Composition Machines: Programming Self-Organising Software Models for the Emergence of Sequential Program Spaces

Damian Arellanes[0000−0002−0074−390X]

Lancaster University, Lancaster LA1 4WA, United Kingdom
damian.arellanes@lancaster.ac.uk

Abstract. We are entering a new era in which software systems are becoming more and more complex and larger. So, the composition of such systems is becoming infeasible by manual means. To address this challenge, self-organising software models represent a promising direction since they allow the (bottom-up) emergence of complex computational structures from simple rules. In this paper, we propose an abstract machine, called the composition machine, which allows the definition and the execution of such models. Unlike typical abstract machines, our proposal does not compute individual programs but enables the emergence of multiple programs at once. We particularly present the machine's semantics and demonstrate its operation with well-known rules from the realm of Boolean logic and elementary cellular automata.

Keywords: Automatic Software Composition · Self-Organising Software Models · Emergent Software · Applied Category Theory.

1 Introduction

Building programs that build programs is a long-standing challenge which has been considered the holy grail of Computer Science since the inception of Artificial Intelligence [15,23,31]. As software systems are becoming increasingly large and complex (e.g., Cyber-Physical Systems [6]), this challenge needs to be seriously considered for avoiding the infeasibility of manual composition of complex software at scale.

As a first step towards this challenge, we have introduced the notion of *composition by self-organisation* in which complex software models are not explicitly programmed, but emerge from simple rules in a decentralised manner [4]. In this paper, we propose a discrete-time abstract machine that fits into such a paradigm, called the *composition machine*, for the definition and the execution of *self-organising software models*. In this context, a software model is not a single program performing a specific functionality, but a space of programs evolving through well-defined, deterministic self-organisation rules. Thus, the goal of the composition machine is not the computation of a fixed functionality, but the time-dependent emergence of multiple programs at once. Multiple programs are encapsulated in spaces which are self-similar computational structures rooted in Category Theory. We adopt this formalism for leveraging the large corpus of existing theorems in categorical settings, particularly those related to composition properties.

The rest of the paper is structured as follows. Section 2 sets the context of the paper and discusses related work. Section 3 introduces preliminaries for the formal definition of the proposed machine. Section 4 presents the semantics of composition and program spaces. Section 5 describes the semantics of the composition machine. Section 6 presents an example in which complex program spaces emerge from simple rules. Finally, Section 7 presents future directions and final remarks.

2 Related Work

The work we propose in this paper is motivated by the problem of automated software composition, an active research problem that falls under the umbrella of program synthesis [15]. Program synthesis is a field that has aimed at providing methods, techniques and tools to automatically generate individual programs without explicitly coding them, usually from high-level specifications. Diverse algorithmic techniques to achive this task have been proposed over time such as inductive program synthesis [38], stochastic search [2] and constraint solving [19]. Unlike most of the work done in program synthesis, our intent is not to offer efficient algorithms to synthesize individual programs, but to provide an actual abstract machine able to automatically produce spaces of sequential programs at every step of its discrete-time operation process. Our abstract machine is implementation-independent and serves to capture the very essence of program emergence from simple rules. Accordingly, it just requires an initial configuration, a set of simple operation rules and a fixed number of elementary programs (from which complex sequential programs emerge over time).

Likewise, our work is related to automatic component composition (or what we call *synthesis-in-the-large*) [22] which consists on automatically finding optimal component-based configurations (or software architectures) that better satisfy specific (functional or non-functional) requirements such as reliability, response time or even end-user specifications. A vast number of approaches have been proposed over time to tackle this problem, ranging from AI planning [26] to evolutionary algorithms [18,30], especially for the realm of service-oriented computing. Apart from purely algorithmic solutions, self-adaptive component models [10,13,14,17,34] have also been proposed to enable structural manipulation of individual composite components under uncertain, dynamically changing environments. Unfortunately, dynamically changing a composite requires *ad hoc* algorithmic mechanisms that lie beyond component model semantics, e.g., a feedback control loop or an assembly-perception-learning framework. Some self-adaptive component models provide semantic constructs to dynamically switch between a small number of hardcoded composite variants [13,14]. These variants are not generated automatically but need to be manually coded by some designer. We are aware of only one component model [7] offering algebraic operators for semantically defining explicit spaces of sequential composites. Although individual composites are not created manually, the spaces that contain them are. To algorithmically collapse spaces into individual composites, such a model has been endowed with a MAPE-K controller [5] which, again, is "semantically intrusive" as it does not form part of the model semantics. In contrast to existing self-adaptive component models, the abstract machine we propose in this paper does not require any "intrusive entities" since the synthesis process is entirely governed by the operational semantics of the machine *per se*.

Like our abstract machine, self-assembly models of computation [37] are not intrusive in the sense they offer well-founded semantic constructs for the emergence of complex structures from simple rules. Algorithmic tile-based self-assembly [12] is the most representative example in this class of models of computation. Computing via self-assembly has its roots in the core principles of Cellular Automata (CAs). Generally speaking, CAs are discrete dynamical systems that exhibit complex global behaviour from a set of locally interacting components [36]. Network automata [8,28,32,35] form a particular class of CAs, whose aim is to encode fixed transition rules for the evolution of a graph configuration. In this way, complex graph structures can emerge over time and halting occurs when a certain (stable) configuration pattern oscillates *ad infinitum*.

Although they have some similarities, our abstract machine and CA-related approaches differ in their core underlying semantics and purpose. While a CA is just a collection of cells that form some pattern via well-defined rules, our machine defines a collection of cells that categorically represent some sequential computation each. So, the global state of our machine is equivalent to a *space of programs* that exist at some point in time. More precisely, the global state is a category that defines all the possible sequential compositions at a certain instant. So, unlike a CA, the purpose of our machine is not the computation of a single program but the emergence of entire spaces each containing a single or multiple sequential programs.

Kolmogorov machines [16] and storage modification machines [29] are also related to our work as they allow the dynamic evolution of connected graphs. However, like CAs, they only operate over time-varying graphs which can represent a single program each. Graph rewriting [25] also yields a model for computing over individual directed acyclic graphs via rule-based graph transformations. Their focus is on the computation of individual programs, not on on the emergence of program spaces.

3 Preliminaries

This section presents preliminaries in terms of *Category Theory* and *Quiver Theory* for the formal specification of the proposed machine. Our intention is not to provide a deeper analysis, but just to present the most important definitions and theorems for our purposes. For a deeper treatment of categorical foundations and quivers, we refer the reader to [9,11,24].

3.1 Category Theory

A category consists of a collection of objects, a collection of morphisms between objects, an identity morphism for each object and a way to compose morphisms. There are two laws that composition must satisfy: *associativity* and *unity*. Associativity yields the same composite no matter how composition operands are grouped, and unity leaves unchanged a non-identity morphism when it is composed with an identity morphism. To better understand the semantics of a category, let us describe an example:

Definition 1 (Category of Sets). *$\mathscr{S}$ is the category where:*

- *$\mathscr{S}_0$ is a collection of sets, called objects,*
- *$\mathscr{S}_1$ is a collection of functions between sets, called morphisms,*
- *$dom\colon \mathscr{S}_1 \to \mathscr{S}_0$ is a collection morphism that takes each function $f \in \mathscr{S}_1$ to its domain $dom(f) \in \mathscr{S}_0$,*
- *$cod\colon \mathscr{S}_1 \to \mathscr{S}_0$ is a collection morphism that takes each function $f \in \mathscr{S}_1$ to its codomain $cod(f) \in \mathscr{S}_0$,*
- *there is an identity function $1_X\colon X \to X$ for each set $X \in \mathscr{S}_0$ such that $1_X \in \mathscr{S}_1$,*
- *there is a composite function $g \circ f \in \mathscr{S}_1$ for every pair $(f, g) \in \mathscr{S}_1 \times \mathscr{S}_1$ satisfying $cod(f) = dom(g)$,*
- composition is associative*: $\forall (f, g, h) \in \mathscr{S}_1 \times \mathscr{S}_1 \times \mathscr{S}_1, (h \circ g) \circ f = h \circ (g \circ f) \iff cod(f) = dom(g) \ \wedge \ cod(g) = dom(h)$, and*
- composition satisfies unity*: $\forall (f\colon X \to Y) \in \mathscr{S}_1, 1_Y \circ f = f = f \circ 1_X$.*

Remark 1. We say that a category is *small* if and only if it is internal in $\mathscr{S}$.

