Hindawi
Complexity
Volume 2019, Article ID 6573793, 11 pages
https://doi.org/10.1155/2019/6573793

Research Article

A Fully Operational Framework for Handling Cellular Automata Templates

Mauricio Verardo[1] **and Pedro P. B. de Oliveira**[1,2]

[1]*Programa de Pós-Graduação em Engenharia Elétrica e Computação, Universidade Presbiteriana Mackenzie, Rua da Consolação 896, Consolação, 01302-907 São Paulo, SP, Brazil*
[2]*Faculdade de Computação e Informática, Universidade Presbiteriana Mackenzie, Rua da Consolação 896, Consolação, 01302-907 São Paulo, SP, Brazil*

Correspondence should be addressed to Pedro P. B. de Oliveira; pedrob@mackenzie.br

Received 6 December 2018; Accepted 24 March 2019; Published 3 April 2019

Academic Editor: Carlos Gershenson

Cellular automata are fully discrete, computational, or dynamical systems, characterised by a local, totally decentralised action. Although extremely simple in structure, they are able to represent arbitrarily complex phenomena. However, due to the very big number of rules in any nontrivial space, finding a local rule that globally unfolds as desired remains a challenging task. In order to help along this direction, here we present the current state of cellular automata *templates*, a data structure that allows for the representation of sets of cellular automata in a compact manner. The template data structure is defined, along with processes by which interesting templates can be built. In the end, we give an illustrative example showcasing how templates can be used to explore a very large cellular automaton space. Although the idea itself of template has been introduced before, only now its conceptual underpinnings and computational robustness rendered the notion effective for practical use.

1. Introduction

Cellular automata (CAs) can be regarded as both fully discrete computational and dynamical systems, characterised by a local, totally decentralised action. What makes them an attractive object of study is the fact that, in spite of their extremely simple structure, they may present arbitrarily complex behaviour, depending on the local rule chosen to govern the system [1]. The rule acts by changing the discrete state of every position (a cell) of a regular grid, according to its current state and those of its neighbouring cells.

The set of all possible one-dimensional CAs can be partitioned using two important parameters: the number of states the CA allows a cell to take on at any given time step and the size of its neighbourhood, defined by the number of neighbours a cell has to check in order to decide which state it will be at the next time step. Each resulting block of the partition defines a CA space or family.

Except for the small elementary space (made up of only 256 one-dimensional rules, with 2 states per cells and 3 cells in the neighbourhood), whenever there is the necessity of searching for CAs with a given behaviour in a given space, one is overwhelmed by the very high number of possibilities to check. This is why many applications end up revolving around the elementary rules, as in [2].

The approaches used to perform searches in larger spaces have usually been made automatic by means of search algorithms, especially evolutionary, designed to look for rules that would have particular emergent features, such as those that can be associated with the solution of particular computational problems [3], or the direct attempt to analyse constrained CA spaces, defined by required properties of the rule tables or the expected behaviour to be observed [4, 5]. In order to escape from the arbitrariness of the latter, different representational tools sometimes are employed, most notably the DeBruijn graphs, used in [6], to explore the parity problem's solvability for CAs in different spaces, and the conditionally matching rules, explored in [7].

Regardless of the case, the fact is that the possibility of representing families of CA rules in a compact way would

be greatly beneficial, both theoretically and for practical applications. A particularly powerful representational tool in this context is the *template*. First introduced in [8], templates are data structures which represent potentially extremely large sets of CAs, further allowing to combine and manipulate them with operations such as the intersection, before effectively enumerating the whole set.

Even though the idea itself of template has been introduced before, only now its conceptual underpinnings and computational robustness rendered the notion effective for practical use. In tune with that, the idea has now been crystallised into an open-source package for the Wolfram *Mathematica* system [9] called *CATemplates* [10]. The explanations provided in this paper are partially based on the implementations used in *CATemplates*.

Although cellular automata can be defined in any number of dimensions, here we focus on the one-dimensional case, first because the major benchmarking problems from the literature are one-dimensional and second because this is the case implemented in *CATemplates*.

The rest of this paper is organised as follows. Section 2 is a brief overview of cellular automata and their inner workings, Section 3 defines a cellular automata template, and Section 4 provides explanations about the operations applicable to them. Section 5 then presents examples of template builder functions, Section 6 discusses a sample application of templates to find rules that share both totality and captivity in extremely large CA spaces, and Section 7 closes the paper with some concluding remarks and ideas for future work.

2. Cellular Automata

Cellular automata are dynamical, decentralised systems, governed by simple, local rules. Given the relative simplicity of CAs, they serve as a good model to analyse how the interaction among simple components can give rise to complex behaviour in dynamical systems, in particular, how solutions to global problems like the density classification task arise [11].

CAs are composed of a regular lattice of finite automata (the *cells*), whose states change with time, following a predetermined local rule. The lattice can be arranged in any number of dimensions and can have a finite or infinite number of cells. From now on, whenever a CA is mentioned, we always mean one-dimensional CAs (whose lattices are arranged as a line), each cell can take on a discrete state value, and the update is parallel, at discrete time steps.

A CA's cell state is usually represented by numbers in the $[0, k-1]$ interval, or a colour from a set of k colours previously defined. The local rule of a CA acts on each cell's neighbourhood, which is a set composed of the cell itself together with its $2r$ adjacent cells, with r being the *radius* of the rule, that is, the number of cells at each side. The fractional value .5 for r indicates that the neighbourhood is asymmetric, as in $r = 1.5$, meaning that the center cell has 2 neighbours on one side and 1 on the other (in *CATemplates*, by convention, the left-hand side contains the larger number of neighbours).

Table 1: Rule table for the elementary CA 184.

(1,1,1)	1
(1,1,0)	0
(1,0,1)	1
(1,0,0)	1
(0,1,1)	1
(0,1,0)	0
(0,0,1)	0
(0,0,0)	0

By defining a value for k and r, a space (or family) of CAs is established. The *elementary space* is currently one of the most extensively studied CA families, as it is composed of only 256 rules, even though displaying a rich phenomenology [1].

The size of a CA space is a function of k and r. In the one-dimensional case, this means $s(k, r) = k^{k^{\lfloor 2r+1 \rfloor}}$. As a consequence, the increase of k or r quickly leads to exponentially large space sizes; for instance, the space of binary CAs with radius 2 is composed of 2^{32} rules.

Every CA's temporal evolution is governed by a local rule. This local rule is usually represented as a rule table, consisting of state transitions that point every possible neighbourhood to an output state. At every time step, every cell in the lattice is updated according to its neighbourhood, following its state transition table.

