Understanding the open-endedness of biological evolution is a grand challenge [4], especially in the field of artificial life. If implemented in an open-ended computational system, it would have major implications far beyond artificial life [2, 3]. It would allow us to invent virtually everything, including new architectures, furniture, cars, games and of course algorithms and software in general [2, 3]. It would most likely bring us closer to strong artificial intelligence, since only biological evolution has created it so far [2, 3].

Furthermore, open-endedness has been observed in various complex systems such as human languages, legal systems, economic and financial systems and technological innovation, showing its relevance as well as urging its study [5, 6]. These systems are an important part of our society. A better understanding of their open-ended dynamics, that is, when a system is completely reorganizing itself, is key to managing these systems. Reorganization of a system can be, for example, a crash of a financial system or the collapse of an ecological system. It happens unpredictably and from time to time in systems relevant to society and is therefore related to some of our biggest challenges, including climate change and socioeconomic stability [7].

The open-ended evolution community made remarkable progress defining and exploring open-endedness by creating systems such as Avida [8] and Geb [9], algorithms such as novelty-driven approaches [10], and even programming languages such as ulam [11]. By studying evolution in general, evolutionary biology and complex systems science also contribute to the understanding of open-endedness, although in an implicit way. For example, evolutionary biology provides insight into speciation mechanisms [12], complex systems science gives an evolutionary model that shows punctuated equilibria [13, 14], and dynamical systems theory contributed to the more formal development of open-ended evolution based on theoretical concepts such as Poincaré recurrence time [15] and methods derived from algorithmic complexity theory [16]. The technical contribution of the open-ended evolution and artificial life community regarding implementing these systems is also impressive since it requires some kind of self-modification and self-referencing capabilities to account for novelties at runtime [6]. Despite this progress, there is no model yet that produces novelties and shows creativity as observed in complex systems such as the economy or biological evolution. It seems that we are missing an important ingredient or idea.

To get an overview of potential model building blocks of open-endedness and implementation approaches across different fields, this paper first presents a short review of some recent advances in

modeling and implementing open-ended evolutionary systems from the fields of artificial life, including the open-ended evolution community, complex systems science, dynamical systems theory, artificial intelligence and evolutionary algorithms, and evolutionary biology. Then, to connect the ideas from different fields within a coherent framework, the so-called allagmatic method is introduced. It describes, models, implements and allows interpretation of complex systems. After highlighting some current modeling and implementation challenges, model building blocks of open-ended evolutionary systems are identified and a system metamodel of open-ended evolution is proposed as part of the allagmatic method. In terms of implementing open-ended evolutionary systems, a self-modifying code prototype in a high-level programming language is presented, and the allagmatic method is used as guidance to create code blocks with the developed self-modifying code prototype. It is concluded that the identified model building blocks and the proposed self-modifying code provide a promising starting point to model and implement open-endedness in a computational system that potentially allows us to interpret novelties at runtime.

2. Recent Advances in Modeling Open-Ended Evolutionary Systems

2.1 Definitions

Although progress has been made, especially by the open-ended evolution community, much remains to be explored [17]. Before having a closer look at modeling, we start with some preliminaries regarding the definition of open-ended evolution or open-endedness.

Open-endedness has been defined as the ability to continually produce novelty and/or complexity whereby novelty is classified as variation, innovation and emergence [6]. Based on creativity research [18], different terms for this classification were suggested, namely exploratory, expansive and transformational novelty, respectively [19]. The latter terms will be used here to avoid interpretation issues with innovation and emergence. Regardless of the terminology, both definitions relate to a formal model and metamodel of the system under study. *Exploratory novelty* can be described using the current model, *expansive novelty* requires a change in that model but still uses concepts in the metamodel, and *transformational novelty* introduces new concepts necessitating a change in the metamodel [6, 19]. With their connection to model and metamodel, they provide a way to determine whether and which kind of novelty emerges in an open-ended evolutionary system.

Defining complexity and its measurement in open-ended evolutionary systems is a topic of ongoing research too. Dolson et al. [20]

https://doi.org/10.25088/ComplexSystems.32.4.353

recommend an information-theoretic approach based on the count of informative sites across all components in a population and suggest improving it by also accounting for all possible mutations and by considering epistatic interactions. Channon [21] defines individual complexity as the diversity of adaptive components in the individual, that is, the number of active genes. Furthermore, in evolutionary biology, information is quantified with respect to different sources available to an adapting organism, from ancestors and the environment [22].

2.2 Modeling Contributions from Artificial Life and Open-Ended Evolution Community

Banzhaf et al. [6] have argued that open-endedness in physical systems such as in a computation are hard to prove in a finite universe and therefore we might endeavor to create a sufficient rather than an infinite number of open-ended events, which is then called *effectively* open-ended. To achieve this despite the limits on computational power, it was suggested to hard-code certain elements of the model, for example, the process of replication, into so-called *shortcuts* [6, 19]. Taylor [19] bases shortcuts on generally accepted processes of Darwinian evolution: phenotype generation (from the genotype), phenotype evaluation and reproduction with variation. Ongoing evolutionary activity and with that exploratory open-endedness are promoted by modifying the adaptive landscape, the topology of genetic space or the genotype-phenotype map. He further argues that none of these expand the phenotype space itself and thus do not help us for an expansive and transformational open-endedness, where so-called *door-opening* states in phenotype space are needed. The complexity of physical and chemical laws provides a vast space for biological systems, whereas in computational systems we might dynamically increase the space instead, for example, providing access to additional resources on the internet [18, 19, 23]. In contrast, at the third workshop on open-ended evolution, Taylor and others from the open-ended evolution community mentioned that current computational systems implement rather scanty environments and organisms.

Taylor [19] also proposes two possible intrinsic mechanisms to access new states. The first is via *exaptation*, where a trait changes its function to a different one from the one it was originally adapted for. Physical systems are composed of multi-property components having several properties in different domains (mechanical, chemical, electrical, etc.) [19]. For example, a multifunctional enzyme has multiple properties in the same domain, which can produce expansive novelties, whereas transformational novelties can be achieved by properties in different domains [19]. The second is via *non-additive composition*, which is phenotype generation by assembling several components drawn from a set of component types [19]. For example, the

construction of proteins from amino acid sequences, producing new molecules and introducing new functions [19].

This assembly of lower-level elements into higher-level structures is also highlighted by Banzhaf et al. [6]. With the above-mentioned metamodel that defines novelties, they also provide an abstract way to model multiple levels, accounting for such constructed structures at different levels [6]. They also mention that having several levels drastically increases the combinatorial possibilities to construct new structures and with that also the demand for computational power [6]. It therefore seems to be a way to increase the opportunities to create something new. It also implies that open-ended evolution in computational systems is computationally expensive.

2.3 Modeling Contributions from Complex Systems Science

There are also relevant modeling contributions from the field of complex systems science. W. Brian Arthur is known for his work on complexity economics [24] and technology evolution [25, 26]. He proposed the concept of *combinatorial evolution*, which states that new technologies are created out of existing technologies and iteratively, these newly created technologies become building blocks for yet further technologies [26, 27]. The collective of technology is therefore self-creating or autopoietic with the agency of human beings [25, 27]. In a simple computer model of circuits, Arthur and Polak [28] showed that complicated technologies (in their case circuits) could be created out of simpler building blocks, and they found evidence of self-organized criticality. It requires some kind of modularity and the evolution of simpler steppingstone technologies [25– 28]. The latter means that we cannot create a technology ahead of time without first creating the simpler precursor technologies. Natural phenomena also provide technological elements that can be combined [25, 27]. In terms of open-endedness, there seems to be a vast space of possible combinations, and with the conversion of discovered natural phenomena into technological elements there is a mechanism in place to expand that space.