To avoid set-theoretic issues related to the Russell-Zermelo paradox [27], Category Theory defines a category of categories, Cat, where objects are small categories and morphisms are functors. A functor is a structure-preserving mapping between two categories, which assigns the objects of one category to the objects of another one. It also assigns the morphisms of one category to the morphisms of the other. Functors preserve both identity morphisms and composition of morphisms.

Definition 2 (Functor). *A functor $F\colon \mathscr{D} \to \mathscr{E}$ assigns each object $a \in \mathscr{D}_0$ to an object $F(a) \in \mathscr{E}_0$ and each morphism $f \in \mathscr{D}_1$ to a morphism $F(f) \in \mathscr{E}_1$ such that:*

- *$F(1_a) = 1_{F(a)}$ for each object $a \in \mathscr{D}_0$, and*
- *$F(g \circ f) = F(g) \circ F(f)$ for all composable morphisms $f, g \in \mathscr{D}_1$.*

Definition 3 (Natural Transformation). *Let $F, G\colon \mathscr{D} \to \mathscr{E}$ be two functors each from the category $\mathscr{D}$ to the category $\mathscr{E}$. A natural transformation $\eta\colon F \to G$ is a morphism between F and G, which satisfies the following:*

- *for every object $a \in \mathscr{D}_0$, there is a morphism $\eta_a\colon F(a) \to G(a)$ between objects of $\mathscr{E}$, and*
- *for every morphism $(f\colon a \to b) \in \mathscr{D}_1$, we have $\eta_b \circ F(f) = G(f) \circ \eta_a$.*

3.2 Quiver Foundations

The underlying structure of a category is typically represented as a *quiver* which is a directed multigraph with loops allowed. The term quiver is used among category theorists to avoid confusions derived from the multiple meanings of the word *graph*. The formal definition of a quiver is presented below, along with other related formalisms. Whenever convenient, we treat a function as a set of relations.

Notation 1 (Walking Quiver Category) *Let $\mathscr{Q}$ be the walking quiver category consisting of a collection $\mathscr{Q}_0$ of vertices, a collection $\mathscr{Q}_1$ of arrows, a source morphism $s\colon \mathscr{Q}_1 \to \mathscr{Q}_0$ and a target morphism $\tau\colon \mathscr{Q}_1 \to \mathscr{Q}_0$.*

Definition 4 (Quiver). *A* quiver *Q is a functor $\mathscr{Q} \to \mathscr{S}$. Intuitively, it is a quadruple (Q_0, Q_1, s, τ) where Q_0 is a set of vertices, Q_1 is a set of arrows, $s\colon Q_1 \to Q_0$ is a source function and $\tau\colon Q_1 \to Q_0$ is a target function. If an arrow $\alpha \in Q_1$ has source vertex $x \in Q_0$ and target vertex $y \in Q_0$, we say that α is directed from x to y. This is denoted $\alpha\colon x \to y$ with $x = s(\alpha)$ and $y = \tau(\alpha)$. We call Q finite if and only if the sets Q_0 and Q_1 are both finite.*

Definition 5 (Subquiver). *We say that $Q' = (Q'_0, Q'_1, s', \tau')$ is a* subquiver *of $Q = (Q_0, Q_1, s, \tau)$, written $Q' \subseteq Q$, if and only if $Q'_0 \subseteq Q_0$, $Q'_1 \subseteq Q_1$, $s' = s|_{Q'_1}$ and $\tau' = \tau|_{Q'_1}$.*

Definition 6 (Non-Trivial Path). *A* non-trivial path *$\rho = (\alpha_n, \alpha_{n-1}, \ldots, \alpha_1)$ in a quiver Q is a finite sequence of arrows such that $n \geq 1$ is the length of ρ, $\alpha_i \in Q_1$ for all $i \in \mathbb{N} \cap [1, n]$ and $\tau(\alpha_j) = s(\alpha_{j+1})$ for all $j \in \mathbb{N} \cap [1, n-1]$. The source vertex of ρ is denoted by $s(\rho)$ and the target vertex by $\tau(\rho)$ so that $s(\rho) = s(\alpha_1)$ and $\tau(\rho) = \tau(\alpha_n)$ (clearly an abuse of notation but convenient). By convention, all paths are read from right to left as in function composition, and the same arrow can occur more than once in ρ.*

Definition 7 (Trivial Path). *A* trivial path *ρ_x in a quiver Q is just a vertex $x \in Q_0$. Its length is 0 and there are as many trivial paths as there are vertices in Q_0.*

Definition 8 (Path Concatenation). *Let ρ_1 and ρ_2 be two paths. We say that $\rho_2 * \rho_1$ is a new path, called a path concatenation, if and only if $\tau(\rho_1) = s(\rho_2)$. Unambiguously, the source of $\rho_2 * \rho_1$ is denoted by $s(\rho_2 * \rho_1)$ and its target by $\tau(\rho_2 * \rho_1)$ such that $s(\rho_2 * \rho_1) = s(\rho_1)$ and $\tau(\rho_2 * \rho_1) = \tau(\rho_2)$.*

Definition 9 (Set of Paths). *By convention, Q_m is the set of paths of length $m \geq 0$ in a quiver Q, and $Q_* = \bigcup_{m \in \mathbb{N}} Q_m$ is the set of all the possible paths in Q.*

Theorem 1 ([11]). *Given a quiver Q, Q_* is finite if and only if Q is finite and has no oriented cycles.*

Definition 10 (Path Category). *The* path category *$P(Q)$ on a quiver Q consists of:*

– *a collection $P(Q)_0$ whose objects are the vertices in Q_0,*
– *a collection $P(Q)_1$ whose morphisms are the paths in Q_*,*
– *a collection morphism $s\colon P(Q)_1 \to P(Q)_0$ that maps each path $\rho \in P(Q)_1$ to its source vertex $s(\rho) \in P(Q)_0$,*
– *a collection morphism $\tau\colon P(Q)_1 \to P(Q)_0$ that maps each path $\rho \in P(Q)_1$ to its target vertex $\tau(\rho) \in P(Q)_0$,*
– *a trivial path $1_x \in P(Q)_1$ for every vertex $x \in P(Q)_0$, called the identity of x, and*
– *a path concatenation $\rho_2 * \rho_1 \in P(Q)_1$ for every pair $(\rho_1, \rho_2) \in P(Q)_1 \times P(Q)_1$ satisfying $\tau(\rho_1) = s(\rho_2)$.*

Remark 2. The category $P(Q)$ is sound because:

– *composition is associative*: $\forall(\rho_1, \rho_2, \rho_3) \in P(Q)_1 \times P(Q)_1 \times P(Q)_1, (\rho_3 * \rho_2) * \rho_1 = \rho_3 * (\rho_2 * \rho_1) \iff \tau(\rho_1) = s(\rho_2) \ \wedge\ \tau(\rho_2) = s(\rho_3)$, and
– *composition satisfies unity*: $\forall \rho \in P(Q)_1, 1_{\tau(\rho)} * \rho = \rho = \rho * 1_{s(\rho)}$.

Remark 3. Given a path category $P(Q)$, the composition of two morphisms in $P(Q)_1$ is defined by the concatenation of two paths in Q_*, and an identity path $1_x \in P(Q)_1$ corresponds to a trivial path $\rho_x \in Q_0$. Also, every path $\rho \in Q_1$ is mapped to its one morphism in $P(Q)_1$.

Theorem 2 ([9]). *Let $\mathscr{S}^{\mathscr{Q}}$ be the category where objects are functors $Q\colon \mathscr{Q} \to \mathscr{S}$ (i.e., quivers) and morphisms are natural transformations between such functors. Then, there is a functor $P\colon \mathscr{S}^{\mathscr{Q}} \to Cat$ that sends each quiver Q to its corresponding path category $P(Q)$, by taking each vertex $x \in Q_0$ to an object $P(x) \in P(Q)_0$ and each path $\rho \in Q_*$ to a morphism $P(\rho) \in P(Q)_1$.*

4 Semantics of Composition and Program Spaces

In this section, we present the semantics of composition and program spaces for the rest of the paper. For this, we introduce the notion of a *computon* which is the fundamental unit of computation in our proposal.[1] Informally, a computon is a sequential program that produces exactly one output value for a given input value via some finite computation. A value is an element of some set referred to as data type.[2] Formally:

Definition 11 (Data Type). *A data type d is a set of values equipped with at least one* k-ary *operation $d^k \to d$ that can be performed on those values.*

[1] The word *computon* derives from the Latin root for computation (*computus*) and the Greek suffix *-on*. In Physics, this suffix is traditionally used to designate subatomic particle names.