Since a rule table has to cover all possible neighborhoods in the lattice, it consists of $k^{\lfloor 2r+1 \rfloor}$ state transitions ordered using Wolfram's lexicographic order, in which the first transition is the one corresponding to the neighbourhood formed only by cells in the $k-1$ state, and the last one is the transition corresponding to the neighbourhood of zeros. To illustrate the point, the rule table for elementary CA 184 is shown in Table 1.

It is possible to compress the rule table by using its k-ary representation. By discarding Table 1's first column, we are left with the outputs of each transition: $(1, 0, 1, 1, 1, 0, 0, 0)$. Since we are still following Wolram's convention, we can switch freely between both representations. If interpreted as a binary number, this representation can be converted to its decimal form, 184, which is referred to as the rule (Wolfram) number.

The nature of the $s(k, r)$ function represents a great challenge when searching for specific sets of CAs. One of the employed strategies for this kind of search is to restrict these very large spaces into smaller ones, using *static* properties of CAs. The template framework described in this paper aims to automate this process. Static properties are those we can draw directly from the rule table, instead of looking at the temporal evolution. This means that they determine specific ways a rule table should be built, so that the resulting rule would possess them. As will be detailed in Section 5, template builder functions use this idea in order to create templates representing specific subsets of CA spaces.

3. Cellular Automata Templates

A cellular automaton template (referred to simply as *template* from here on) is a data structure that allows for the representation of a (potentially large) subset of a CA's family in a compact manner.

Template builder functions are defined as a means to create templates of interesting subsets of CAs, usually following a well defined static property derived from CA's rule tables. They are explained in detail in Section 5.

Two main operations are defined over templates: Expansion and Intersection. Expansion is used to convert a template into the rule set it represents, and Intersection is applied to two templates to generate a third, representing the rule set of the resulting intersection. Both operations are detailed in Section 4.

The template concept was first described in [8, 12] and has gone through significant improvements since then. For instance, the template's *core*, essentially what was described as the template itself in the original papers, is now a part of a tuple composed by important attributes, as shown in the definition below.

Also, new template builder algorithms have been developed, the concept of expansion modifier functions has been implemented, and the intersection operation, at that time implicitly couched in terms of a superseded "composition" operation—which was just a part of the original template builder algorithms described in the original papers—has been made explicit and evolved into a generalised operation.

Furthermore, the original work was totally restricted to binary rules, which no longer is the case; the code was completely *ad hoc* (and therefore, private to the authors), with no internal structure that would allow it to grow toward a usable and useful system, beyond its original conception; and only two rule properties have been accounted for (additional information about these two properties is given later in the paper).

So, in its origin, there was a still immature but promising idea that was conveyed at the time to two distinct audiences: the early technical details of the implementation to *Mathematica* users [8] and the actual idea and its prospects as a short communication to the cellular automata community [12]. From then on, the work developed, until reaching the required degree of maturation that it deserved, as embodied in the present report. All the further developments are now consolidated as a fully functional open-source implementation in the form of the *CATemplates* package [10], within Wolfram *Mathematica* software [9].

Armed with this framework, one can efficiently build and manipulate templates as needed and avoid the expensive enumeration of the sets they stand for until a template is obtained, representing a small enough rule set.

Definition 1 (template). A template is the tuple (k, r, c, e), where k and r are the number of states and the neighbourhood radius, respectively, of the cellular automata rules of the space that is being represented, c is the template core, and e is an expansion modifier function.

The *core* is by far the most important part of a template. It is essentially a rule table in k-ary form with a twist: every output of this rule table is allowed to be a function of the other outputs. This is achieved by introducing variables semantically bounded across all transitions of the classical rule-table representation. By means of the core, a template is able to represent a set of CAs. The following template (T_1) is an example:

$$T_1 = \left(2, 1.0, \left(0, 1 - x_1, 0, 1, x_3, 1, x_1, 0\right), IdentityMapper\right). \quad (1)$$

The $IdentityMapper$ and other expansion modifier functions will be explained in more detail in Section 4.

Remark. In the *CATemplates* package, k, r, and e may be omitted, in which case, the default values of $k = 2$, $r = 1.0$, and $e = IdentityMapper$ are assumed; similarly, but for the sake of simplicity, from now on in this paper whenever we refer to a template's core in isolation, say, $c(T_1)$, the reader should assume the same previous default values for the corresponding template (T_1).

We indicate the state transitions of the template's core by an index, starting at 0, on the right-most position, up to $k^{\lfloor 2r+1 \rfloor} - 1$, on the left-most, so as to mirror their correspondence to the neighborhoods they represent. As such, whenever T_1's transition 0 is mentioned, we are referring to the first 0 that appears in T_1's core, from right to left. Transition 1 refers to x_1, and so on.

T_1's transition 0 is a sample of a *fixed* output. This means any CA with a value different from 0 on its rule table's first transition—which is the same as the state transition (0,0,0) leading to 1—is not an element of the set represented by T_1.

T_1's transition 1 is a sample of a *free variable* (x_1). A free variable is assumed to represent any value in the range $[0, k-1]$. As a convention, free variables are indexed following the transition where they first appear. Were we to consider only T_1's first two transitions, we could say it represents the set $\{(0, 0), (1, 0)\}$, since the first transition is fixed and the second is a free variable.

Transition 6 is a sample of a *function*. As T_1 is a binary template (since $k = 2$), the expression $1 - x_1$ means this transition will always be evaluated to the opposite of the value of transition 1. It is important to note that we could use any algebraic expression here, potentially referring to any other variable in the template, and we would still be able to derive the set this template represents. This is actually done in some template builder functions, as will be seen in Section 5.

It follows from the previous observations that T_1 actually corresponds to the binary rule set $\{(0, 1, 0, 1, 0, 1, 0, 0), (0, 1, 0, 1, 1, 1, 0, 0), (0, 0, 0, 1, 0, 1, 1, 0), (0, 0, 0, 1, 1, 1, 1, 0)\}$, which could also be written as $\{22, 30, 84, 92\}$, following Wolfram's numbering convention.

A template whose core is made of only free variables is called a *Base Template* and essentially represents the whole CA space given by k and r; for instance, the *Base Template* for the elementary CAs is

$$\left(2, 1.0, \left(x_7, x_6, x_5, x_4, x_3, x_2, x_1, x_0\right), IdentityMapper\right). \quad (2)$$

Base templates are often used by template builder functions as a starting point.

4. Template Operations

Currently, templates' processing supports two operations: expansion (E) and intersection (I). This section presents details on how they work.

4.1. Expansion. Every template represents a set R of k-ary rule tables. This set is obtained by means of the expansion operation.