Combinatorial evolution is also part of a more general approach to modeling evolution by the complex system scientist Stefan Thurner. He and his colleagues recently introduced the co-evolutionary, combinatorial and critical evolution model (CCC model) [14, 29–33]. It models evolution as an open-ended process of creation and destruction of new entities emerging from the interactions of existing entities with each other and with their environment [14]. The spaces of entities and of interactions co-evolve, and new entities emerge spontaneously or through the combination of existing entities. This leads to power law statistics in the event histories [14, 30]. Selection is modeled by specifying rules for what can be created and what will be

https://doi.org/10.25088/ComplexSystems.32.4.353

destroyed [14, 29, 30]. This model captures so-called *punctuated equilibria* in biological evolution [34] or *Schumpeterian business cycles* in economic evolution [35], where an equilibrium is destabilized or destroyed by a critical transition leading to another equilibrium in an ongoing and thus open-ended process [13, 14, 29, 30]. It is interesting to note that it could be shown that in economic innovation, *creative deconstruction* is happening and not *niche filling* as usually assumed for biological innovation [14, 29, 31].

A more detailed and formal description of the CCC model is given in the following, based on [13] and [14]. Evolution is described in three steps as a process: (1) A new entity is created and placed in an environment. An entity can be a species, product, technology and so on. An entity's state is described by σ_i and its environment by σ_j in the same state vector $\boldsymbol{\sigma}$. (2) The newly added entity interacts with its environment, which also includes the already existing entities. Based on these interactions, the entity is either removed from or added to the system. The interaction of entity i and environment j is described by an interaction matrix M_{ij}^{α}, where α denotes the type of interaction. If more than two entities are interacting, for example, entity k, a further dimension is needed and thus a tensor M_{ijk}^{α}. The evolution of states can now be given by

$$\frac{d}{dt}\sigma_i(t) \sim F\big(M_{ijk\ldots}^{\alpha}(t),\, \sigma_j(t)\big), \tag{1}$$

where F is some function depending on the state vectors and the interactions. (3) If the new entity is added to the system, it becomes part of the environment, which therefore changes the environment. This induces existing entities trying to relax toward a new equilibrium that was disrupted by the changing environment (boundary condition). Since entities are added and removed from the system over time, the interactions M also change over time. To account for that, a second equation to describe the evolution of interactions is introduced

$$\frac{d}{dt}M_{ijk\ldots}^{\alpha}(t) \sim G\big(M_{ijk\ldots}^{\alpha}(t),\, \sigma_j(t)\big), \tag{2}$$

where G is some function depending on the state vectors and the interactions. The combination of these two evolutionary equations results in *co-evolutionary dynamics*.

2.4 Modeling Contributions from Dynamical Systems Theory

Adams et al. [15] provide another systems approach and formally define and treat open-ended evolution as a more general problem in dynamical systems theory. They introduce a criterion for open-ended evolution based on the features *unbounded evolution* and *innovation*.

Poincaré recurrence time, that is the maximum time until the dynamical trajectory repeats in an isolated system, is used to formally define a minimal criterion for unbounded evolution in finite dynamical systems. They state that an unbounded system is one that does not repeat within the expected Poincaré recurrence time. Since finite deterministic systems do not meet this criterion, some kind of external perturbation is required, and thus unbounded evolution is only possible in a subsystem interacting with an external environment. The second criterion of innovation is defined as dynamical trajectories not observed in isolated, unperturbed systems. Also in this case, an interaction between at least two subsystems is required. A subsystem is therefore open ended if it is unbounded and innovative. Applying this definition in cellular automata, Adams et al. [15] show that systems with time-dependent rules as a function of their state are statistically better at meeting the defined criteria for open-endedness than systems with externally driven time dependence. Through the coupling to larger environments, these results show that state-dependent systems provide a mechanism for generating open-ended evolution. They conclude that open-ended evolution is a general property of dynamical systems with update rules that are time dependent. This is in contrast to the classical modeling approach, where dynamical rules remain fixed. It is interesting to observe that also in the CCC model described earlier, the environment opens the system to external perturbations, underlining the importance of this possible mechanism of open-ended evolution. The mechanism is also comparable to door-opening states in a phenotype space as described by Taylor [19, 36].

The study from Hernández-Orozco et al. [16] also formally defines open-ended evolution in dynamical systems, however, using methods derived from algorithmic complexity theory and investigating whether undecidability is a requirement for open-ended evolution. They characterize open-ended evolution in computable dynamical systems as a process in which families of objects with increasing complexity are produced and present a general mathematical model for adaptation. This allowed them to show that decidability imposes universal limits on the growth of complexity in computable systems. Furthermore, they also show that the undecidability of adapted states and the unpredictability of the behavior of the systems at each state are required for open-ended evolution and that such behavior is irreducible.

2.5 Modeling Contributions from Artificial Intelligence and Evolutionary Algorithms

Open-ended evolution is also studied in artificial intelligence and evolutionary algorithms. It is an emerging topic where the research of Kenneth O. Stanley and his colleagues serves as an example here. They tried to get rid of the prevailing concept of optimizing a fitness

https://doi.org/10.25088/ComplexSystems.32.4.353

function and have even suggested abandoning objectives in general [37–39]. They showed that a novelty-driven approach finds solutions faster and results in solutions with less genomic complexity in comparison to traditional evolutionary computation [10]. They also devised several algorithms, including novelty search with explicit novelty pressure, MAP-Elites and innovation engines with explicit elitism within niches in an otherwise divergent process, and minimal criterion co-evolution where problems and solutions can co-evolve divergently [40, 41]. Like Thurner, avoiding objectives also allowed Stanley and his colleagues to model punctuated equilibria with transitions between equilibria in a simple simulation with voxel structures [42]. Also, in this case co-evolution and the apparently never-ending creation of anything new by combining existing structures were essential ingredients.

2.6 Modeling Contributions from Evolutionary Biology

The work of Thurner and Stanley indicates that transitions between equilibria are an important part of open-ended evolution. In evolutionary biology, the major evolutionary transitions are of great interest too, for example, the transition from unicellular to multicellular organisms [43, 44]. Here, only a small selection of research is presented, mainly on mechanisms that can explain rapid increases in diversity and biological innovation. The work of evolutionary ecologist Ole Seehausen illustrates this well, as he is interested in mechanisms by which diversity arises. Especially relevant here is the possibility of speciation through combinatorial mechanisms. In such cases, new combinations of old gene variants can quickly generate reproductively isolated species and thus provide a possible explanation for rapid speciation [12]. For example, he showed that hybridization between two divergent lineages provides ample genetic starting variation. This is then combined and sorted into many new species, fueling rapid cichlid fish adaptive radiations [45]. Seehausen furthermore investigates and underlines the importance of jointly considering species traits and environmental factors in speciation and adaptive radiation as they affect one another [46, 47]. His work therefore supports the importance of co-evolutionary and combinatorial dynamics for open-ended evolution, even though co-evolution is between species in a heterogeneous environment and combinations happen at the gene level.

Biological insights into innovation itself are also relevant. The work of evolutionary biologist Andreas Wagner illustrates this nicely [48, 49]. For example, he showed that recombination creates phenotypic innovation in metabolic networks much more readily than random changes in chemical reactions [50]. The work of Wagner suggests that recombination of genetic material is a general

mechanism that greatly increases the diversity of genotypes [51, 52]. Also, relevant here is his work on evolutionary innovation through exaptation. He found that simulated real metabolic networks were not only able to metabolize on a specific carbon source but also on several others, which shows that metabolic systems may harbor hidden pre-adaptations that could potentially lead to evolutionary innovations [53]. Combinatorial interactions at the gene level again play a crucial role, and the latter study revealing hidden pre-adaptations is like Stanley's open-ended algorithms, creating many potential solutions before it is applied to solve an actual problem.

Besides this limited and biased review of contributions from evolutionary biology, it seems nevertheless important to point out that the field has shown that combinatorial interactions matter at organizational levels above the genes and that a changing environment can greatly affect species diversity and vice versa.

3. Recent Advances in Implementing Open-Ended Evolutionary Systems

3.1 Implementation Contributions from Artificial Life and Open-Ended Evolution Community

We first consider implementations from the artificial life and open-ended evolution community. Banzhaf et al. [6] and Taylor [19] provide some implementation suggestions. The implementation of computational systems that can detect and integrate novelties into the model and metamodel as described by Banzhaf et al. [6] and Taylor [19] provides a challenge in its own right. It is argued that operations should be defined *intrinsically* in the system and by the system itself [19, 54]. It requires program code that can recognize and modify itself. Banzhaf et al. [6] state that this can be achieved by representing entities as strings of assembly language code or by using a high-level language designed specifically for this purpose [55], or a *reflective* language. A reflective language allows implementing programs that have the ability to manipulate and observe their states during their own execution [56]. Indeed, it was possible to generate exploratory, expansive and transformative novelties with Stringmol, where modifications happen in sequences of assembly language code [57]. A replicator and some of the observed operations and structures were defined extrinsically, whereas some others could be defined intrinsically [57].