[2] Examples of data types include the set of integers and the set of real numbers [21,33]. A detailed overview of data types is out of the scope of this paper. This is because the definition of type systems depends on the application domain.

Definition 12 (Computon). *A* computon $f\colon d_1 \to d_2$ *is a function from an input data type* d_1 *to an output data type* d_2*. We say that it is an identity computon if* $d_1 = d_2$ *and* $\forall v \in d_1, f(v) = v$*. Otherwise,* f *is a non-identity computon.*

Definition 13 (Composite Computon). *A* composite computon $f_2 \circ f_1$ *is a higher-order function given by the composition of computons* f_1 *and* f_2 *where* $cod(f_1) = dom(f_2)$*.*

Definition 14 (Computon Category). *A* computon category $\mathscr{C}$ *consists of:*

- *a collection* $\mathscr{C}_0$ *of data types,*
- *a collection* $\mathscr{C}_1$ *of computons,*
- *a morphism* $in\colon \mathscr{C}_1 \to \mathscr{C}_0$ *that takes each computon* $f \in \mathscr{C}_1$ *to an input data type* $d \in \mathscr{C}_0$*,*
- *a morphism* $out\colon \mathscr{C}_1 \to \mathscr{C}_0$ *that takes each computon* $f \in \mathscr{C}_1$ *to an output data type* $d \in \mathscr{C}_0$*,*
- *a composite computon* $f_2 \circ f_1 \in \mathscr{C}_1$ *for every pair* $(f_1, f_2) \in \mathscr{C}_1 \times \mathscr{C}_1$ *satisfying* $out(f_1) = in(f_2)$*, and*
- *an identity computon* $1_d \in \mathscr{C}_1$ *for every data type* $d \in \mathscr{C}_0$*,*

such that composition of computons*:*

- is associative*:* $\forall (f_3, f_2, f_1) \in \mathscr{C}_1 \times \mathscr{C}_1 \times \mathscr{C}_1, (f_3 \circ f_2) \circ f_1 = f_3 \circ (f_2 \circ f_1) \iff out(f_1) = in(f_2) \;\wedge\; out(f_2) = in(f_3)$*, and*
- satisfies unity*:* $\forall (f\colon d_1 \to d_2) \in \mathscr{C}_1,\ 1_{d_2} \circ f = f = f \circ 1_{d_1}$*.*

Definition 15 (Program Space). *Given a subcategory* $\mathscr{C}'$ *of a computon category* $\mathscr{C}$*, let* $R\colon \mathscr{C}' \to \mathscr{S}$ *be an injective functor which takes each data type* $d \in \mathscr{C}'_0$ *to its underlying set* $R(d) \in \mathscr{S}_0$ *and each computon* $f \in \mathscr{C}'_1$ *to a function* $R(f) \in \mathscr{S}_1$*. A* program space *is the subcategory* $R(\mathscr{C}')$ *of* $\mathscr{S}$*, where* $R(\mathscr{C}')$ *denotes the image of* $\mathscr{C}'$ *under* R*.*

Intuitively, a program space $R(\mathscr{C}')$ on a computon category $\mathscr{C}'$ is a quadruple $(R(\mathscr{C}')_0, R(\mathscr{C}')_1, in, out)$ where $R(\mathscr{C}')_0$ is a set of data types, $R(\mathscr{C}')_1$ is a set of computons, $in\colon R(\mathscr{C}')_1 \to R(\mathscr{C}')_0$ is an input function and $out\colon R(\mathscr{C}')_1 \to R(\mathscr{C}')_0$ is an output function.

Definition 16 (Transformer). *A* transformer $T\colon P(Q) \to \mathscr{C}$ *is a functor for presenting the path category on a quiver* Q *in a computon category* $\mathscr{C}$*. It takes the collection* $P(Q)_0$ *of vertices to a collection* $\mathscr{C}_0$ *of data types and the collection* $P(Q)_1$ *of paths to a collection* $\mathscr{C}_1$ *of computons such that:*

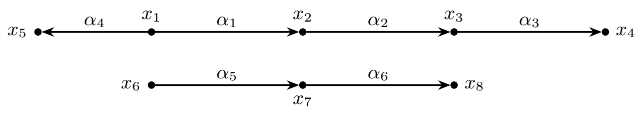
- *there is an identity computon* $1_{T(x)} \in \mathscr{C}_1$ *for each vertex* $x \in P(Q)_0$ *where* $T(x) \in \mathscr{C}_0$*, and*
- *there is a composite computon* $T(\rho_n * \cdots * \rho_1) \in \mathscr{C}_1$ *for every path concatenation* $\rho_n * \ldots * \rho_1 \in P(Q)_1$ *such that* $T(\rho_n * \cdots * \rho_1) = T(\rho_n) \circ \cdots \circ T(\rho_1)$ *with* $T(\rho_1), \ldots, T(\rho_n) \in \mathscr{C}_1$*.*

Notation 2 *$(T \circ P)(Q)$ denotes a computon category generated from the path category on a quiver Q. Its collection of data types is denoted by $(T \circ P)(Q)_0$ and its collection of computons by $(T \circ P)(Q)_1$.*

Definition 17. *From Definitions 15 and 16, we can specify a composite functor $R \circ T \colon P(Q) \to \mathscr{S}$ for transforming the path category on a quiver Q into its corresponding program space $(R \circ T \circ P)(Q)$. For short, we refer $(R \circ T \circ P)(Q)$ to as the program space on Q.*

Example 1. This example is a walkthrough for the construction of the program space $(R \circ T \circ P)(Q)$ on the quiver $Q = (Q_0, Q_1, s, \tau)$. This quiver is illustrated in Fig. 1 and is defined as follows:

- $Q_0 = \{x_i \mid i \in \mathbb{N} \cap [1, 8]\}$,
- $Q_1 = \{\alpha_i \mid i \in \mathbb{N} \cap [1, 6]\}$,
- $s = \{(\alpha_1, x_1), (\alpha_2, x_2), (\alpha_3, x_3), (\alpha_4, x_1), (\alpha_5, x_6), (\alpha_6, x_7)\}$, and
- $\tau = \{(\alpha_1, x_2), (\alpha_2, x_3), (\alpha_3, x_4), (\alpha_4, x_5), (\alpha_5, x_7), (\alpha_6, x_8)\}$.



$$\begin{tikzcd}
x_5 & x_1 \arrow[l, "\alpha_4"'] \arrow[r, "\alpha_1"] & x_2 \arrow[r, "\alpha_2"] & x_3 \arrow[r, "\alpha_3"] & x_4 \\
 & x_6 \arrow[r, "\alpha_5"] & x_7 \arrow[r, "\alpha_6"] & x_8 &
\end{tikzcd}$$

Fig. 1: Graphical representation of the quiver Q described in Example 1.

Once the quiver Q has been defined, we can transform it into a computon category as follows. Let $D = \{d_i \mid i \in \mathbb{N} \cap [1, 8]\}$ be a set of data types and $F = \{f_j \mid j \in \mathbb{N} \cap [1, 6]\}$ be a set of computons with $f_1 \colon d_1 \to d_2$, $f_2 \colon d_2 \to d_3$, $f_3 \colon d_3 \to d_4$, $f_4 \colon d_1 \to d_5$, $f_5 \colon d_6 \to d_7$ and $f_6 \colon d_7 \to d_8$. According to Definition 16, we can specify a transformer T to present $P(Q)$ in a computon category $(T \circ P)(Q)$. We accomplish this by mapping each vertex $P(x_i) \in P(Q)_0$ to a data type $d_i \in D$ for all $i \in \mathbb{N} \cap [1, 8]$ and each path $P(\alpha_j) \in P(Q)_1$ of length one to a computon $f_j \in F$ for all $j \in \mathbb{N} \cap [1, 6]$. Particularly, composite computons correspond to path concatenations, and identity computons are trivial paths. This construction is formally defined as follows:

- $\forall i \in \mathbb{N} \cap [1, 8], x_i \in Q_0 \implies T(P(x_i)) \in (T \circ P)(Q)_0$ such that $P(x_i) \in P(Q)_0$ and $T(P(x_i)) = d_i \in D$,
- $\forall i \in \mathbb{N} \cap [1, 8], x_i \in Q_0 \implies T(1_{P(x_i)}) \colon T(P(x_i)) \to T(P(x_i))$ such that $1_{P(x_i)} \in P(Q)_1$, $T(1_{P(x_i)}) \in (T \circ P)(Q)_1$, $P(x_i) \in P(Q)_0$, $T(P(x_i)) \in (T \circ P)(Q)_0$ and $T(P(x_i)) = d_i \in D$,
- $\forall i \in \mathbb{N} \cap [1, 6], \alpha_i \in Q_1 \implies T(P(\alpha_i)) \in (T \circ P)(Q)_1$ such that $P(\alpha_i) \in P(Q)_1$ and $T(P(\alpha_i)) = f_i \in F$, and
- $\forall (\rho_n * \cdots * \rho_1) \in P(Q)_1, [T(\rho_n) \circ \cdots \circ T(\rho_1)] \in (T \circ P)(Q)_1$.