The i-th expansion of template T, denoted as $E_i(T)$, begins with the extraction of all variables present in the template's core, in a total of m variables. For instance, given the template's core $c(T_1) = (0, 1-x_1, 0, 1, x_3, 1, x_1, 0)$, the ordered list of extracted variables would be (x_3, x_1), ordered by the variable's index, in decreasing order. Next, i is converted to base k, padded with zeros to the left, up to achieving m digits. For $i = 2$, we would have the ordered list of digits $(1, 0)$, since 10 is 2 in binary. We then pair the extracted variables with the digits in order and replace every occurrence of the variable with the corresponding value, like $(0, 1-0, 0, 1, 1, 1, 0, 0)$. Any algebraic expression left in the core is then resolved, resulting in $E_2(T_1) = (0, 1, 0, 1, 1, 1, 0, 0)$.

With the process to find the i-th expansion defined, it is only a matter of mapping it over every number in the interval $[0, k^m - 1]$, and we have the set R of CA rule tables represented by the template. This is how the full expansion $E(T)$ is implemented.

Still considering T_1, since $m = 2$, we only have $k^2 = 4$ possible expansions for this template. By performing the expansion procedure with $i \in \{0, 1, 2, 3\}$, we end up with the set $R = \{(0, 1, 0, 1, 0, 1, 0, 0), (0, 1, 0, 1, 1, 1, 0, 0), (0, 0, 0, 1, 0, 1, 1, 0), (0, 0, 0, 1, 1, 1, 1, 0)\}$.

Upon expansion, templates can generate invalid results; for instance, let us consider the template's core $c(T_{invalid}) = (0, 0, 0, 0, 2 - x_3, 1, 1, 1)$.

Since $m = 1$ and $k = 2$, this template has 2 possible expansions, namely, $E(T_{invalid}) = \{(0, 0, 0, 0, 2, 1, 1, 1), (0, 0, 0, 0, 1, 1, 1, 1)\}$; however, the set contains an invalid state transition, as it leads to the 2-state, which is not in the $[0, k-1]$ range.

Expansion modifier functions can be used as a mechanism by which the template provides auxiliary information for the expansion operation, in terms of some details it should abide by.

Every template has at least one expansion modifier function defined. Whenever the expansion operation is performed on the i-th expansion, its result is passed on to the template's expansion modifier function, which is defined by the template builder function.

This means that every template itself is able to define specific behaviours to the expansion operation, deciding how to deal with unwanted results.

Expansion modifier functions are currently divided into two categories: *Mappers* and *Filters*.

4.1.1. Mappers. A *Mapper* is a function that receives the i-th expansion result and performs some kind of transformation on it. So far, we have identified the need for two expansion modifier functions of the Mapper kind: *ModKMapper* and *IdentityMapper*.

(i) *IdentityMapper* simply receives the expansion and leaves it untouched.

(ii) *ModKMapper* performs the *modulus* operation on each transition of the expansion result using the template's k as divisor, thus obtaining a valid expansion.

As an example of the latter, let us define T_{modk} as a variation of $T_{invalid}$ that uses *ModKMapper* instead of the *IdentityMapper*:

$$T_{modk} = \left(2, 1.0, \left(0, 0, 0, 0, 2 - x_3, 1, 1, 1\right), ModKMapper\right). \quad (3)$$

Now, the result of expanding T_{modk} becomes

$$E\left(T_{modk}\right) = \{(0, 0, 0, 0, 0, 1, 1, 1), (0, 0, 0, 0, 1, 1, 1, 1)\}. \quad (4)$$

4.1.2. Filters. A *Filter* is a function that checks the expansion result and decides whether it is an acceptable result for the given template. If it finds an unacceptable expansion candidate, it filters out this result, by removing it from the set R. So far, we have identified the need for two expansion modifier functions of the Filter kind: *FilterKOutOfRange* and *FilterVariableAssignments*.

(i) *FilterKOutOfRange* simply removes any invalid expansion. For instance, by defining $T_{filterK}$ as a variation of $T_{invalid}$ that uses *FilterKOutOfRange* instead of the *IdentityMapper*, and this results in $E(T_{filterK}) = \{(0, 0, 0, 0, 1, 1, 1, 1)\}$.

(ii) *FilterVariableAssignments* serves a different kind of need. It allows for templates to define annotations on its core's variables, which dictate the values from the $[0, k-1]$ range the variable is allowed to have.

The annotation referred to in *FilterVariableAssignments* is expressed with a special syntax on the template's core. Consider the following template T_{va} and its $i = 0$ expansion:

$$T_{va} = \left(2, 1.0, \left(0, 0, 0, 0, x_3 \in \{1\}, 1, 1, 1\right), FilterVariableAssignments\right). \quad (5)$$

The process should result in the expansion candidate $(0, 0, 0, 0, 0 \in \{1\}, 1, 1, 1)$. But since $0 \in \{1\}$ is false, *FilterVariableAssignments* detects this as unacceptable and removes it from R.

In the binary case, *FilterVariableAssignments* is seldom needed, since the expression $x_{free} \in \{0, 1\}$ is equivalent to a free variable x_{free}, and the expression $x_{fixed} \in \{0\}$ is equivalent to the fixed value 0.

Therefore, this expansion modifier function becomes useful when $k > 2$, for various reasons. For further clarification, Section 5.2 discusses an application of this filter.

4.2. Intersection. Template intersection is a function of two templates, which results in a template whose associated set R is equivalent to intersecting the R sets of the two templates received as arguments. In another form,

$$I(T_1, T_2) = T_3 \Longleftarrow E(T_3) = E(T_1) \cap E(T_2). \quad (6)$$

Template intersection initially consists of building an equation system with the template's cores received as operands. The equation system is then submitted to *Mathematica*'s Solve function [9], which solves it for the variables that admit a solution, and elicits the relations among the remaining variables. The resulting set of variable values and relations is then applied to any of the two operands, and the outcome is a new template representing only the intersection between both of the input templates.

If the system has no solution, this means both templates represent disjoint R sets and, by definition, do not intersect.

As an example, consider the template cores $c(T_1) = (x_7, 0, x_5, 0, 1, 0, x_2, x_0)$ and $c(T_2) = (1, 0, 1 - x_1, 1 - x_3, x_3, 0, x_1, 0)$. The associated equation system thus becomes

$$\begin{aligned} x_7 &= 1 \\ 0 &= 0 \\ x_5 &= 1 - x_1 \\ 0 &= 1 - x_3 \\ 1 &= x_3 \\ 0 &= 0 \\ x_1 &= x_1 \\ x_0 &= 0. \end{aligned} \quad (7)$$

The solution set of the equation system, as given by *Mathematica*'s Solve, is $S = \{x_0 = 0, x_3 = 1, x_2 = x_1, x_5 = 1 - x_1, x_7 = 1\}$. We can then substitute the variables in $c(T_1)$ using this set, thus rendering the template's core $(1, 0, 1 - x_1, 0, 1, 0, x_1, 0)$. Notice that the same result would be achieved had we applied the substitutions to $c(T_2)$.