There are several computational systems, of which Avida [8] and Geb [9] are two prominent examples. Usually, digitally simulated organisms are represented by assembly code competing for limited CPU resources. Most of these systems implement extrinsically common shortcuts such as replication and a certain fitness function, which makes them a powerful tool to explore biological questions

https://doi.org/10.25088/ComplexSystems.32.4.353

such as the genotype-phenotype mapping [58] in a highly controlled way. Another strength of computational systems is that they usually involve visualizations, for example, Sims [59], and thus contribute to the exploration of complex evolutionary dynamics. With respect to open-endedness, however, Pugh et al. [42] point out that none of these systems has generated explosions of complexity, as seen in biological evolution during transitions, and therefore something must still be missing. With Voxelbuild, Pugh et al. [42] have contributed the most relevant computational system in this respect. A first prototype demonstrated that a certain organization of voxels emerged, which was used as a steppingstone for other organizations that emerged later. This seems to be a kind of combinatorial evolution. Additionally, they report that exaptation has occurred, which is reminiscent of evolutionary biology studies.

3.2 Implementation Contributions from Complex Systems Science and Dynamical Systems Theory

Thurner et al. [14] add another important aspect to the implementation. They argue that only a so-called *algorithmic* implementation and thus discrete formulation can work because in evolutionary systems, boundary conditions cannot be fixed (the environment evolves as a consequence of the system dynamics), and the phase space is not well defined as it changes over time [14]. It would lead to a system of dynamical equations that are dynamically coupled to their boundary conditions, which is according to them a mathematical monster and the reason why evolutionary systems cannot be implemented following an analytical approach. In addition, with the CCC model, they provide a general description of a complex evolving system that is so general that it applies to every evolutionary system. It therefore provides a general metamodel layer of a computational evolutionary system, which suggests that it might not need a change to capture novelties.

The algorithmic approach is capable of creating complex structures and behaviors based on simple rules described with an algorithm and run iteratively with time on a computer. This has been shown with elementary cellular automata [60, 61] and hypergraphs [62, 63]. Adams et al. [15] add to this an implementation of cellular automata with time-dependent update rules, providing a way to implement dynamical systems with the capability to expand their state space at runtime.

4. The Allagmatic Method

4.1 Modeling Contribution

We have developed the so-called *allagmatic method* [64, 65] to describe, model, implement and allow interpretation of complex

systems (Figure 1). It consists of a system metamodel inspired and guided by philosophical concepts of Gilbert Simondon [66, 67] and Alfred North Whitehead [68, 69]. Simondon's metaphysics gives an operational and systemic account of how technical and natural objects emerge and evolve. It allows the abstract definition of a system with the concepts *structure* and *operation*, since according to him, systems develop from a seed through a constant interplay between operations and structures [70]. More concretely, but still general, we defined model building blocks in a system metamodel. The main building blocks are *entity*, *milieu*, *update function*, *adaptation function* and *target*, for which we recently provided a mathematical formalism [71]. The concepts entity, adaptation and control are borrowed from Whitehead [68, 69] as described in a recent conference paper [72]. The creation of a system model and metamodel can be followed through three regimes: In the virtual regime, abstract definitions with classes corresponding to interpretable philosophical concepts are given. Using generic programming [73], the type of states an entity can have are defined by defining a system model object that has not yet initialized any parameters. At this point the metastable regime starts, where step by step the object/model is concretized with parameters such as number of entities and concrete update functions (model individuation). Once all parameters are defined, the object is executed in the actual regime. If there are any adaptation processes involved, the allagmatic method cycles between the metastable and actual regimes.

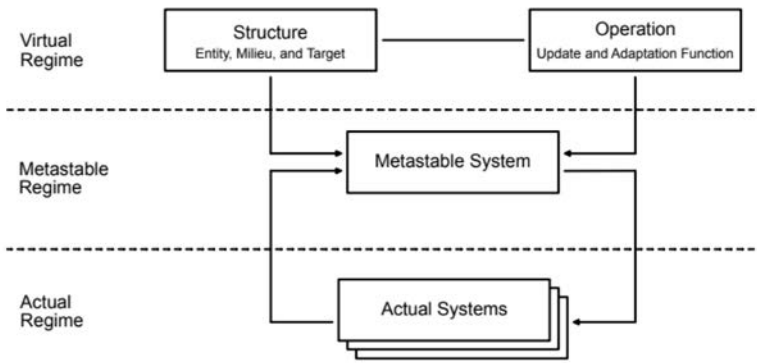

Figure 1. Overview of the allagmatic method and its system metamodel's main building blocks. Figure adapted from [72].

4.2 Implementation Contribution

The programming of the allagmatic method with its system metamodel is aligned with philosophical concepts. This allows not only an

interpretation of the final result in the context of the related metaphysics but also the tracing of the developmental steps a model is undergoing, and thus provides a way to study the emergence of the typical characteristics of complex systems. We have recently outlined how adaptation can be studied in this way [72]. In this work, we also introduced the possibility and concepts to form hierarchies and define control, which further support the use of the allagmatic method to define concepts that are difficult to pin down.

Furthermore, we showed how the method might be used for automatic programming [65]. We found that the abstract model building blocks are well suited to be automatically combined by self-modifying code in a high-level language. Our work shows that certain philosophical concepts and even metaphysics as a whole can be defined and implemented in program code, providing the opportunity to run these concepts or the whole metaphysics and study them in action.

We also created a prototype of open-ended automatic programming through combinatorial evolution [74, 75]. Like Arthur and Polak [28], we created a computational model based on combinatorial evolution but instead of evolving circuits, we evolved computer code. Useful code blocks were stored in a repository and could be used in later iterations. Starting with basic keywords available in the programming language, more complex code blocks including classes, void methods and variable declarations evolved.

5. Current Modeling and Implementation Challenges

5.1 Modeling Open-Ended Evolutionary Systems

Co-evolutionary dynamics, *combinatorial interactions* and a *changing environment* seem to be important ingredients of open-ended evolutionary systems. The work of evolutionary biologists including Seehausen and Wagner supports the view that co-evolutionary dynamics and combinatorial interactions are key elements. They also indicate that biological evolution exhibits different levels or types of combinatorial interactions, and that the environment is an important driver and mediator of change. The CCC model accounts for co-evolutionary dynamics and combinatorial interactions and successfully generates the statistics of economic data with reoccurring transitions between equilibria [14, 29–33]. It could also show that economic innovations are driven by creative destruction, thus Schumpeterian evolution. This provides important insights into the open-ended dynamics of economic evolution [14, 29, 31]. However, it still needs to be investigated in other evolutionary systems, especially in biological evolution. Banzhaf et al. [6], Taylor [19], Adams et al. [15] and Hernández-Orozco et al. [16] provide guidance for modeling open-ended

evolution in general, which might allow us to come up with a model that captures open-ended dynamics of any evolutionary system, including economic and biological systems.

5.2 Implementing Open-Ended Evolutionary Systems

There is the challenge of an intrinsic implementation of open-ended evolutionary systems. The programming techniques for this already exist; however, the real challenge is linking the structure and events in the implementation with interpretable concepts. To illustrate this problem, assume that we dispense with all shortcuts and let the program completely overwrite the model and metamodel. Having no replicator or other prevailing concepts makes it hard to understand and see what is going on in the evolutionary simulation. This problem was discussed at the third workshop on open-ended evolution [17, 76]. It is mostly uncharted territory that requires much more research, including how to identify certain concepts and components from simulation data and how to implement such systems in a purely intrinsic manner, where generated novelties are meaningfully integrated into the model/metamodel by the evolving systems themselves.