Finally, the resulting computon category $(T \circ P)(Q)$ can be presented in $\mathscr{S}$ via the functor R (see Definition 15). Intuitively, this is the program space $(R \circ T \circ P)(Q) = ((R \circ T \circ P)(Q)_0, (R \circ T \circ P)(Q)_1, in, out)$ where:

- $(R \circ T \circ P)(Q)_0 = D$,
- $(R \circ T \circ P)(Q)_1 = F \cup \{f_2 \circ f_1, f_3 \circ f_2, f_3 \circ f_2 \circ f_1, f_6 \circ f_5, 1_{d_1}, 1_{d_2}, 1_{d_3}, 1_{d_4}, 1_{d_5}, 1_{d_6}, 1_{d_7}, 1_{d_8}\}$ such that $f_2 \circ f_1 \colon d_1 \to d_3$, $f_3 \circ f_2 \colon d_2 \to d_4$, $f_3 \circ f_2 \circ f_1 \colon d_1 \to d_4$, $f_6 \circ f_5 \colon d_6 \to d_8$ and $1_{d_i} \colon d_i \to d_i$ for all $i \in \mathbb{N} \cap [1, 8]$,
- $in = \{(f_1, d_1), (f_2, d_2), (f_3, d_3), (f_4, d_1), (f_5, d_6), (f_6, d_7), (f_2 \circ f_1, d_1), (f_3 \circ f_2, d_2), (f_3 \circ f_2 \circ f_1, d_1), (f_6 \circ f_5, d_6), (1_{d_1}, d_1), (1_{d_2}, d_2), (1_{d_3}, d_3), (1_{d_4}, d_4), (1_{d_5}, d_5), (1_{d_6}, d_6), (1_{d_7}, d_7), (1_{d_8}, d_8)\}$, and
- $out = \{(f_1, d_2), (f_2, d_3), (f_3, d_4), (f_4, d_5), (f_5, d_7), (f_6, d_8), (f_2 \circ f_1, d_3), (f_3 \circ f_2, d_4), (f_3 \circ f_2 \circ f_1, d_4), (f_6 \circ f_5, d_8), (1_{d_1}, d_1), (1_{d_2}, d_2), (1_{d_3}, d_3), (1_{d_4}, d_4), (1_{d_5}, d_5), (1_{d_6}, d_6), (1_{d_7}, d_7), (1_{d_8}, d_8)\}$.

The program space $(R \circ T \circ P)(Q)$ is illustrated in Fig. 2. Although we omit identity computons for clarity, the above definition reveals that the identities are elements of the set $(R \circ T \circ P)(Q)_1$. Without loss of generality, hereafter we do not show identity computons when depicting program spaces.

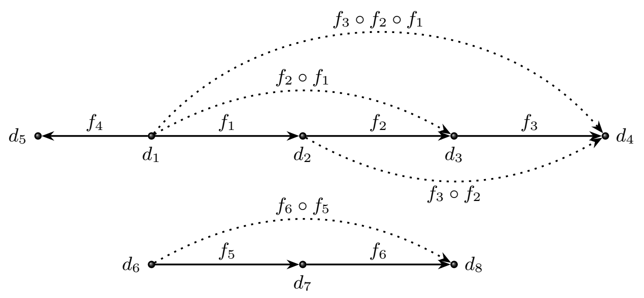


Fig. 2: Graphical representation of the program space $(R \circ T \circ P)(Q)$ described in Example 1.

5 Composition Machines

We propose the notion of composition machines for the emergence of sequential program spaces. These machines allow the definition and the execution of self-organising software models [4], and represent a shift from the computation of single programs to the emergence of program spaces.

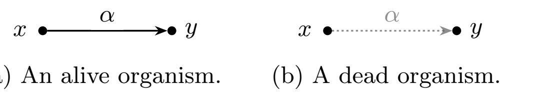
Definition 18 (Composition Machine). *A* composition machine *M is a septuple $(D, F, Q, \mu, S, N, \delta)$ where:*

- *D is a non-empty finite set of data types,*
- *F is a non-empty finite set of computons such that $\forall f, g \in F, cod(f) = cod(g) \iff f = g$ and $\forall f \in F, dom(f) \neq cod(f)$,*
- *Q is an acyclic quiver where each vertex $x \in Q_0$ represents a data type $d \in D$ and each arrow $\alpha \in Q_1$ (called an organism) represents a computon $f \in F$ such that $\forall \alpha_1, \alpha_2 \in Q_1, \tau(\alpha_1) = \tau(\alpha_2) \iff \alpha_1 = \alpha_2$ and $\forall \alpha \in Q_1, s(\alpha) \neq \tau(\alpha)$,*

- *μ consists of a pair of bijective functions, $\mu_0 \colon Q_0 \to D$ and $\mu_1 \colon Q_1 \to F$, for mapping vertices to data types and organisms to computons, respectively, such that $\mu_1(\alpha) = f \iff \mu_0(s(\alpha)) = dom(f) \ \wedge \ \mu_0(t(\alpha)) = cod(f)$,*
- *$S = \{0,1\}$ is the set of possible organism states,*
- *$N = \{N_1, N_2, N_3, N_4\}$ is a non-empty finite collection of neighbourhoods:*
  - *$N_1 \subseteq Q_1$ is a set where each element $(\alpha) \in N_1$ is the unary neighbourhood of an isolated organism $\alpha \in Q_1$ that has has no neighbours to the right and no neighbours to the left, i.e., $\forall(\alpha) \in N_1, \nexists \alpha_0 \in Q_1, \tau(\alpha_0) = s(\alpha) \ \vee \ \tau(\alpha) = s(\alpha_0)$.*
  - *$N_2 \subseteq Q_1 \times Q_1$ is a set where each element $(\alpha_1, \alpha_2) \in N_2$ is the binary neighbourhood of an organism $\alpha_1 \in Q_1$ that has only one neighbour to the right, i.e., $\forall(\alpha_1, \alpha_2) \in N_2, (\nexists \alpha_0 \in Q_1, \tau(\alpha_0) = s(\alpha_1)) \ \wedge \ \tau(\alpha_1) = s(\alpha_2)$.*
  - *$N_3 \subseteq Q_1 \times Q_1$ is a set where each element $(\alpha_1, \alpha_2) \in N_3$ is the binary neighbourhood of an organism $\alpha_2 \in Q_1$ that has only one neighbour to the left, i.e., $\forall(\alpha_1, \alpha_2) \in N_3, (\nexists \alpha_0 \in Q_1, \tau(\alpha_2) = s(\alpha_0)) \ \wedge \ \tau(\alpha_1) = s(\alpha_2)$.*
  - *$N_4 \subseteq Q_1 \times Q_1 \times Q_1$ is a set where each element $(\alpha_1, \alpha_2, \alpha_3) \in N_4$ is the ternary neighbourhood of an organism $\alpha_2 \in Q_1$ which has exactly one neighbour to the right and exactly one neighbour to the left, i.e., $\forall(\alpha_1, \alpha_2, \alpha_3) \in N_4, \tau(\alpha_1) = s(\alpha_2) \ \wedge \ \tau(\alpha_2) = s(\alpha_3)$.*
- *δ consists of four local state transition functions:*
  - *$\delta_1 \colon S \to S$ for every $\alpha \in Q_1$ with a neighbourhood $(\alpha) \in N_1$,*
  - *$\delta_2 \colon S^2 \to S$ for every $\alpha_1 \in Q_1$ with a neighbourhood $(\alpha_1, \alpha_2) \in N_2$,*
  - *$\delta_3 \colon S^2 \to S$ for every $\alpha_2 \in Q_1$ with a neighbourhood $(\alpha_1, \alpha_2) \in N_3$, and*
  - *$\delta_4 \colon S^3 \to S$ for every $\alpha_2 \in Q_1$ with a neighbourhood $(\alpha_1, \alpha_2, \alpha_3) \in N_4$.*

Intuitively, a composition machine $M = (D, F, Q, \mu, S, N, \delta)$ consists of $|F|$ *organisms* operating in discrete time.[3] Each organism represents a computon $f \in F$ and has at most three neighbours (including itself). We say that two different organisms $\alpha_1 \in Q_1$ and $\alpha_2 \in Q_1$ are neighbours if there is a non-trivial path (α_1, α_2) or a non-trivial path (α_2, α_1). Equivalently, two different organisms are neighbours if the computons they represent are composable (see Definition 13).