If a template's core has variable restrictions, a second step is needed to guarantee that the solution found will not violate them.

For instance, consider the templates' cores $c(T_{r1})$ and $c(T_{r1})$, both with restricted variables: $c(T_{r1}) = (x_7, 0, x_5, 0, 1, 0, x_2 \in \{0, 1\}, x_0)$ and $c(T_{r2}) = (1, 0, 1 - x_1, 1 - x_3, x_3, 0, x_1 \in \{0\}, 0)$. To intersect T_{r1} and T_{r2}, we first extract the variable restriction expressions and then build and solve the system as usual. This time, however, the substitutions are applied to both arguments, therefore resulting in the set $\{(1, 0, 1 - x_1, 0, 1, 0, x_1 \in \{0, 1\}, 0), (1, 0, 1 - x_1, 0, 1, 0, x_1 \in \{0\}, 0)\}$.

Now, we extract variable restriction expressions and create a second equation system, whose solutions indicate which values the variables may have. In the example, the set of expressions $\{x_1 \in \{0, 1\}, x_1 \in 0\}$ would be extracted, thus yielding the equation system:

$$\begin{aligned} x_1 = 1 \vee x_1 = 0 \\ x_1 = 0 \end{aligned} \quad (8)$$

By solving this system, the solution set $S = \{x_1 = 0\}$ can finally be applied to $c(T_{r1})$, yielding $(1, 0, 1, 0, 1, 0, 0, 0)$ as the result.

5. Template Builders

So far we have seen how templates can be built in order to represent specific subsets of CA families. We can now go about making evident the true power of templates, by considering template builder functions.

Template builder functions are algorithms tailored to generate templates for specific subsets of CAs that share a given static property.

Currently, six template builders for static properties are implemented, associated to rules of the following kinds: captive; totalistic; outer-totalistic; number conserving; colorblind; and maximum internal symmetry values (see [1]) for reflection, conjugacy, and their composition. The latter two are not discussed herein partly because we have already discussed them previously (see [8]), and, at least in the context number conservation, the corresponding template builder simply did not change since its conception. However, the builder for rule table symmetries has undergone a generalisation that now allows for any value of symmetry to be accounted for, not only maximum. As such, the *CATemplates* package contains the *SymmetryTemplate* builder, fully available for use, including its corresponding documentation, but we decided not to include it in the paper because it cannot yet handle arbitrary number of states in the case of conjugation.

So, this section discusses builder functions for totalistic, outer-totalistic, captive, and colorblind templates.

5.1. Template Builder for Totalistic and Outer-Totalistic CAs

Definition 2 (totality). A totalistic CA—or, alternatively, a CA that presents the totality property—is one whose state transitions only depend on the sum of the state values of the cells in a given neighbourhood.

Formally, let f be the local transition function of a CA, defined over the set N of possible neighborhoods for the CA's space. Let n be the size of a neighbourhood of N. Let $A = (\alpha_1, \ldots, \alpha_n)$ and $B = (\beta_1, \ldots, \beta_n)$ be neighborhoods of N. A CA is said to be totalistic if and only if the following condition is true:

$$\forall A, B \in N, \quad \left(\sum_{i=1}^{n} \alpha_i = \sum_{j=1}^{n} \beta_j\right) \Longleftrightarrow \left(f(A) = f(B)\right). \quad (9)$$

Since totality is a static property, we can derive a general algorithm to find all rule tables of CAs that share the *totality property*, given a value for k and r.

The algorithm receives as arguments the values of k and r for which the template will be generated. It then proceeds to enumerate all possible neighborhoods of the defined space and calculates their sum.

Now, for every sum value found in the previous step, the algorithm picks the neighbourhood representing the smallest k-ary number and assigns a variable corresponding to that neighbourhood's decimal value to all of the state transitions that shared a result with it. For instance, let us consider all neighborhoods of the elementary space whose states sum equals 2: $\{(1,1,0),(1,0,1),(0,1,1)\}$. Since the smallest neighbourhood in this set is $(0,1,1)$, which is decimal 3, we assign to their respective transitions the result x_3.

The template's core for the elementary space becomes

$$c\left(T_{totalisticES}\right) = \left(x_7, x_3, x_3, x_1, x_3, x_1, x_1, x_0\right). \quad (10)$$

Upon expansion, this template yields the rule tables of the 16 totalistic elementary CAs. Notice that this template's core is composed of only free variables and references to other free variables. This being the case, every expansion of this template shall yield a valid result, excluding the need to define any expansion modifier function other than $IdentityMapper$.

For $k = 3$ and $r = 1$, the resulting template would be

$$\begin{aligned} T_{tot31} = \big(&3, 1.0, \big(x_{26}, x_{17}, x_8, x_{17}, x_8, x_5, x_8, x_5, x_2, x_{17}, x_8, \\ &x_5, x_8, x_5, x_2, x_5, x_2, x_1, x_8, x_5, x_2, x_5, x_2, x_1, x_2, x_1, x_0\big), \\ &IdentityMapper\big). \end{aligned} \quad (11)$$

Upon expansion, this template yields the 2187 totalistic rules of the referred space.

Definition 3 (outer-totality). An outer-totalistic CA—or, alternatively, a CA that presents the outer-totality property—is one whose local rule considers the sum of the state values of the external cells of a neighbourhood (i.e., those except the center cell), in addition to the state value of the center cell itself.

Formally, let f be the local transition function of a CA, defined over the set N of possible neighborhoods for the CA's space. Let n be the size of a neighbourhood of N, and let $m = \lceil n/2 \rceil$ be the index of the cell that is meant to undergo a state change (i.e., the center cell when n is an odd number). Let $A = (\alpha_1, \ldots, \alpha_m, \ldots, \alpha_n)$ and $B = (\beta_1, \ldots, \beta_m, \ldots, \beta_n)$ be neighborhoods of N. A CA is said to be outer-totalistic if and only if the following condition is true:

$$\begin{aligned} &\forall A, B \in N, \\ &\left(\left(\sum_{i=1}^{m-1} \alpha_i + \sum_{i=m+1}^{n} \alpha_i = \sum_{j=1}^{m-1} \beta_j + \sum_{j=m+1}^{n} \beta_j\right)\right. \\ &\left.\wedge\left(\alpha_m = \beta_m\right)\right) \Longleftrightarrow \left(f(A) = f(B)\right). \end{aligned} \quad (12)$$

The algorithm for outer-totalistic templates is almost the same as the one for totalistic rules, with the exception that it considers the sum of external cells in the neighbourhood instead of its entirety.