Another challenge is the choice of digital organisms and environment. The CCC model [14, 29–33] provides a mathematical formalism for theoretical considerations and ways to perform statistical analyses. Computational systems from artificial life and the open-ended evolution community such as Voxelbuild [42] usually come with powerful visualizations; however, they lack a mathematical formalism.

6. The Allagmatic Method for Open-Ended Evolutionary Systems

6.1 Model Building Blocks of Open-Ended Evolutionary Systems

Observing evolving systems like technology or the rain forest makes it clear that not only entities evolve but also the interactions among them. Co-evolution implies that species have a mutual influence on each other [77]. Since species are also part of the environment, co-evolution leads to a changing environment, providing more possibilities for state changes. Also, external environmental input can change and affect species and their interactions. Combinatorial interactions create new entities from existing entities [14, 25, 26, 29]. These newly created entities might be able to exploit different parts of the changing environment and thus perhaps fill any niches that may arise. Chromaria [78] captures this to some degree, as entities become part of the environment and thus change the environment that interacts with further new entities. Changing the interactions between entities and

https://doi.org/10.25088/ComplexSystems.32.4.353

between entities and their environment leads to complex cascades of changes, potentially resulting in disruptive changes in the system that can be regarded as novelties. This is further supported by the findings of Adams et al. [15] and Hernández-Orozco et al. [16]. Combinatorial interactions also lead to evolutionary changes and potential novelties; they combine existing entities to form new entities. This can be observed nicely in the evolution of technology [25, 26]. It is a possibility for how transitions might be explained, for example, from unicellular to multicellular organisms [43, 44].

We could therefore use the allagmatic method to capture the co-evolutionary dynamics, including the environment and the combinatorial interactions as given by the CCC model. The CCC model has been able to generate an ongoing evolutionary process with punctuated equilibria when, in addition, the lifetime of an entity was limited [14, 29, 30]. It showed the statistical behavior of open-ended evolutionary systems. Here, the CCC model is formalized within the allagmatic method to allow interpretation within the implemented metaphysics of Simondon [66, 67] and Whitehead [68, 69]. The system metamodel of the allagmatic method and the CCC model both follow a complex systems perspective, which makes them compatible.

The model building blocks or general concepts to be captured with the allagmatic method are specifically: evolving entities, entity lifetime parameters, co-evolutionary operations of entities and environment and combinatorial interactions.

6.2 System Model and Metamodel of Open-Ended Evolution

The allagmatic method consists of a system metamodel for modeling systems in general and complex systems in particular (see Christen and Del Fabbro [71] for detailed mathematical definitions). The system metamodel describes individual parts of a system as entities defined with an entity e-tuple $\mathcal{E} = (\hat{e}_1, \hat{e}_2, \hat{e}_3, \ldots, \hat{e}_e)$, where $\hat{e}_i \in Q$ with Q being the set of k possible entity states. Entity states are updated over time according to an update function $\phi : Q^{m+1} \to Q$ with m being the number of neighboring or linked entities. The update function ϕ therefore describes how entities evolve over time, dependent on the states of neighboring entities. Update rules and thus the logic are stored in the structure update rules $\mathcal{U}$. Entities are thereby considered connected together in a network structure and defined with the milieus e-tuple $\mathcal{M} = \left(\hat{\mathcal{M}}_1, \hat{\mathcal{M}}_2, \hat{\mathcal{M}}_3, \ldots, \hat{\mathcal{M}}_e\right)$, where $\hat{\mathcal{M}}_i = (\hat{m}_1, \hat{m}_2, \hat{m}_3, \ldots, \hat{m}_m)$ is the milieu of the i^{th} entity $\hat{e}_i$ of $\mathcal{E}$ consisting of m neighbors of $\hat{e}_i$. Over time, update function ϕ and milieus $\mathcal{M}$ might be changing as well, which is described with the adaptation function ψ.

We now extend the system metamodel where needed with concepts to model open-ended evolution as identified earlier. *Evolving entities*: The entity e-tuple $\mathcal{E}$ captures evolving entities (entities changing their state over time) in the same sense as the general evolution algorithm [14] does with the state vector σ. The general evolution algorithm can be regarded as the metamodel from which the CCC model is created [14]. *Co-evolutionary operations of entities and environment*: With the creation of new entities (novelty), new possibilities for interactions also emerge. This is key for open-endedness and is captured by the co-evolution of entities and their interactions in the general evolution algorithm [14]. Formally, the update equations of the entity state vector σ and the interaction tensors M are simultaneously updated over time in the general evolution algorithm. In the system metamodel of the allagmatic method, this is described with the update function ϕ and the adaptation function ψ, which can both be modeled in such a way that the update function ϕ updates entity states in $\mathcal{E}$ simultaneously with the adaptation of their interactions in the milieus $\mathcal{M}$ through the adaptation function ψ. This is a concretization of the system metamodel into a metamodel of open-ended evolution. The environment is also part of co-evolution and is described in the state vector σ in the general evolution algorithm [14] and the entity e-tuple $\mathcal{E}$ in the system metamodel. *Combinatorial interactions*: In complex systems, interactions are of combinatorial nature consisting of rules determining how new entities can be formed out of existing entities. The creation and destruction of entities are encoded in rules that do not change with time. They can be regarded as physical or chemical laws determining which transformations and reactions are possible, respectively. Please note that this covers the typical evolutionary mechanisms of selection, competition and reproduction. At runtime, models created from this metamodel make use only of a subset of these rules at any given time point, which is captured with so-called active productive/destructive rules. This is formally described with the function F in the general evolution algorithm [14] and the update function ϕ in the system metamodel. *Entity lifetime parameter*: Besides the creation of new entities through combinatorial interactions, entities can spontaneously appear, which would be like discovering a new law or element in nature. By introducing a decay rate λ, the CCC model did not freeze [14]. It thus plays an important role for open-ended evolution. In the system metamodel, this parameter can be described as a further structure with a respective further operation.

6.3 Self-Modifying Code Prototype in C#

An intrinsic implementation as suggested by Banzhaf et al. [6] and Taylor [19] requires self-modifying program code and some way to

https://doi.org/10.25088/ComplexSystems.32.4.353

add novelties to the model or metamodel. Interpreting these novelties in the context of a certain metaphysical framework will most likely require a high-level language with the capabilities to modify program code during runtime and reflect on it. C# provides these capabilities with the open-source Roslyn .NET compiler [79]. The compiler platform provides dynamic code manipulation with syntax trees and many other features, including reflection as well as comprehensive code analysis. Syntax trees can either be created from a string containing program code or they can be assembled using predefined classes. As opposed to writing program code into a file and then compiling it, syntax trees can be stored as an object and compiled and executed at runtime.

In the allagmatic method, a general layer in the system metamodel that is not modifiable by the code is suggested here. These are the model building blocks every complex evolutionary system requires. However, there is also a layer in the metamodel that is modifiable by the code. It consists of less general model building blocks that are basically more concrete instances of the general layer. With these different layers and controllable code self-modification, it will potentially be possible to link concepts defined in the metamodel to newly generated code, improving interpretability. The present study provides a first prototype of self-modifying code in C# [80], bringing us one little step closer to that ambitious goal.