At each time step $t \in \mathbb{N}$, an organism $\alpha \in Q_1$ is in a state $c(\alpha)^t \in S$. If $c(\alpha)^t = 1$, we say that α is alive; otherwise, we say that α is dead (see Fig. 3).



(a) An alive organism. (b) A dead organism.

Fig. 3: At time t, an organism α is (a) alive if $c(\alpha)^t = 1$ or (b) dead if $c(\alpha)^t = 0$.

The configuration of M at t is then a function $c \colon Q_1 \to S$ that assigns a state to each organism $\alpha \in Q_1$. Intuitively, it is a snapshot of all the states in the system of organisms at some moment in time. Initially, at $t = 0$, M is in the so-called *initial configuration* and, in subsequent steps, the states of its organisms are updated in parallel according to four δ functions.

A δ function is a local update rule that defines the next state of an organism based on the current state of n neighbours. More concretely, if the neighbours

[3] As the function μ_1 is bijective (see Definition 18), we have that $|F| = |Q_1|$.

of an organism have states $c(\alpha_1)^t, c(\alpha_2)^t, \ldots, c(\alpha_n)^t$ at t, then the state of that organism at $t+1$ is given by $\delta_i(c(\alpha_1)^t, c(\alpha_2)^t, \ldots, c(\alpha_n)^t)$ such that $i \in \mathbb{N} \cap [1, 4]$ and $n \in \mathbb{N} \cap [1, 3]$. Particularly, $n = 1$ for unary neighbourhoods, $n = 2$ for binary neighbourhoods and $n = 3$ for ternary neighbourhoods.

Moreover, as an organism can be alive or dead, there are 2^1, 2^2 and 2^3 possible patterns for unary, binary and ternary neighbourhoods, respectively (see Fig. 4). So, $|dom(\delta_1)| = |S| = 2$, $|dom(\delta_2)| = |dom(\delta_3)| = |S|^2 = 4$ and $|dom(\delta_4)| = |S|^3 = 8$. Accordingly, there are $2^2 \times 2^4 \times 2^4 \times 2^8 = 2^{18}$ possible rules for a composition machine.

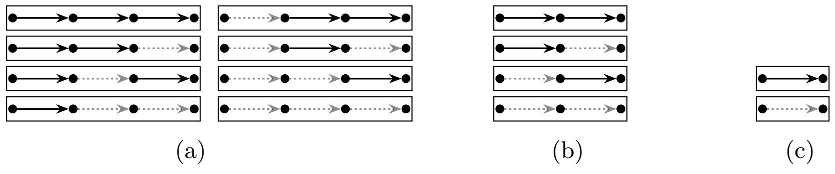


Fig. 4: Possible patterns for (a) ternary neighbourhoods, (b) binary neighbourhoods and (c) unary neighbourhoods.

To explain how rules can be defined, let us consider a simple transition function for organisms that have only one neighbour to the right:

$$\delta_2(c(\alpha_1)^t, c(\alpha_2)^t) = c(\alpha_1)^t \oplus c(\alpha_2)^t \tag{1}$$

where $\oplus$ is the exclusive-or (XOR) operator defined in Boolean algebra.

Without loss of generality, let us assume $(\alpha_1, \alpha_2) \in N_2$ is the binary neighbourhood of some organism α_1. If both organisms are alive at time t, then the next state of α_1 would be given by $c(\alpha_1)^{t+1} = \delta_2(1, 1) = 0$, as shown in Fig. 5.

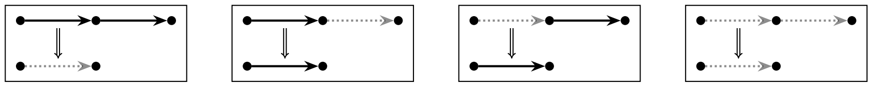


Fig. 5: An example of a local transition rule for an organism that has only one neighbour to the right. This rule is equivalent to an XOR operation and corresponds to the δ_2 function described in Equation 1.

Fig. 5 shows an example of a local transition rule for the binary neighbourhood N_2 in some machine M. In a similar fashion, we can define rules for the other neighbourhoods to independently transform organisms' states. These transformations occur by simultaneously applying the appropriate rule for each $\alpha \in Q_1$, leading to a time evolution of the global configuration of M. Particularly, evolving a configuration c into another one is given by a global transition function $G\colon S^{Q_1} \to S^{Q_1}$.[4] So, the time evolution of M results from the repeated application of G, i.e., $c \mapsto G(c) \mapsto (G \circ G)(c) \mapsto (G \circ G \circ G)(c) \mapsto \ldots$

As c is the initial configuration, we can deduce that the machine's orbit $orb(c)$ is equal to $c, G(c), (G \circ G)(c), (G \circ G \circ G)(c), \ldots$[5] In this case, time refers

[4] S^{Q_1} is the set of all the possible configurations in a given machine, i.e., the set of all the functions from Q_1 to S.

[5] An orbit is a finite/infinite sequence of configurations of some composition machine.

to the number of applications of G. Particularly, the configuration c exists at time $t = 0$, the configuration $G(c)$ is present at time $t = 1$, the configuration $(G \circ G)(c)$ appears at time $t = 2$, and so on. Indeed, the time evolution of a composition machine is similar to that of a synchronous cellular automaton [1,20,36]. The difference lies in their underlying semantics.

In a composition machine M, each organism $\alpha \in Q_1$ is causally related to some computon $f \in F$ via μ_1. Furthermore, the global state of M at t is semantically equivalent to a category which defines all the possible sequential compositions at that moment in time. So, the machine's purpose is not the computation of a single program but the emergence of a whole program space. More formally, a program space in M at t is generated from the path category $P(\overrightarrow{Q}^t)$, where $\overrightarrow{Q}^t$ is referred to as the *alive quiver* at t.

Definition 19 (Alive Quiver). *Let $M = (D, F, Q, \mu, S, N, \delta)$ be a composition machine and c be a machine configuration at t. The* alive quiver *$\overrightarrow{Q}^t \subseteq Q$ is a quadruple $(\overrightarrow{Q}_0, \overrightarrow{Q}_1, \overrightarrow{s}, \overrightarrow{\tau})$ where $\overrightarrow{Q}_0 = \{x \in Q_0 \mid \exists \alpha \in Q_1, c(\alpha)^t = 1 \ \wedge \ s(\alpha) = x \ \vee \ \tau(\alpha) = x\}$, $\overrightarrow{Q}_1 = \{\alpha \in Q_1 \mid c(\alpha)^t = 1\}$, $\overrightarrow{s} = \{(\alpha, s(\alpha)) \in Q_1 \times Q_0 \mid c(\alpha)^t = 1\}$, and $\overrightarrow{\tau} = \{(\alpha, \tau(\alpha)) \in Q_1 \times Q_0 \mid c(\alpha)^t = 1\}$.*

The path category on an alive quiver $\overrightarrow{Q}^t$ generates the space $(R \circ T \circ P)(\overrightarrow{Q}^t)$ of all possible sequential computons at time t. As per Definition 17, this space is produced through a computon category $(T \circ P)(\overrightarrow{Q}^t)$ which we refer to as the *alive category* at t.