The algorithm's output for the elementary space is

$$c\left(T_{outer-totalisticES}\right) = \left(x_7, x_3, x_5, x_1, x_3, x_2, x_1, x_0\right). \quad (13)$$

Upon expansion, this template generates the 64 outer-totalistic elementary CAs.

For reference, here is the result for $k = 3$ and $r = 1$ which, after expansion, would yield the 14,348,907 outer-totalistic rules of the space, since 15 distinct variables are present in the template:

$$\begin{aligned} T_{outer-tot31} = \big(&3, 1.0, \big(x_{26}, x_{17}, x_8, x_{23}, x_{14}, x_5, x_{20}, x_{11}, x_2, \\ &x_{17}, x_8, x_7, x_{14}, x_5, x_4, x_{11}, x_2, x_1, x_8, x_7, x_6, x_5, x_4, x_3, x_2, \\ &x_1, x_0\big), IdentityMapper\big). \end{aligned} \quad (14)$$

5.2. Template Builder for Captive CAs

Definition 4 (captivity [13]). A CA is said to be captive—or, alternatively, to have the captivity static property—when every transition of its rule table outputs a state that appears in the corresponding neighbourhood.

Formally, if f is the local transition function of a CA, $A = (\alpha_1, \ldots, \alpha_m, \ldots, \alpha_n)$ is a neighbourhood of size n, out of all possible neighborhoods N, and β is an arbitrary state, a CA is said to be captive if and only if the following condition is true:

$$\begin{aligned} &\forall A \in N, \\ &f(\alpha_1, \ldots, \alpha_n) = \beta \Longleftrightarrow \beta \in \{\alpha_1, \ldots, \alpha_n\}. \end{aligned} \quad (15)$$

In the binary case, this naturally simplifies to the fact that every CA with $f(0_1, 0_2, \ldots, 0_n) = 0$ and $f(1_1, 1_2, \ldots, 1_n) = 1$ is captive, for any radius r.

As a static property, a general algorithm to find templates of captive CAs for a given space does exist. As such, the algorithm receives the values for k and r and starts by generating all possible neighborhoods for the space.

It then uses the following simple rules to decide how to transform this neighbourhood in a template output:

(i) If the neighbourhood is homogeneous, like $(1,1,1)$, the transition should output the only state that appears in the neighbourhood (1).

(ii) If the neighbourhood is composed of all of the values in the interval $[0, k-1]$, like $(1,2,0)$ for $k = 3$ and $r = 1$, the transition should output a free variable, indexed by the decimal value of the neighbourhood, namely, x_{15} (since 120 in base $k = 3$ corresponds to the decimal 15).

(iii) If the neighbourhood is composed of only some of the possible states, the states of the interval $[0, k-1]$, like $(0,2,0)$ for $k = 3$ and $r = 1$, the output should be a variable indexed by the neighbourhood, this time restricted by the present values ($x_6 \in \{0,2\}$). This type of neighbourhood only appears when $k > 2$.

Note that since this template uses restricted variables, it must use the *FilterVariableAssignments* expansion modifier function.

As a result of all the latter, the captivity template for the elementary space becomes

$$T_{captive} = \left(2, 1.0, \left(1, x_6, x_5, x_4, x_3, x_2, x_1, 0\right), \\ FilterVariableAssignments\right), \tag{16}$$

while, for $k = 3$ and $r = 1$, the template is

$$\begin{aligned} T_{captive31} = \big(&3, 1.0, \big(2, x_{25} \in \{1,2\}, x_{24} \in \{0,2\}, x_{23} \\ &\in \{1,2\}, x_{22} \in \{1,2\}, x_{21}, x_{20} \in \{0,2\}, x_{19}, x_{18} \\ &\in \{0,2\}, x_{17} \in \{1,2\}, x_{16} \in \{1,2\}, x_{15}, x_{14} \\ &\in \{1,2\}, 1, x_{12} \in \{0,1\}, x_{11}, x_{10} \in \{0,1\}, x_9 \\ &\in \{0,1\}, x_8 \in \{0,2\}, x_7, x_6 \in \{0,2\}, x_5, x_4 \\ &\in \{0,1\}, x_3 \in \{0,1\}, x_2 \in \{0,2\}, x_1 \in \{0,1\}, 0\big), \\ &FilterNotAllowed\big). \end{aligned} \tag{17}$$

5.3. Template Builder for Colorblind CAs. In order to fully grasp the notion of colorblind rules, let us first discuss the meaning of permutations in the context of CAs.

Definition 5 (colour permutation (π)). Let framework K be the set of possible states for a CA in the range $[0, k-1]$. A colour permutation π is a bijective function of K to itself, which can be written as a set of rules of the form $k_i \longrightarrow k_j$, meaning that a cell in state k_i should be switched to state k_j when this permutation is applied.

For instance, a valid permutation for the case where $K = \{0, 1, 2\}$ is $\pi_1 = \{0 \longrightarrow 1, 1 \longrightarrow 2, 2 \longrightarrow 0\}$.

It is important to notice that every colour permutation can be represented by an algebraic function that interpolates the points defined by the permutation. For instance, π_1 could be written as $\pi_1(x) = 1 + (1 - 3/2(-1 + x))x$.

Definition 6 (symmetric group (S_K)). The symmetric group S_K, of a set K, is the set of all possible permutations over K.

For instance, the symmetric group for $K = \{0, 1, 2\}$ is

$$\begin{aligned} S_K = \{&\{0 \longrightarrow 0, 1 \longrightarrow 1, 2 \longrightarrow 2\}, \\ &\{0 \longrightarrow 0, 1 \longrightarrow 2, 2 \longrightarrow 1\}, \\ &\{0 \longrightarrow 1, 1 \longrightarrow 0, 2 \longrightarrow 2\}, \\ &\{0 \longrightarrow 1, 1 \longrightarrow 2, 2 \longrightarrow 0\}, \\ &\{0 \longrightarrow 2, 1 \longrightarrow 0, 2 \longrightarrow 1\}, \\ &\{0 \longrightarrow 2, 1 \longrightarrow 1, 2 \longrightarrow 0\}\}. \end{aligned} \tag{18}$$

Definition 7 (neighbourhood permutation ($\rho(\pi)$)). Let π be a colour permutation and let N be the set of all possible neighborhoods for a given CA. The neighbourhood permutation $\rho(\pi)$ is a permutation N that uses π to transform $n_i \in N$ into neighbourhood $n_j \in N$, by applying π to every cell of n_i.