Fundamentally, implementing self-modifying code requires considering at least three basic questions: what words to use, how to concatenate these words to create valid code and how to implement the duality between code and data. In theoretical computer science, a *word* or string is defined as a finite sequence of symbols over a given alphabet [81]. An *alphabet* is a finite set of symbols [81] and *symbols* are the basic constituents of any language (i.e., the set of all words over a given alphabet), for example, letters, digits or any other characters [81]. To define the words, we consider universal computation and code interpretability. We want to choose words that do not limit the generated code and therefore require universal computation or Turing completeness. It has been shown that only the instructions load, store, increment and goto (unconditional branching) are required to achieve universal computation [82]. Most widely used programming languages including C++ and C# provide words to generate these instructions and many other instructions and are thus capable of universal computation. Regarding code interpretability, we suggest including the complete or most of the syntax of a high-level programming language since these languages are designed to be human readable and interpretable. The first words to include are therefore all the C# keywords as defined in the C# language reference [83]. In addition to keywords, we also include special characters, the member access

expression and operators as defined in the C# language specification [84], as well as some further words. Please note that we treat symbols such as operators as words since we can concatenate them with other words to create sentences (i.e., instructions). In the following, all included words are listed:

- Keywords: abstract, as, base, bool, break, byte, case, catch, char, checked, class, const, continue, decimal, default, delegate, do, double, else, enum, event, explicit, extern, false, finally, fixed, float, for, foreach, goto, if, implicit, in, int, interface, internal, is, lock, long, namespace, new, null, object, operator, out, override, params, private, protected, public, readonly, ref, return, sbyte, sealed, short, sizeof, stackalloc, static, string, struct, switch, this, throw, true, try, typeof, uint, ulong, unchecked, unsafe, ushort, using, virtual, void, volatile, while
- Special characters: { } ( ) [ ] " ; ,
- Member access expression: .
- Arithmetic operators: + - * / %
- Relational operators: < > <= >= == !=
- Logical operators: & ^ | && ||
- Assignment operator: =
- Further words: IDENTIFIER, NUMBER, PLACEHOLDER

There are some further words that need to be explained. Program code contains words that are used as a name or identifier, for example, for variables and classes. To account for that, the word IDENTIFIER is included as a possible word. If the self-modifying code chooses this word, an identifier is generated and inserted. Natural numbers are inserted in the same way, replacing the word NUMBER. If the word PLACEHOLDER is chosen, an instruction that is a combination of words representing valid code is inserted. Such a placeholder allows generating a nested code structure, for example, a variable declaration inside a method [74, 75]. In this first prototype of self-modifying code, syntax to make use of the extensive .NET API is not included. Thus, the language used here is a subset of the C# language.

We now address the second question regarding the combination of the defined words to create valid code. The general algorithm to achieve this is based on the concept of combinatorial evolution as proposed by W. Brian Arthur [26, 27] and already used in our earlier study to evolve programming concepts such as variable declarations and classes in Java [74, 75]. The algorithm uses two data structures, a list words (set W) containing the previously defined words and a list codeBlocks (set C) storing valid code blocks (sentences), which is initially empty ($C = \emptyset$). It is an iterative process in which several

steps are repeated: (1) The first step is to generate a new code block. This is achieved by randomly selecting a given number of words from set W, which are then concatenated, separated by a space. The number of words in a code block is set randomly between two and eight as in the previous studies where combinatorial evolution was simulated [28, 74, 75]. If the chosen word is PLACEHOLDER, it is replaced by an already existing valid code block from set C or by another word if set C is still empty. Similarly, if the chosen word is IDENTIFIER, it is replaced by a numbered identifier. If the chosen word is NUMBER, it is replaced by a randomly generated integer. (2) The second step is checking the validity of the newly generated code block. This is achieved by parsing the code block into a syntax tree, which is then analyzed making use of the Roslyn API (the .NET Compiler Platform SDK) [79]. A compilation object is created from the syntax tree [85] that is compiled at runtime, avoiding time-consuming read and write operations in file-based compilation. (3) In the third step, if the compilation is successful, the code block is added to set C.

The described algorithm was implemented in the present study using the programming language C# and the mentioned Roslyn API. Several computational experiments were run, each time for one million iterations. Since the combination of code blocks is completely random and no selection toward a certain objective is added, it is not surprising that most of the generated code is not valid and generates a compiler error. However, in all of the conducted test runs, some valid code was generated. This included the declaration of variables (e.g., `char identifier98242 ;`), scope definitions (e.g., `{ }`) and combinations of the two (e.g., `char identifier98242 ; { }`).

The third question is how to implement a duality between code and data. We need to address this question because we want to execute the generated code as well as modify it. It also includes transferring the state of data such as the state of an entity from code to data and back to code again. Such transitioning between code and data allows implementing self-modifying operations on data structures, for example, the update function ϕ and update rules $\mathcal{U}$, where the current state of an entity is required to be transferred from code to data and after self-modification and running of the code, back again into code as the updated state of the entity. There are certain programming languages where program code is also represented as data and thus can be manipulated as such. It is a language property often referred to as homoiconic, and a prominent example is Lisp. However, it can also be achieved with C#. If, for example, we want to transfer the value of the variable `input` from code to data, we can use the `String.Replace` [86] and `String.ToString` [87] methods as follows:

```
code.Replace("input", input.ToString()).
```

Here, code is a string containing code as data, "input" represents the variable input in that code as data, and input represents the variable input in the code as code. The variable value of the latter is converted into a string with the String.ToString method, and then this value is inserted into the code as data by replacing "input" with the String.Replace method. We can then run the code as data (the string code) as described earlier in the self-modifying code prototype. Once we have executed the code at runtime, we also want to transfer back the output to our program, therefore from data to code. One way to achieve that in C# is to redirect the console output stream to a variable. We first set the output stream to a StringWriter object [88]. When the code as data (the string code) is executed, the value of the variable output is printed out in the console with the Console.WriteLine method [89] as follows:

```
Console.WriteLine(output).
```

Because of the redirection of the output stream, output is not printed in the console but stored in the StringWriter object. From there, we can use it in our program, and thus we have transferred data to code.

6.4 The Allagmatic Method as Guidance to Create Code Blocks

The system metamodel $\mathcal{SM}$ of the allagmatic method is defined as:

$$\mathcal{M}(\mathcal{E}, Q, \mathcal{M}, \mathcal{U}, \mathcal{A}, \mathcal{P}, \ldots, \hat{s}_s, \phi, \psi, \ldots, \hat{o}_o), \tag{3}$$

where $\mathcal{E}$ is the entities e-tuple, Q the set of possible entity states, $\mathcal{M}$ the milieus e-tuple, $\mathcal{U}$ the update rules u-tuple, $\mathcal{A}$ the adaptation rules a-tuple, $\mathcal{P}$ the adaptation end p-tuple, $\hat{s}_s$ are further structures, ϕ the update function, ψ the adaptation function, $\hat{o}_o$ are further operations, and $\hat{s}_i \in S \wedge \hat{o}_j \in O$ [71]. It intertwines structures and operations to create (complex) systems at the most abstract level in the virtual regime. Every such system contains at least the structures entities $\mathcal{E}$, entity states Q, milieus $\mathcal{M}$, update rules $\mathcal{U}$, adaptation rules $\mathcal{A}$, adaptation end $\mathcal{P}$, the operations update function ϕ and adaptation function ψ. As stated previously, to create an open-ended evolutionary system, we need the further structure lifetime parameter $\hat{s}_{\text{lt}}$ and a respective further operation $\hat{o}_{\text{lt}}$ to remove entities as soon as they have reached that time limit. We also need a simultaneous update of entity states and $\mathcal{U}$ and $\mathcal{M}$. Since the update function ϕ modifies entity states and the adaptation function ψ modifies $\mathcal{U}$ and $\mathcal{M}$, this is achieved by running ϕ and ψ in the same iteration. From the virtual regime, structures and operations are concretized, creating a metastable system in the metastable regime. In this concretization process, the self-modifying code prototype can now be used to generate concrete instances of the identified structures and operations. In the

following, a description of how this could be implemented is provided for each structure and operation:

- Entities $\mathcal{E}$ and their possible states Q: entities are concretized by defining the number of entities e and number and kind of possible states Q and k. By providing respective words, the self-modifying code is guided or restricted in its generation of code blocks. An entity is implemented as an object containing fields capturing all its states. For example, it is possible to randomly define the number of entities and states and based on that, randomly choose a data type for each field from predefined words implementing different data types. A list of length e is then created with the defined entity objects.
- Milieus $\mathcal{M}$: milieus are concretized by defining the number of connected entities m for each entity and the connections between entities forming a network structure. For example, it is possible to randomly define the number of connected entities for each entity and which entities are connected to it.
- Update function ϕ and update rules $\mathcal{U}$: update function and rules are concretized by defining what operations should be performed on which entity fields under which conditions. For example, it is possible to randomly define some Boolean operations for Boolean fields and arithmetic operations for integer and float fields. These operations are then coupled with some randomly defined if conditions choosing states of connected entities and relational operators.
- Adaptation function ψ, adaptation rules $\mathcal{A}$ and adaptation end P: adaptation function, rules and end are concretized by defining new update rules $\mathcal{U}$, milieus $\mathcal{M}$, and adaptation end $\mathcal{P}$. For example, it is possible to overwrite update rules with newly created rules, overwrite the network structure and randomly choose a value for each entity field to define the adaptation end.
- Lifetime parameter $\hat{s}_{\text{lt}}$ and respective operation $\hat{o}_{\text{lt}}$: lifetime parameter and operation are concretized by defining a time limit in terms of iterations and the way entities are removed. For example, it is possible to randomly define an integer value for the lifetime parameter and to decide whether entities are removed after they were present for the defined number of iterations or by randomly removing them after the defined number of iterations has passed.