Definition 20 (Alive Category). *Given a composition machine $M = (D, F, Q, \mu, S, N, \delta)$ and some alive quiver $\overrightarrow{Q}^t = (\overrightarrow{Q}_0, \overrightarrow{Q}_1, \overrightarrow{s}, \overrightarrow{\tau})$, the* alive category *$(T \circ P)(\overrightarrow{Q}^t)$ generated from $P(\overrightarrow{Q}^t)$ consists of:*

- *a collection $(T \circ P)(\overrightarrow{Q}^t)_0$ of data types such that $\forall x \in P(\overrightarrow{Q}^t)_0, \exists! d \in (T \circ P)(\overrightarrow{Q}^t)_0, d = \mu_0(x)$, and*
- *a collection $(T \circ P)(\overrightarrow{Q}^t)_1$ of computons where:*
  - *$\forall x \in P(\overrightarrow{Q}^t)_0, \exists! 1_d \in (T \circ P)(\overrightarrow{Q}^t)_1, d = \mu_0(x)$, and*
  - *$\forall (\alpha_n, \ldots, \alpha_1) \in P(\overrightarrow{Q}^t)_1, \exists! (f_n \circ \cdots \circ f_1) \in (T \circ P)(\overrightarrow{Q}^t)_1$, $f_1 = \mu_1(\alpha_1) \ \wedge \ \cdots \ \wedge \ f_n = \mu_1(\alpha_n)$ with $n \geq 1$.*

Theorem 3. *Given a composition machine $M = (D, F, Q, \mu, S, N, \delta)$, $(R \circ T \circ P)(Q)$ is the maximal program space defining all the possible sequential programs that can exist in M at any moment in time.*

Proof. Let $M = (D, F, Q, \mu, S, N, \delta)$ be a composition machine and $\overrightarrow{Q}^t$ be some alive quiver at time t. By Definition 19, we know that $\overrightarrow{Q}^t \subseteq Q$. Then, it trivially follows that $\overrightarrow{Q}^t = Q$ is the largest subset and, therefore, the maximal alive quiver that can be formed at any moment in time. By Definition 17, we construct the maximal program space $(R \circ T \circ P)(Q)$ as required.

A sequence of different program spaces can emerge periodically as a consequence of an intermittent configuration pattern appearing in the system of organisms. If a unique configuration pattern is repeated *ad infinitum* after time t in a machine M, then we say that M halts at t (see Definitions 21 and 22).

Definition 21 (Configuration Pattern). *A configuration pattern of period $k \geq 1$ is a finite orbit $orb(c_1) = c_1, c_2, \ldots, c_k$ where c_i and c_{i+1} are different configurations for all $i \in \mathbb{N} \cap [1, k-1]$.*

Definition 22 (Composition Machine Halting). *A composition machine M halts at time t if and only if $c_1, c_2, \ldots, c_k$ is a unique pattern appearing every k time steps after t. Equivalently, M halts when a unique finite sequence of distinct program spaces emerges every k time steps after t.*

6 Example

We now present an example to demonstrate how (complex) sequential program spaces emerge from simple rules. For this, we consider the composition machine $M = (D, F, Q, \mu, S, N, \delta)$ where:

- $D = \{d_i \mid i \in \mathbb{N} \cap [1, 13]\}$,
- $F = \{f_i \colon d_i \to d_{i+1} \mid i \in \mathbb{N} \cap [1, 6] \wedge d_i, d_{i+1} \in D\} \cup \{f_7 \colon d_8 \to d_9 \mid d_8, d_9 \in D\}$ $\cup \{f_i \colon d_{i+2} \to d_{i+3} \mid i \in \mathbb{N} \cap [8, 10] \;\wedge\; d_{i+2}, d_{i+3} \in D\}$,
- $Q = (Q_0, Q_1, s, \tau)$ where:
  - $Q_0 = \{x_i \mid i \in \mathbb{N} \cap [1, 13]\}$,
  - $Q_1 = \{\alpha_i \mid i \in \mathbb{N} \cap [1, 10]\}$,
  - $s = \{(\alpha_i, x_i) \mid i \in \mathbb{N} \cap [1, 6]\} \cup \{(\alpha_7, x_8)\} \cup \{(\alpha_i, x_{i+2}) \mid i \in \mathbb{N} \cap [8, 10]\}$,
  - $\tau = \{(\alpha_i, x_{i+1}) \mid i \in \mathbb{N} \cap [1, 6]\} \cup \{(\alpha_7, x_9)\} \cup \{(\alpha_i, x_{i+3}) \mid i \in \mathbb{N} \cap [8, 10]\}$,
- $\mu_0 = \{(x_i, d_i) \in Q_0 \times D \mid i \in \mathbb{N} \cap [1, 13]\}$,
- $\mu_1 = \{(\alpha_i, f_i) \in Q_1 \times F \mid i \in \mathbb{N} \cap [1, 10]\}$,
- $S = \{0, 1\}$,
- $N = \{N_1, N_2, N_3, N_4\}$ with $N_1 = \{(\alpha_7)\}$, $N_2 = \{(\alpha_1, \alpha_2), (\alpha_8, \alpha_9)\}$, $N_3 = \{(\alpha_5, \alpha_6), (\alpha_9, \alpha_{10})\}$ and $N_4 = \{(\alpha_1, \alpha_2, \alpha_3), (\alpha_2, \alpha_3, \alpha_4), (\alpha_3, \alpha_4, \alpha_5), (\alpha_4, \alpha_5, \alpha_6), (\alpha_8, \alpha_9, \alpha_{10})\}$,
- δ_1 is the NOT function, δ_2 and δ_3 are XOR operations, and δ_4 corresponds to *Rule 54* from elementary cellular automata. In its Boolean form, Rule 54 is equivalent to a function defined by $(p, q, r) \mapsto q \oplus (p \vee r)$ where $(p, q, r) \in \{0, 1\} \times \{0, 1\} \times \{0, 1\}$ and $q \oplus (p \vee r) \in \{0, 1\}$ (cf. [36]).

Fig. 6 illustrates the quiver Q and the maximal program space in M, viz. $(R \circ T \circ P)(Q)$. According to Theorem 3, this space is the most complex that our example machine can construct when all the organisms are alive. For simplicity, we do not show the identity computons looping over each data type, and we do not draw labels on composites to improve readability.

To arise familiarity, all the local transition functions of M are based on existing ones. Particularly, isolated organisms evolve through a NOT operation. Organisms with a binary neighbourhood change their states via the XOR operator. And organisms with a ternary neighbourhood evolve according to *Rule 54*. Thus, the orbit of M is given by the synchronous application of the rules illustrated in Fig. 7, starting from the initial configuration $c = \{(\alpha_1, 1), (\alpha_2, 1), (\alpha_3, 0), (\alpha_4, 1), (\alpha_5, 1), (\alpha_6, 0), (\alpha_7, 1), (\alpha_8, 0), (\alpha_9, 0), (\alpha_{10}, 1)\}$. This initial configuration and its corresponding program space $(R \circ T \circ P)(\overrightarrow{Q}^0)$ are illustrated in $t = 0$ of Fig. 8.

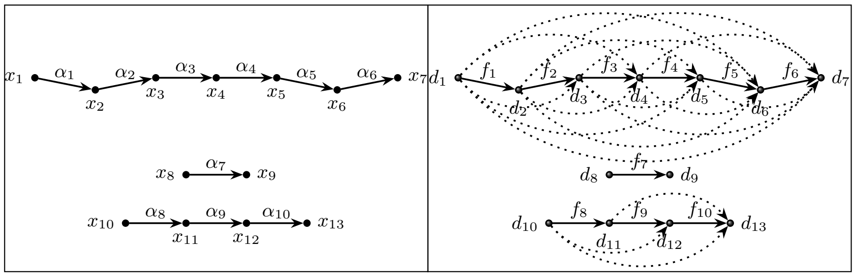


Fig. 6: The quiver Q and the corresponding maximal program space in M.