For instance, with $K = \{0, 1, 2\}$, $r = 1$, and $\pi_1 = \{(0, 1, 2)\}$, a valid transformation due to $\rho(\pi_1)$ would be $(0, 2, 2) \longrightarrow (1, 0, 0)$. The other transformations can be derived by applying the same reasoning to all of the neighborhoods of N.

Definition 8 (cycle notation of a permutation). A permutation can be written by means of a cycle notation, formed by a set of vectors that represent cycles associated with the permutation.

For instance, the colour permutation $\pi_2 = \{0 \longrightarrow 1, 1 \longrightarrow 0, 2 \longrightarrow 2\}$ can be written as $\pi_2 = \{(0, 1), (2)\}$, where each vector has the form $(x_{i_1}, x_{i_2}, \ldots, x_{i_j})$, $1 \leq j \leq k$, and represents a cycle due to the permutation, namely, the two-step cycle $0 \longrightarrow 1 \longrightarrow 0$, and the self-loop $2 \longrightarrow 2$. Similarly, The previous π_1 permutation would be written in cycle notation as $\pi_1 = \{(0, 1, 2)\}$.

As for neighbourhood permutations, $\rho(\pi_1)$, for example, would be represented as follows:

$$\begin{aligned} \rho(\pi_1) = \{&((0,2,2), (1,0,0), (2,1,1)), \\ &((0,2,1), (1,0,2), (2,1,0)), \\ &((0,2,0), (1,0,1), (2,1,2)), \\ &((0,1,2), (1,2,0), (2,0,1)), \\ &((0,1,1), (1,2,2), (2,0,0)), \\ &((0,1,0), (1,2,1), (2,0,2)), \\ &((0,0,2), (1,1,0), (2,2,1)), \\ &((0,0,1), (1,1,2), (2,2,0)), \\ &((0,0,0), (1,1,1), (2,2,2))\}. \end{aligned} \tag{19}$$

With the previous characterisations, we can now define color blindness.

Definition 9 (color blindness [14]). A CA whose cells can be in any of the states of the set K is said to be colorblind—or, alternatively, to possess the color blindness property—if its space-time evolution does not change, unless by a colour change, after applying any neighbourhood permutation $\rho(\pi), \pi \in S_K$, to all state transitions that define its rule table.

Notice that, for a CA to be invariant to $\rho(\pi_1)$, its rule table's outputs o_i should be consistent with the cycles formed by the application of $\rho(\pi_1)$ over its neighborhoods b_i, i being the index of the state transition on the CA's rule table.

In other words, for a rule to be invariant to π_1, the output of, say, neighbourhood $(0, 2, 2)$, should be equal to the application of π_1 to the output of neighbourhood $(1, 0, 0)$, since $(0, 2, 2)$ is transformed into $(1, 0, 0)$ due to π_1; for the same reason, the output of $(1, 0, 0)$ should be equal to the application of π_1 to the output of neighbourhood $(2, 1, 1)$. The same happens, analogously, for the remaining cycles of $\rho(\pi_1)$.

Now, in order to generate a template representing CAs that are invariant to the neighbourhood permutation $\rho(\pi_1)$, we need to assign an output to every neighbourhood and derive a rule table that could be turned into a template.

So that outputs follow $\rho(\pi_1)$, we use its cycle notation representation to assign to every cycle start a free variable, indexed by its corresponding neighbourhood's decimal representation, ending up with a tuple formed by neighbourhood and output. The $(0, 2, 2)$ neighbourhood, being a cycle start, is turned into the tuple $((0, 2, 2), x_8)$, for instance.

For every remaining neighbourhood in a cycle, we now apply π_1 to the free variable found in the previous step d times, d being the distance from this neighbourhood to the start of the cycle. The cycle $((0, 2, 2), (1, 0, 0), (2, 1, 1))$ is turned into $(((0, 2, 2), x_8), ((1, 0, 0), \pi_1(x_8)), ((2, 1, 1), \pi_1(\pi_1(x_8))))$, for instance.

Applying this procedure to every cycle of π_1, we obtain the set:

$$\begin{aligned}
&\{(((0,2,2),x_8),((1,0,0),\pi_1(x_8)),\\
&\quad((2,1,1),\pi(\pi_1(x_8)))),(((0,2,1),x_7),\\
&\quad((1,0,2),\pi_1(x_7)),((2,1,0),\pi(\pi_1(x_7)))),\\
&\quad(((0,2,0),x_6),((1,0,1),\pi_1(x_6)),\\
&\quad((2,1,2),\pi(\pi_1(x_6)))),(((0,1,2),x_5),\\
&\quad((1,2,0),\pi_1(x_5)),((2,0,1),\pi(\pi_1(x_5)))),\\
&\quad(((0,1,1),x_4),((1,2,2),\pi_1(x_4)),\\
&\quad((2,0,0),\pi(\pi_1(x_4)))),(((0,1,0),x_3),\\
&\quad((1,2,1),\pi_1(x_3)),((2,0,2),\pi(\pi_1(x_3)))),\\
&\quad(((0,0,2),x_2),((1,1,0),\pi_1(x_2)),\\
&\quad((2,2,1),\pi(\pi_1(x_2)))),(((0,0,1),x_1),\\
&\quad((1,1,2),\pi_1(x_1)),((2,2,0),\pi(\pi_1(x_1)))),\\
&\quad(((0,0,0),x_0),((1,1,1),\pi_1(x_0)),\\
&\quad((2,2,2),\pi(\pi_1(x_0))))\}.
\end{aligned} \tag{20}$$

We can now flatten a level of this set and turn it into an ordered list sorted lexicographically by neighbourhood. This procedure yields the rule table shown in Table 2.

The core of a template for CAs invariant to $\rho(\pi_1)$ can now be found by compressing this rule table into its k-ary form:

$$\begin{aligned}
T_{\rho(\pi_1)} = {} &(\pi(\pi_1(x_0)),\pi(\pi_1(x_2)),\pi(\pi_1(x_1)),\\
&\pi(\pi_1(x_6)),\pi(\pi_1(x_8)),\pi(\pi_1(x_7)),\pi(\pi_1(x_3)),\\
&\pi(\pi_1(x_5)),\pi(\pi_1(x_4)),\pi_1(x_4),\pi_1(x_3),\pi_1(x_5),\\
&\pi_1(x_1),\pi_1(x_0),\pi_1(x_2),\pi_1(x_7),\pi_1(x_6),\pi_1(x_8),x_8,\\
&x_7,x_6,x_5,x_4,x_3,x_2,x_1,x_0))
\end{aligned} \tag{21}$$

Remember from Definition 5 that any permutation π can be represented by an algebraic function, which is the actual representation used by the algorithm. We use the π notation here since it is compact and serves the explanation's purpose.