Also, modification of code can be implemented with this approach by allowing certain parts of these defintions to be overwritten.

7. Discussion and Conclusion

Based on recent advances, the model building blocks *evolving entities*, *entity lifetime parameter*, *co-evolutionary operations of entities and environment* and *combinatorial interactions* are identified to

characterize open-ended evolutionary systems. These concepts led to punctuated equilibria in the co-evolutionary, combinatorial and critical evolution model (CCC model) model [14], which means that it never reaches an equilibrium state where the generation of further adaptive novelty is ceased and thus can be regarded as open ended. This study provides a formal description of a system metamodel for open-ended evolution according to the CCC model thus also capable of generating punctuated equilibria. It also provides a self-modifying code prototype in C# and guidance to create code blocks that potentially will allow an intrinsic implementation of open-ended evolutionary systems as suggested by Banzhaf et al. [6] and Taylor [19].

The proposed self-modifying code prototype and the guidance of the allagmatic method to create code blocks seem to be a promising way to change program code at runtime and potentially account for novelties. This is achieved by controlling the self-modification of code within abstractly defined building blocks of a system metamodel describing complex and evolutionary systems in general. It could thus be a way to interpret novelties without limiting possible solutions.

It is interesting to note that certain models anticipate changes that might occur to them. In all evolutionary systems, new entities arise and other entities disappear, which will not only change how many entities there are but also their interactions with each other and the environment. The CCC model [14] is capable of accounting for such changes in the model through co-evolution of entity states and interactions. On this level, it therefore does not need self-modification of the code but generic programming [73] of certain structures to dynamically adapt them to these changes.

In addition, the interpretation of concepts within a metaphysical framework as described with the allagmatic method provides a promising starting point to interpret novelty generated at runtime. This study provides a system metamodel of open-ended evolution and a prototype of self-modifying code implemented in C#. Using this prototype in the allagmatic method allows us to modify certain structures and operations of the system model and metamodel in a controlled way and potentially will allow us to interpret novelties in computational systems.

In conclusion, the identified model building blocks *evolving entities*, *entity lifetime parameter*, *co-evolutionary operations of entities and environment* and *combinatorial interactions* and the proposed self-modifying code provide a promising starting point to model and implement open-endedness in a computational system that potentially allows us to interpret novelties at runtime.

https://doi.org/10.25088/ComplexSystems.32.4.353

Acknowledgments

This work was supported by the Hasler Foundation under grant No. 21017. I thank Tom Van Dooren, Frietson Galis, Olivier Del Fabbro, Stefan Thurner, Sagi Nedunkanal and the members of the Complexity Club for their helpful comments on a draft of the manuscript.

References

[1] P. Christen, "Modelling and Implementing Open-Ended Evolutionary Systems," in *The Fourth Workshop on Open-Ended Evolution (OEE4), The 2021 Conference on Artificial Life (ALife 2021)*, 2021. workshops.alife.org/oee4/papers/christen-oee4-camera-ready.pdf.

[2] K. O. Stanley, "Why Open-Endedness Matters," *Artificial Life*, **25**(3), 2019 pp. 232–235. doi:10.1162/artl_a_00294.

[3] K. O. Stanley, J. Lehman and L. Soros. "Open-Endedness: The Last Grand Challenge You've Never Heard Of." O'Reilly Media, Inc. (Jan 16, 2024) www.oreilly.com/radar/open-endedness-the-last-grand-challenge-youve-never-heard-of.

[4] M. A. Bedau, J. S. McCaskill, N. H. Packard, S. Rasmussen, C. Adami, D. G. Green, T. Ikegami, K. Kaneko and T. S. Ray, "Open Problems in Artificial Life," *Artificial Life*, **6**(4), 2000 pp. 363–376. doi:10.1162/106454600300103683.

[5] M. A. Bedau, N. Gigliotti, T. Janssen, A. Kosik, A. Nambiar and N. Packard, "Open-Ended Technological Innovation," *Artificial Life*, **25**(1), 2019 pp. 33–49. doi:10.1162/artl_a_00279.

[6] W. Banzhaf, B. Baumgaertner, G. Beslon, R. Doursat, J. A. Foster, B. McMullin, V. V. de Melo, et al., "Defining and Simulating Open-Ended Novelty: Requirements, Guidelines, and Challenges," *Theory in Biosciences*, **135**(3), 2016 pp. 131–161. doi:10.1007/s12064-016-0229-7.

[7] S. Thurner, *Die Zerbrechlichkeit der Welt*, Wien: edition a, 2020.

[8] C. Ofria and C. O. Wilke, "Avida: A Software Platform for Research in Computational Evolutionary Biology," *Artificial Life*, **10**(2), 2004 pp. 191–229. doi:10.1162/106454604773563612.

[9] A. D. Channon and R. I. Damper, "Towards the Evolutionary Emergence of Increasingly Complex Advantageous Behaviours," *International Journal of Systems Science*, **31**(7), 2010 pp. 843–860. doi:10.1080/002077200406570.

[10] B. G. Woolley and K. O. Stanley, "A Novel Human-Computer Collaboration: Combining Novelty Search with Interactive Evolution," in *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation (GECCO '14)* (C. Igel, ed.), Vancouver, BC, Canada, New York: Association for Computing Machinery, 2014 pp. 233–240. doi:10.1145/2576768.2598353.

[11] D. H. Ackley and E. S. Ackley, "The *ulam* Programming Language for Artificial Life," *Artificial Life*, **22**(4), 2016 pp. 431–450. doi:10.1162/ARTL_a_00212.

[12] D. A. Marques, J. I. Meier and O. Seehausen, "A Combinatorial View on Speciation and Adaptive Radiation," *Trends in Ecology & Evolution*, **34**(6), 2019 pp. 531–544. doi:10.1016/j.tree.2019.02.008.

[13] S. Thurner, "A Simple General Model of Evolutionary Dynamics," *Principles of Evolution: From the Planck Epoch to Complex Multicellular Life* (H. Meyer-Ortmanns and S. Thurner, eds.), Berlin and Heidelberg: Springer, 2011 pp. 119–144. doi:10.1007/978-3-642-18137-5_4.

[14] S. Thurner, R. Hanel and P. Klimek, *Introduction to the Theory of Complex Systems*, New York: Oxford University Press, 2018.

[15] A. Adams, H. Zenil, P. C. W. Davies and S. I. Walker, "Formal Definitions of Unbounded Evolution and Innovation Reveal Universal Mechanisms for Open-Ended Evolution in Dynamical Systems," *Scientific Reports*, **7**(1), 2017 p. 997. doi:10.1038/s41598-017-00810-8.

[16] S. Hernández-Orozco, F. Hernández-Quiroz and H. Zenil, "Undecidability and Irreducibility Conditions for Open-Ended Evolution and Emergence," *Artificial Life*, **24**(1), 2018 pp. 56–70. doi:10.1162/ARTL_a_00254.

[17] N. Packard, M. A. Bedau, A. Channon, T. Ikegami, S. Rasmussen, K. O. Stanley and T. Taylor, "An Overview of Open-Ended Evolution: Editorial Introduction to the Open-Ended Evolution II Special Issue," *Artificial Life*, **25**(2), 2019 pp. 93–103. doi:10.1162/artl_a_00291.

[18] M. A. Boden, "Creativity and ALife," *Artificial Life*, **21**(3), 2015 pp. 354–365. doi:10.1162/ARTL_a_00176.