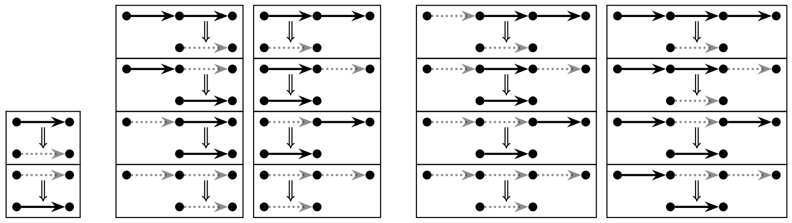
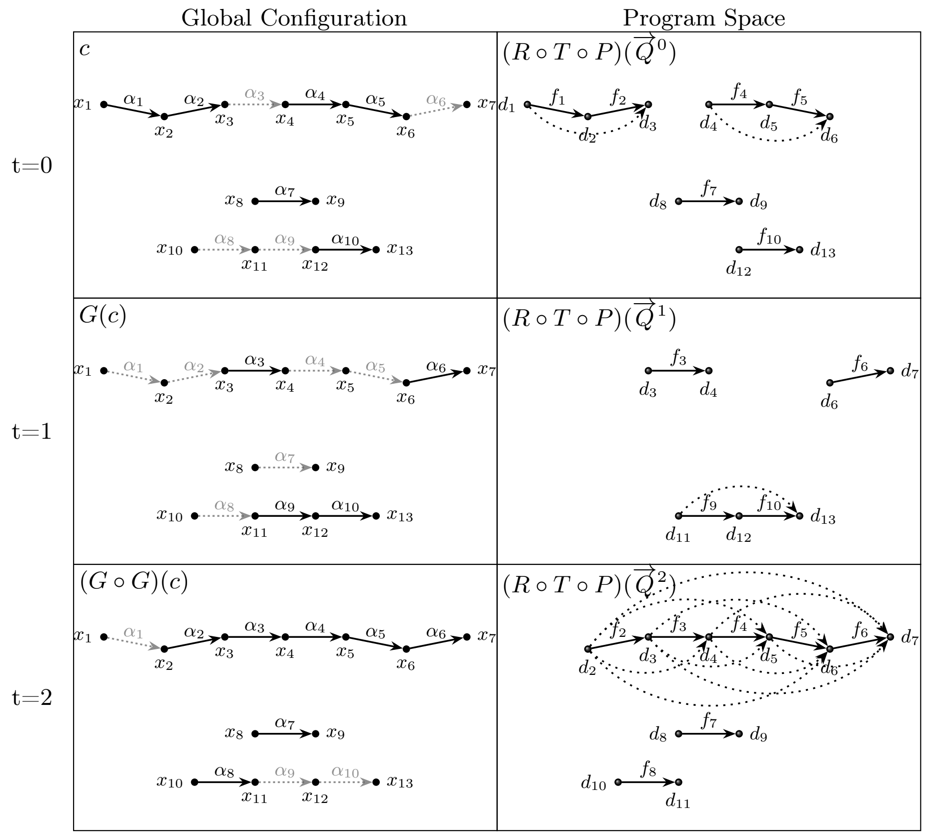


Fig. 7: Unary neighbourhoods are associated with the NOT operator, binary neighbourhoods with the XOR operator and ternary ones with *Rule 54*.

More precisely, Fig. 8 shows the orbit of M from $t = 0$ to $t = 2$, where it is clear that the isolated organism α_7 alternates its state via the NOT rule. Accordingly, the computon f_7 emerges at even time steps and disappears at odd ones. This figure also shows that there are are two, one and ten composite computons in $(R \circ T \circ P)(\overrightarrow{Q}^0)_1$, $(R \circ T \circ P)(\overrightarrow{Q}^1)_1$ and $(R \circ T \circ P)(\overrightarrow{Q}^2)_1$, respectively. For example, $f_{10} \circ f_9$ is present in $(R \circ T \circ P)(\overrightarrow{Q}^1)_1$ because $c(\alpha_9)^1 = 1$, $c(\alpha_{10})^1 = 1$, $\mu_1(\alpha_9) = f_9$ and $\mu_1(\alpha_{10}) = f_{10}$. Supposing f_9 is a program of type $\mathbb{Z} \to \mathbb{R}$ that multiplies an integer by 0.5 and f_{10} is a program of type $\mathbb{R} \to$ *String* that converts a real number into its equivalent string representation, then $f_{10} \circ f_9 \colon \mathbb{Z} \to \mathit{String}$ defines a (composite) sequential program that takes an integer z and returns the string representation of $z \times 0.5$. This composite program is not created manually by some coder, but it is automatically generated by M at time $t = 1$. As the notion of a composition machine is implementation-agnostic, in this paper we do not deal with concrete data types or concrete computons. The example we consider is rather abstract with the aim of preserving generality.

In our example, organisms with a binary neighbourhood evolve via the XOR rule. So, they are alive in the next time step if only one neighbour is alive in the current time step. For example, α_6 becomes alive at $t = 1$ because its left neighbour α_5 is alive at $t = 0$. But α_1 is dead at $t = 1$ since $c(\alpha_1)^0 = c(\alpha_2)^0 = 1$. Consequently, we have $f_6 \in (R \circ T \circ P)(\overrightarrow{Q}^1)_1$ and $f_1 \notin (R \circ T \circ P)(\overrightarrow{Q}^1)_1$. More concretely, the application of the XOR rule on binary neighbourhoods from $t = 0$ to $t = 1$ results in the following state transitions:

$$
\begin{aligned}
c(\alpha_1)^1 &= \delta_2(c(\alpha_1)^0, c(\alpha_2)^0) = 1 \oplus 1 = 0 \\
c(\alpha_8)^1 &= \delta_2(c(\alpha_8)^0, c(\alpha_9)^0) = 0 \oplus 0 = 0 \\
c(\alpha_6)^1 &= \delta_3(c(\alpha_5)^0, c(\alpha_6)^0) = 1 \oplus 0 = 1 \\
c(\alpha_{10})^1 &= \delta_3(c(\alpha_9)^0, c(\alpha_{10})^0) = 0 \oplus 1 = 1
\end{aligned}
$$



Global Configuration
Program Space
t=0
t=1
t=2
c
$G(c)$
$(G \circ G)(c)$
$(R \circ T \circ P)(\overrightarrow{Q}^0)$
$(R \circ T \circ P)(\overrightarrow{Q}^1)$
$(R \circ T \circ P)(\overrightarrow{Q}^2)$
x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12} x_{13}
α_1 α_2 α_3 α_4 α_5 α_6 α_7 α_8 α_9 α_{10}
d_1 d_2 d_3 d_4 d_5 d_6 d_7 d_8 d_9 d_{10} d_{11} d_{12} d_{13}
f_1 f_2 f_3 f_4 f_5 f_6 f_7 f_8 f_9 f_{10}

Fig. 8: Orbit of M over three time steps via the rules described in Fig. 7.

Organisms with a ternary neighbourhood evolve according to *Rule 54*. For instance, the organism α_9 becomes alive at $t = 1$ since $c(\alpha_8)^0 = c(\alpha_9)^0 = 0$ and $c(\alpha_{10})^0 = 1$; thus, making available the computon $f_9 \in (R \circ T \circ P)(\overrightarrow{Q}^1)_1$. As the organism α_{10} is also alive at that moment in time, the composite computon $f_{10} \circ f_9 \in (R \circ T \circ P)(\overrightarrow{Q}^1)_1$ emerges. In general, the application of *Rule 54* on ternary neighbourhoods from $t = 0$ to $t = 1$ results in the following transitions:

$$
\begin{aligned}
c(\alpha_2)^1 &= \delta_4(c(\alpha_1)^0, c(\alpha_2)^0, c(\alpha_3)^0) = \delta_4(1, 1, 0) = 0 \\
c(\alpha_3)^1 &= \delta_4(c(\alpha_2)^0, c(\alpha_3)^0, c(\alpha_4)^0) = \delta_4(1, 0, 1) = 1 \\
c(\alpha_4)^1 &= \delta_4(c(\alpha_3)^0, c(\alpha_4)^0, c(\alpha_5)^0) = \delta_4(0, 1, 1) = 0
\end{aligned}
$$

$$c(\alpha_5)^1 = \delta_4(c(\alpha_4)^0, c(\alpha_5)^0, c(\alpha_6)^0) = \delta_4(1,1,0) = 0$$
$$c(\alpha_9)^1 = \delta_4(c(\alpha_8)^0, c(\alpha_9)^0, c(\alpha_{10})^0) = \delta_4(0,0,1) = 1$$

To further investigate the emergence of program spaces, we implemented our composition machine M.[6] After executing this example over 10000 time steps, we found that the most complex program space is formed at $t = 2$, i.e., when all the organisms in the path $(\alpha_6, \ldots, \alpha_2)$ become alive. Accordingly, $f_6 \circ f_5 \circ f_4 \circ f_3 \circ f_2 \in (R \circ T \circ P)(\overrightarrow{Q}^2)_1$ is the most complex composite computon (i.e., the largest sequential program) in our example. With our tool, we also found that there is a repeating pattern every four time steps, starting from $t = 5$, meaning that the compositions formed before $t = 5$ are unique and, after that, there are only four different program spaces. Such a repeating pattern indicates that M halts at $t = 4$. Although we do not show it due to space constraints, the pattern is depicted in [3]. In [3], we additionally present a more complex example.