Table 2: Rule table for the template representative of CAs invariant to π_1.

(2, 2, 2)	$\pi(\pi_1(x_0))$
(2, 2, 1)	$\pi(\pi_1(x_2))$
(2, 2, 0)	$\pi(\pi_1(x_1))$
(2, 1, 2)	$\pi(\pi_1(x_6))$
(2, 1, 1)	$\pi(\pi_1(x_8))$
(2, 1, 0)	$\pi(\pi_1(x_7))$
(2, 0, 2)	$\pi(\pi_1(x_3))$
(2, 0, 1)	$\pi(\pi_1(x_5))$
(2, 0, 0)	$\pi(\pi_1(x_4))$
(1, 2, 2)	$\pi_1(x_4)$
(1, 2, 1)	$\pi_1(x_3)$
(1, 2, 0)	$\pi_1(x_5)$
(1, 1, 2)	$\pi_1(x_1)$
(1, 1, 1)	$\pi_1(x_0)$
(1, 1, 0)	$\pi_1(x_2)$
(1, 0, 2)	$\pi_1(x_7)$
(1, 0, 1)	$\pi_1(x_6)$
(1, 0, 0)	$\pi_1(x_8)$
(0, 2, 2)	x_8
(0, 2, 1)	x_7
(0, 2, 0)	x_6
(0, 1, 2)	x_5
(0, 1, 1)	x_4
(0, 1, 0)	x_3
(0, 0, 2)	x_2
(0, 0, 1)	x_1
(0, 0, 0)	x_0

The same procedure can now be repeated for every permutation of the S_k set, essentially mapping each permutation of the set to a corresponding template of CAs invariant to that permutation. It is now only a matter of intersecting all of the templates found, and the result is a template that represents all colorblind rules of the space defined by k and r.

When applied to $k = 2$ and $r = 1$, we obtain

$$\begin{aligned}
&T_{colourblind}\\
&\quad = (1-x_0, 1-x_1, 1-x_2, 1-x_3, x_3, x_2, x_1, x_0),
\end{aligned} \tag{22}$$

whereas, for $k = 3$ and $r = 1$, the result becomes

$$\begin{aligned}
T_{colourblind31} = {} &(2, 2-x_1, 2-2x_1, 2-x_3, 2-x_4, 2\\
&-x_5, 2-2, x_3, 2-2x_5, 2-2x_4, 1-2x_4, 1-2x_3, 1\\
&-2x_5, 1-2x_1, 1, 1+2x_1, 1+2x_5, 1+2x_3, 1\\
&+2x_4, 2x_4, 2x_5, 2x_3, x_5, x_4, x_3, 2x_1, x_1, 0).
\end{aligned} \tag{23}$$

6. A Detailed Example

Here we discuss a detailed application for the template framework presented in this paper, which is to find the rules that share a set of static properties. This can be achieved with the template builder functions provided with *CATemplates*, or one could implement custom builder functions on top of the core concepts provided by the package. What follows is a sample of how the framework can be used to conduct an experiment of this sort.

More specifically, let us suppose we are interested in the rules which are both totalistic and captive. This might come from a purely theoretical interest, or motivated by the need for obtaining rules of this type for some practical application.

We then start by building the templates for totalistic and captive rules of the elementary space, as shown in the previous sections ($T_{totalistic}$ and $T_{captive}$). By performing their intersection, we end up with a template representing all totalistic and captive rules of the elementary space, as follows:

$$T_{k_2 r_1} = \left(2, 1.0, \left(1, x_3, x_3, x_1, x_3, x_1, x_1, 0\right), \\ FilterVariableAssignments\right). \tag{24}$$

Upon expansion, this template yields the 4 totalistic elementary CAs which are also captive: {128, 150, 232, 254}. Notice that the template being expanded has only 2 variables, which means the expansion has to consider only 4 expansion candidates, which is a substantially smaller subset of the original space, which has 256 rules.

Since the elementary space is a small one, let us perform the same example in larger spaces. We can generate the equivalent templates for the space defined by $k = 3$ and $r = 1$ and perform the intersection between them. The result is

$$T_{k_3 r_1} = \left(3, 1.0, \left(2, x_{17} \in \{1, 2\}, 2, x_{17} \right.\right. \\ \in \{1, 2\}, 2, 1, 2, 1, 0, x_{17} \in \{1, 2\}, 2, 1, 2, 1, 0, 1, 0, x_1 \\ \left.\in \{0, 1\}, 2, 1, 0, 1, 0, x_1 \in \{0, 1\}, 0, x_1 \in \{0, 1\}, 0\right) \\ \left. FilterVariableAssignments\right). \tag{25}$$

Notice that this template also has two variables: x_1 and x_{17}. This means there are only $3^2 = 9$ possible expansions for this template, from which some will be filtered out because of the variable assignment annotations. After expansion, this template yields the 4 totalistic rules of the space which are also captive: {223160992, 223161514, 265235104, 265235626}.

This means that we took a space with $3^{27} = 7,625,597,484,987$ rules and found out that only 4 of those share the totality and captivity properties, all without actually enumerating the whole space (which would be not feasible, given its size). Notice that this space could be further reduced, by intersecting this resulting template with templates from other properties, like color blindness.

Extending the previous case, we may now ask the general question of up to which rule space size could we still manage to get access to totalistic and captive rules. Without templates, the rules would have to be obtained directly by scanning the rule space and selecting those having the two properties. But with $k = 3$ this approach would already be impractical even for $r = 1$ on a standard personal computer, since the space has $\simeq 7.6 \times 10^{12}$ rules. A first alternative, now using templates, could be accessing the rules, say, by first expanding the totalistic rules directly from the corresponding template and then filtering out the captive ones. This certainly helps. In fact, by doing so, progressively, starting with radius $r = 0.5$ and increasing its value in steps of 0.5, in a standard personal computer it would hardly be possible to reach beyond $r = 2$ (which corresponds to a space with $\simeq 8.7 \times 10^{115}$ rules), because of the substantial increase in computational time as we would go from a given r to its subsequent value. We omit actual values of processing time here, as this may strongly vary among different machine architectures and their software environment.

However, by relying on the generation of the individual templates of each property and expanding the intersection between them, we easily access spaces up to $r = 4$ (possibly even beyond), which contains $\simeq 1.5 \times 10^{9391}$ rules. It turns out that, in this space, only 256 rules share the two properties, an extremely smaller space to reason about.