[19] T. Taylor, "Evolutionary Innovations and Where to Find Them: Routes to Open-Ended Evolution in Natural and Artificial Systems," *Artificial Life*, **25**(2), 2019 pp. 207–224. doi:10.1162/artl_a_00290.

[20] E. L. Dolson, A. E. Vostinar, M. J. Wiser and C. Ofria, "The MODES Toolbox: Measurements of Open-Ended Dynamics in Evolving Systems," *Artificial Life*, **25**(1), 2019 pp. 50–73. doi:10.1162/artl_a_00280.

[21] A. Channon, "Maximum Individual Complexity Is Indefinitely Scalable in Geb," *Artificial Life*, **25**(2), 2019 pp. 134–144. doi:10.1162/artl_a_00285.

[22] O. Rivoire, "Informations in Models of Evolutionary Dynamics," *Journal of Statistical Physics*, **162**(5), 2016 pp. 1324–1352. doi:10.1007/s10955-015-1381-z.

[23] T. Taylor, J. E. Auerbach, J. Bongard, J. Clune, S. Hickinbotham, C. Ofria, M. Oka, S. Risi, K. O. Stanley and J. Yosinski, "WebAL Comes of Age: A Review of the First 21 Years of Artificial Life on the Web," *Artificial Life*, **22**(3), 2016 pp. 364–407. doi:10.1162/artl_a_00211.

[24] W. B. Arthur, *Complexity and the Economy*, New York: Oxford University Press, 2015.

[25] W. B. Arthur, "How We Became Modern," *Sydney Brenner's 10-on-10: The Chronicles of Evolution* (S. Sim and B. Seet, eds.), Singapore: Wildtype Books, 2018.

[26] W. B. Arthur, *The Nature of Technology: What It Is and How It Evolves*, New York: Free Press, 2009.

[27] W. B. Arthur, "Where Darwin Doesn't Fit," *New Scientist*, **203**(2722), 2009 pp. 26–27. doi:10.1016/S0262-4079(09)62217-X.

[28] W. B. Arthur and W. Polak, "The Evolution of Technology within a Simple Computer Model," *Complexity*, **11**(5), 2006 pp. 23–31. doi:10.1002/cplx.20130.

[29] S. Thurner. "The Creative Destruction of Evolution," *Sydney Brenner's 10-on-10: The Chronicles of Evolution* (S. Sim and B. Seet, eds.), Singapore: Wildtype Books, 2018.

[30] S. Thurner, P. Klimek and R. Hanel, "Schumpeterian Economic Dynamics as a Quantifiable Model of Evolution," *New Journal of Physics*, **12**(7), 2010 075029. doi:10.1088/1367-2630/12/7/075029.

[31] P. Klimek, R. Hausmann and S. Thurner, "Empirical Confirmation of Creative Destruction from World Trade Data," *PLoS ONE*, **7**(6), 2012 e38924. doi:10.1371/journal.pone.0038924.

[32] P. Klimek, S. Thurner and R. Hanel, "Evolutionary Dynamics from a Variational Principle," *Physical Review E*, **82**(1), 2010 011901. doi:10.1103/PhysRevE.82.011901.

[33] R. Hanel, S. A. Kauffman and S. Thurner, "Phase Transition in Random Catalytic Networks," *Physical Review E*, **72**(3), 2005 036117. doi:10.1103/PhysRevE.72.036117.

[34] S. J. Gould and N. Eldredge, "Punctuated Equilibria: The Tempo and Mode of Evolution Reconsidered," *Paleobiology*, **3**(2), 1977 115–151. www.jstor.org/stable/2400177.

[35] J. A. Schumpeter, *Business Cycles: A Theoretical, Historical, and Statistical Analysis of the Capitalist Process*, New York: McGraw-Hill Book Company, Inc., 1939.

[36] T. Taylor, M. Bedau, A. Channon, D. Ackley, W. Banzhaf, G. Beslon, E. Dolson, et al., "Open-Ended Evolution: Perspectives from the OEE Workshop in York," *Artificial Life*, **22**(3), 2016 pp. 408–423. doi:10.1162/artl_a_00210.

[37] K. O. Stanley and J. Lehman, *Why Greatness Cannot Be Planned: The Myth of the Objective*, Cham: Springer, 2015.

[38] J. Lehman and K. O. Stanley, "Abandoning Objectives: Evolution through the Search for Novelty Alone," *Evolutionary Computation*, **19**(2), 2011 pp. 189–223. doi:10.1162/EVCO_a_00025.

[39] K. O. Stanley, "To Achieve Our Highest Goals, We Must Be Willing to Abandon Them," *ACM SIGPLAN Notices*, **45**(10), 2010 p. 3. doi:10.1145/1932682.1869541.

[40] R. Wang, J. Lehman, J. Clune and K. O. Stanley, "POET: Open-Ended Coevolution of Environments and Their Optimized Solutions," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '19)*, Prague, Czech Republic (M. López-Ibáñez, ed.), New York: Association for Computing Machinery, 2019 pp. 142–151. doi:10.1145/3321707.3321799.

[41] J. C. Brant and K. O. Stanley, "Minimal Criterion Coevolution: A New Approach to Open-Ended Search," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '17)*, Berlin Germany, New York: Association for Computing Machinery, 2017 pp. 67–74. doi:10.1145/3071178.3071186.

[42] J. K. Pugh, L. B. Soros, R. Frota, K. Negy and K. O. Stanley, "Major Evolutionary Transitions in the Voxelbuild Virtual Sandbox Game," in *The Fourteenth European Conference on Artificial Life (ECAL 2017)*, Lyon, France, Cambridge, MA: MIT Press, 2017 pp. 553–560. doi:10.1162/isal_a_088.

[43] E. Szathmáry, "Toward Major Evolutionary Transitions Theory 2.0," *Proceedings of the National Academy of Sciences*, **112**(33), 2015 pp. 10104–10111. doi:10.1073/pnas.1421398112.

[44] E. Szathmáry and J. M. Smith, "The Major Evolutionary Transitions," *Nature*, **374**(6519), 1995 pp. 227–232. doi:10.1038/374227a0.

[45] J. I. Meier, D. A. Marques, S. Mwaiko, C. E. Wagner, L. Excoffier and O. Seehausen, "Ancient Hybridization Fuels Rapid Cichlid Fish Adaptive Radiations," *Nature Communications*, **8**(1), 2017 14363. doi:10.1038/ncomms14363.

[46] O. Seehausen, "Speciation Affects Ecosystems," *Nature*, **458**(7242), 2009 1122–1123. doi:10.1038/4581122a.

[47] C. E. Wagner, L. J. Harmon and O. Seehausen, "Ecological Opportunity and Sexual Selection Together Predict Adaptive Radiation," *Nature*, **487**(7407), 2012 pp. 366–369. doi:10.1038/nature11144.

[48] M. E. Hochberg, P. A. Marquet, R. Boyd and A. Wagner, "Innovation: An Emerging Focus from Cells to Societies," *Philosophical Transactions of the Royal Society B: Biological Sciences*, **372**(1735), 2017 20160414. doi:10.1098/rstb.2016.0414.

[49] A. Wagner, *The Origins of Evolutionary Innovations: A Theory of Transformative Change in Living Systems*, New York: Oxford University Press, 2011.

[50] S.-R. Hosseini, O. C. Martin and A. Wagner, "Phenotypic Innovation through Recombination in Genome-Scale Metabolic Networks," *Proceedings of the Royal Society B: Biological Sciences*, **283**(1839), 2016 20161536. doi:10.1098/rspb.2016.1536.

[51] A. Wagner, "The Low Cost of Recombination in Creating Novel Phenotypes," *BioEssays*, **33**(8), 2011 pp. 636–646. doi:10.1002/bies.201100027.

[52] O. C. Martin and A. Wagner, "Effects of Recombination on Complex Regulatory Circuits," *Genetics*, **183**(2), 2009 pp. 673–684. doi:10.1534/genetics.109.104174.

[53] A. Barve and A. Wagner, "A Latent Capacity for Evolutionary Innovation through Exaptation in Metabolic Systems," *Nature*, **500**(7461), 2013 pp. 203–206. doi:10.1038/nature12301.