7 Conclusions

In this paper, we introduced the notion of composition machines for the emergence of sequential program spaces via self-organisation rules. A composition machine evolves a quiver in discrete-time and its global state is semantically equivalent to a category that defines a space of all the possible compositions of computons at some instant. To demonstrate its operation, we presented an example in which complex sequential program spaces emerge from simple rules.

As it is not trivial to engineer emergence, we plan to investigate novel mechanisms for defining rules that match contextual intention or high-level specifications. Also, we plan to study alternative neighbourhoods to allow the evolution of cyclic quivers and, thus, the emergence of infinite categories. In this vein, we would like to explore different program space patterns through the synchronous application of varied local transition rules. In any case, Category Theory will be the *de facto* reasoning framework. We believe that Category Theory will play a fundamental role in the understanding of self-organisation and, in particular, in the study of self-organising software models. This is because such a theory follows an emergentist view rather than a reductionist one. So, the main focus of Category Theory is not to study isolated abstract objects but the relevant interactions between them and their composition. It is indeed a formal framework to reasoning about compositionality and self-organisation.

This paper brings together these worlds in the form of an implementation-independent abstract machine that allows the definition and the execution of self-organising software models. The aim of this machine is not to compute single programs, but to build programs (i.e., composite computons) from other programs (i.e., predefined computons) via self-organisation rules. Under this umbrella, we envision that in the future, instead of manually coding them, complex software models could grow from simple rules (just as biological organisms do).

[6] Our tool is available at https://github.com/damianarellanes/compositionmachine.

References

1. Acerbi, L., Dennunzio, A., Formenti, E.: Conservation of some dynamical properties for operations on cellular automata. Theoretical Computer Science **410**(38), 3685–3693 (2009)
2. Alur, R., Bodik, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Formal Methods in Computer-Aided Design. pp. 1–8 (2013)
3. Arellanes, D.: Composition Machines: Programming Self-Organising Software Models for the Emergence of Sequential Program Spaces. Tech. Rep. arXiv:2108.05402v1, arXiv (2021)
4. Arellanes, D.: Self-Organizing Software Models for the Internet of Things: Complex Software Structures That Emerge Without a Central Controller. IEEE Systems, Man, and Cybernetics Magazine **7**(3), 4–9 (2021). https://doi.org/10.1109/MSMC.2021.3062822
5. Arellanes, D., Lau, K.K.: Workflow Variability for Autonomic IoT Systems. In: International Conference on Autonomic Computing (ICAC). pp. 24–30. IEEE (2019)
6. Arellanes, D., Lau, K.K.: Evaluating IoT service composition mechanisms for the scalability of IoT systems. Future Generation Computer Systems **108**, 827–848 (2020)
7. Arellanes, D., Lau, K.K., Sakellariou, R.: Decentralized Data Flows for the Functional Scalability of Service-Oriented IoT Systems. The Computer Journal **66**(6), 1477–1506 (2023)
8. Arrighi, P., Dowek, G.: Causal graph dynamics. Information and Computation **223**, 78–93 (2013)
9. Awodey, S.: Category Theory. Oxford University Press, New York, NY, USA, 2nd edn. (2010)
10. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.B.: The FRACTAL component model and its support in Java. Software: Practice and Experience **36**(11-12), 1257–1284 (2006)
11. Derksen, H., Weyman, J.: An Introduction to Quiver Representations, Graduate studies in mathematics, vol. 184. American Mathematical Society, Providence, Rhode Island, USA (2017)
12. Doty, D.: Theory of algorithmic self-assembly. Communications of the ACM **55**(12), 78–88 (2012)
13. Filho, R.R., Porter, B.: Defining Emergent Software Using Continuous Self-Assembly, Perception, and Learning. ACM Transactions on Autonomous and Adaptive Systems **12**(3), 16:1–16:25 (2017)
14. Floch, J., Hallsteinsen, S., Stav, E., Eliassen, F., Lund, K., Gjorven, E.: Using architecture models for runtime adaptability. IEEE Software **23**(2), 62–70 (2006)
15. Gulwani, S., Polozov, O., Singh, R.: Program synthesis, Foundations and trends in programming languages, vol. 4. Now Publishers, Hanover, MA Delft (2017)
16. Gurevich, Y.: On kolmogorov machines and related issues. In: Current Trends in Theoretical Computer Science, World Scientific Series in Computer Science, vol. 40, pp. 225–234. World Scientific (1993)
17. Hirsch, D., Kramer, J., Magee, J., Uchitel, S.: Modes for Software Architectures. In: Gruhn, V., Oquendo, F. (eds.) Software Architecture. pp. 113–126. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2006)
18. Iovino, M., Styrud, J., Falco, P., Smith, C.: Learning Behavior Trees with Genetic Programming in Unpredictable Environments. In: IEEE International Conference on Robotics and Automation (ICRA). pp. 4591–4597 (2021)

19. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: ACM/IEEE International Conference on Software Engineering (ICSE). pp. 215–224. ACM (2010)
20. Kari, J.: Theory of cellular automata: A survey. Theoretical Computer Science **334**(1-3), 3–33 (2005)
21. Lehmann, D.J., Smyth, M.B.: Algebraic specification of data types: A synthetic approach. Mathematical systems theory **14**(1), 97–139 (1981)
22. Mohr, F.: Automated Software and Service Composition: A Survey and Evaluating Review. SpringerBriefs in Computer Science, Springer, Cham (2016)
23. Neumann, J.V.: Theory of Self-reproducing Automata. University of Illinois Press, Urbana, Illinois (1966)
24. Pierce, B.C.: Basic Category Theory for Computer Scientists. Foundations of Computing, MIT Press, Cambridge, MA, USA (1991)
25. Plump, D.: Term graph rewriting. In: Handbook of Graph Grammars and Computing by Graph Transformation, pp. 3–61. World Scientific (1999)
26. Rao, J., Su, X.: A Survey of Automated Web Service Composition Methods. In: Cardoso, J., Sheth, A. (eds.) Semantic Web Services and Web Process Composition. pp. 43–54. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2005)
27. Russell, B.: The Principles of Mathematics, vol. 1. Cambridge: Cambridge University Press (1903)
28. Sayama, H.: Generative Network Automata: A Generalized Framework for Modeling Complex Dynamical Systems with Autonomously Varying Topologies. In: IEEE Symposium on Artificial Life. pp. 214–221 (2007)
29. Schönhage, A.: Storage Modification Machines. SIAM Journal on Computing **9**(3), 490–508 (1980)
30. da Silva, A.S., Ma, H., Mei, Y., Zhang, M.: A Survey of Evolutionary Computation for Web Service Composition: A Technical Perspective. IEEE Transactions on Emerging Topics in Computational Intelligence **4**(4), 538–554 (2020)
31. Simon, H.A.: Whether software engineering needs to be artificially intelligent. IEEE Transactions on Software Engineering **SE-12**(7), 726–732 (1986)
32. Smith, D.M.D., Onnela, J.P., Lee, C.F., Fricker, M.D., Johnson, N.F.: Network automata: coupling structure and function in dynamic networks. Advances in Complex Systems **14**(03), 317–339 (2011)
33. Spivak, D.I.: Functorial data migration. Information and Computation **217**, 31–51 (2012)
34. Töpfer, M., Abdullah, M., Bureš, T., Hnětynka, P., Kruliš, M.: Machine-learning abstractions for component-based self-optimizing systems. International Journal on Software Tools for Technology Transfer **25**(5), 717–731 (2023)
35. Waldegrave, R., Stepney, S., Trefzer, M.A.: Developmental Graph Cellular Automata. MIT Press (2023)
36. Wolfram, S.: A New Kind of Science. Wolfram Media, Champaign, IL, USA, 1st edition edn. (2002)
37. Woods, D.: Intrinsic universality and the computational power of self-assembly. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences **373**(2046), 1–13 (2015)
38. Yoon, Y., Lee, W., Yi, K.: Inductive Program Synthesis via Iterative Forward-Backward Abstract Interpretation. Proceedings of the ACM on Programming Languages **7**(174), 1657–1681 (2023)