[54] N. H. Packard, "Intrinsic Adaptation in a Simple Model for Evolution," *Artificial Life* (C. G. Langton, ed.), Redwood City, CA: Addison-Wesley, 1989.

[55] L. Spector and A. Robinson, "Genetic Programming and Autoconstructive Evolution with the Push Programming Language," *Genetic Programming and Evolvable Machines*, **3**(1), 2002 pp. 7–40. doi:10.1023/A:1014538503543.

[56] F.-N. Demers and J. Malenfant, "Reflection in Logic, Functional and Object-Oriented Programming: A Short Comparative Study," in *Proceedings of the IJCAI '95 Workshop on Reflection and Metalevel Architectures and Their Applications in AI*, 1995 pp. 29–38.

[57] S. Stepney and S. Hickinbotham, "Innovation, Variation, and Emergence in an Automata Chemistry," in *ALife 2020*: *The 2020 Conference on Artificial Life*, Montreal, Canada (J. Bongard, J. Lovato, L. Soros and L. Hébert-Dufrésne, eds.), Cambridge, MA: MIT Press Direct, 2020 pp. 753–760. doi:10.1162/isal_a_00265.

[58] M. A. Fortuna, L. Zaman, C. Ofria and A. Wagner, "The Genotype-Phenotype Map of an Evolving Digital Organism," *PLOS Computational Biology*, **13**(2), 2017 e1005414. doi:10.1371/journal.pcbi.1005414.

[59] K. Sims, "Evolving 3D Morphology and Behavior by Competition," *Artificial Life*, **1**(4), 1994 pp. 353–372. doi:10.1162/artl.1994.1.4.353.

[60] S. Wolfram, "Cellular Automata as Models of Complexity," *Nature*, **311**(5985), 1984 pp. 419–424. doi:10.1038/311419a0.

[61] S. Wolfram, *A New Kind of Science*, Champaign, IL: Wolfram Media, Inc., 2002.

[62] S. Wolfram, "A Class of Models with the Potential to Represent Fundamental Physics," *Complex Systems*, **29**(2), 2020 pp. 107–536. doi:10.25088/ComplexSystems.29.2.107.

[63] S. Wolfram, *A Project to Find the Fundamental Theory of Physics*, Champaign, IL: Wolfram Media, Inc., 2020.

[64] P. Christen and O. Del Fabbro, "Cybernetical Concepts for Cellular Automaton and Artificial Neural Network Modelling and Implementation," in *2019 IEEE International Conference on Systems, Man and Cybernetics (SMC)*, Bari, Italy, Piscataway, NJ: IEEE, 2019 pp. 4124–4130. doi:10.1109/SMC.2019.8913839.

[65] P. Christen and O. Del Fabbro, "Automatic Programming of Cellular Automata and Artificial Neural Networks Guided by Philosophy," *New Trends in Business Information Systems and Technology* (R. Dornberger, ed.), Cham: Springer, 2021 pp. 131–146. doi:10.1007/978-3-030-48332-6_9.

[66] G. Simondon, *Individuation in Light of Notions of Form and Information* (T. Adkins, trans.), Minneapolis, MN: University of Minnesota Press, 2020.

[67] G. Simondon, *On the Mode of Existence of Technical Objects* (C. Malaspina and J. Rogove, trans.), Minneapolis, MN: University of Minnesota Press, 2016.

[68] D. Debaise, *Nature as Event: The Lure of the Possible* (M. Halewood, trans.), Durham, NC: Duke University Press, 2017.

[69] A. N. Whitehead, *Process and Reality: An Essay in Cosmology*, corrected ed. (D. R. Griffin and D. W. Sherburne, eds.), New York: Free Press, 1978.

[70] O. Del Fabbro, *Philosophieren mit Objekten: Gilbert Simondons prozessuale Individuationsontologie*, Frankfurt and New York: Campus Verlag, 2021.

[71] P. Christen and O. Del Fabbro, "Philosophy-Guided Mathematical Formalism for Complex Systems Modelling," in *2022 IEEE International Conference on Systems, Man and Cybernetics (SMC)*, Prague, Czech Republic, Piscataway, NJ: IEEE, 2022 pp. 2229–2236. doi:10.1109/SMC53654.2022.9945443.

[72] O. Del Fabbro and P. Christen, "Philosophy-Guided Modelling and Implementation of Adaptation and Control in Complex Systems," in *IEEE World Congress On Computational Intelligence (IEEE WCCI)*, Padua, Italy, Piscataway, NJ: IEEE, 2022. doi:10.1109/IJCNN55064.2022.9892833.

[73] K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Boston: Addison-Wesley, 2000.

[74] S. Fix, T. Probst, O. Ruggli, T. Hanne and P. Christen, "Open-Ended Automatic Programming through Combinatorial Evolution," in *Intelligent Systems Design and Applications, 21st International Conference on Intelligent Systems Design and Applications (ISDA 2021)*, (A. Abraham, N. Gandhi, T. Hanne, T.-P. Hong, T. N. Rios and W. Ding, eds.), Cham: Springer, 2022 pp. 1–12. doi:10.1007/978-3-030-96308-8_1.

[75] S. Fix, T. Probst, O. Ruggli, T. Hanne and P. Christen, "Automatic Programming as an Open-Ended Evolutionary System," *International Journal of Computer Information Systems and Industrial Management Applications*, **14**, 2022 pp. 204–212. mirlabs.org/ijcisim/regular_papers_2022/IJCISIM_18.pdf.

[76] N. Packard, M. A. Bedau, A. Channon, T. Ikegami, S. Rasmussen, K. Stanley and T. Taylor, "Open-Ended Evolution and Open-Endedness: Editorial Introduction to the Open-Ended Evolution I Special Issue," *Artificial Life*, **25**(1), 2019 pp. 1–3. doi:10.1162/artl_e_00282.

[77] D. Debaise, "What Is Relational Thinking?," *Inflexions*, **5**, 2012 pp. 1–11. www.inflexions.org/n5_Debaise.pdf.

[78] L. Soros, *Necessary Conditions for Open-Ended Evolution*, Ph.D. thesis, Department of Computer Science, University of Central Florida, 2018.

[79] The .NET Compiler Platform SDK. (Jan 19, 2024) docs.microsoft.com/en-gb/dotnet/csharp/roslyn-sdk.

[80] J. Skeet, *C# in Depth*, 4th ed., Shelter Island: Manning Publications Co., 2019.

[81] V. Kulkarni, *Theory of Computation*, New Delhi: Oxford University Press, 2013.

[82] R. Rojas, "Conditional Branching Is Not Necessary for Universal Computation in von Neumann Computers," *Journal of Universal Computer Science*, **2**(11), 1996 pp. 756–768. doi:10.3217/jucs-002-11-0756.

[83] "C# Keywords." The .NET C# Documentation. (Jan 19, 2024) docs.microsoft.com/en-gb/dotnet/csharp/language-reference/keywords.

[84] "Expressions." The .NET C# Documentation. (Jan 19, 2024) docs.microsoft.com/en-gb/dotnet/csharp/language-reference/language-specification/expressions.

[85] "CSharpSyntaxTree Class." The .NET API Documentation. (Jan 19, 2024) docs.microsoft.com/en-gb/dotnet/api/microsoft.codeanalysis.csharp.csharpsyntaxtree?view=roslyn-dotnet-4.1.0.

[86] "String.Replace Method." The .NET API Documentation. (Jan 19, 2024) docs.microsoft.com/en-gb/dotnet/api/system.string.replace?view=net-6.0#system-string-replace(system-string-system-string).

[87] "Type.ToString Method." The .NET API Documentation. (Jan 19, 2024) docs.microsoft.com/en-gb/dotnet/api/system.type.tostring?view=net-6.0#system-type-tostring.

[88] "StringWriter Class." The .NET API Documentation. (Jan 19, 2024) docs.microsoft.com/en-gb/dotnet/api/system.io.stringwriter?view=net-6.0.

[89] "Console.WriteLine Method." The .NET API Documentation. (Jan 19, 2024) docs.microsoft.com/en-gb/dotnet/api/system.console.writeline?view=net-6.0#system-console-writeline.