Furthermore, the analysis of all the data generated from the previous experiment suggests that the number of rules with $k = 3$ and the shared properties of totality and captivity seems to be given by 4^r for symmetric rules (those with integer radius), whereas no rule would be present for asymmetric rules. This is quite a remarkable general result, derived from direct probing of the corresponding rule spaces that the full usage of templates allowed us.

As a final note, it is worth emphasising that template expansion is only made when the actual rules represented in a template are needed, which is at the end of the process of answering which rules satisfy a given set of properties. Until then, it suffices to operate directly with the templates; this is where the conciseness of template usage lies. And although expansion may be computationally intensive, it is certainly much less than if the entire corresponding spaces would have to be manipulated directly all the way.

7. Concluding Remarks

Searching for cellular automata with specific behaviours (such as global problem solvers) is a daunting task for anyone looking further than the elementary space.

In this paper, we presented the current state of cellular automata templates, a valuable ally in this front. We have described what templates are and how the two currently implemented operations (expansion and intersection) work, went into details of the implementations of template builder functions for captive, totalistic, outer-totalistic, and color-blind rules, and have shown a sample application of the framework established here.

The sample application has led us to the discovery that only a handful of CAs share both of the properties of totality and captivity (only 64 rules on the $k = 3$, $r = 3$ space, for instance). This little experiment shows how the proposed framework can be used as a means to sieve through very large spaces when searching for CAs that present a given behaviour.

As mentioned in distinct parts of the text, the notion of templates and the implementation of the computational system to support it underwent a long way since its origin, up to its current degree of structural maturity.

The natural course from here is to develop new template builders and whichever other operations we might conceive. More concretely with respect to the latter, we should say that during the last couple of years we have been investigating the notion of "difference" between templates, an operation whose output would yield another template representing the rules that have the property associated with the first template, but not having the property associated with the second. In spite of a conference publication we have on the topic (see [15]), it is not yet mature to go into the publicly available package. But in due time, we intend to do so.

As for new template builders, a generalisation of one to handle arbitrary values of symmetry by conjugation is more pressing. As mentioned earlier, we decided not to include this template builder mostly because the existing one cannot yet handle arbitrary number of states in the case of conjugation. However, noticing that the notion of color blindness is essentially the notion of rules with maximum symmetry by conjugation, generalised for any k-state rule, using the same idea implemented for color blindness it is now possible to face the generalisation of arbitrary values of symmetry by conjugation. This is clearly future work to be done.

Also, it is tempting to add completely new template builders to the present package, referring to other parameters, like those in [16]. In fact, we hope that this paper triggers collaborative participation from the cellular automata community in terms of both the system's expansion and its use in applications. This is why we decided to keep the work in GitHub, publicly available.

As for the possibility of extending the framework to support rules in higher dimensions, we would say this would be readily possible for all current properties but one: number conservation. This is due to the fact that the underlying algorithm we used to implement the corresponding template builder comes from [17], which is restricted to one-dimensional rules. In order to change that, a general algorithm would have to be used, quite likely the one described in [18]. This is certainly feasible and desirable, but we had not yet had the opportunity to go about that.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request, and the CATemplates Mathematica package, directly from GitHub [10].

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

The authors are grateful to financial support from IPM (Instituto Presbiteriano Mackenzie) and CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior, Brazil), by means of STIC-AmSud (CoDANet) project no. 88881.197456/2018-01 and PrInt project no. 88887.310281/2018-00.

References

[1] S. Wolfram, *A New Kind of Science*, Wolfram Media Inc, 2002.

[2] J. L. Zapotecatl, D. A. Rosenblueth, and C. Gershenson, "Deliberative self-organizing traffic lights with elementary cellular automata," *Complexity*, vol. 2017, Article ID 7691370, 15 pages, 2017.

[3] D. Wolz and P. P. B. de Oliveira, "Very effective evolutionary techniques for searching cellular automata rule spaces," *Journal of Cellular Automata*, vol. 3, no. 4, pp. 289–312, 2008.

[4] I. Barragan-Vite, J. C. Seck-Tuoh-Mora, N. Hernandez-Romero, J. Medina-Marin, and E. S. Hernandez-Gress, "Distributed control of a manufacturing system with one-dimensional cellular automata," *Complexity*, vol. 2018, Article ID 7235105, 15 pages, 2018.

[5] J. Yang, W. Liu, Y. Li, X. Li, and Q. Ge, "Simulating intraurban land use dynamics under multiple scenarios based on fuzzy cellular automata: a case study of Jinzhou District, Dalian," *Complexity*, vol. 2018, Article ID 7202985, 17 pages, 2018.

[6] H. Betel, P. P. B. De Oliveira, and P. Flocchini, "Solving the parity problem in one-dimensional cellular automata," *Natural Computing*, vol. 12, no. 3, pp. 323–337, 2013.

[7] M. Bidlo, "On routine evolution of complex cellular automata," *IEEE Transactions on Evolutionary Computation*, vol. 20, no. 5, pp. 742–754, 2016.

[8] P. de Oliveira and M. Verardo, "Representing families of cellular automata rules," *The Mathematica Journal*, vol. 16, no. 8, 2014.

[9] Wolfram Research, "Wolfram mathematica 11," http://www.wolfram.com/mathematica/, 2018.

[10] M. Verardo and P. P. B. de Oliveira, "CATemplates," https://github.com/mverardo/CATemplates, 2019.

[11] P. P. B. de Oliveira, "On density determination with cellular automata: Results, constructions and directions," *Journal of Cellular Automata*, vol. 9, no. 5-6, pp. 357–385, 2014.

[12] P. P. B. de Oliveira and M. Verardo, "Template based representation of cellular automata rules," in *Proceedings of the Exploratory Papers - AUTOMATA 2014 - 20th International Workshop on Cellular Automata and Discrete Complex Systems*, T. Isokawa, K. Imai, N. Matsiu, F. Peper, and H. Umeo, Eds., pp. 199–204, 2014.

[13] G. Theyssier, "Captive cellular automata," in *Mathematical Foundations of Computer Science*, vol. 3153 of *Lecture Notes in Computer Science*, pp. 427–438, 2004.

[14] V. Salo and I. Törmä, "Color blind cellular automata," in *Cellular Automata and Discrete Complex Systems*, J. Kari, M. Kutrib, and A. Malcher, Eds., pp. 139–154, Springer, Heidelberg, Berlin, 2013.

[15] Z. Soares, M. Verardo, and P. P. de Oliveira, "The difference operation between templates of binary cellular automata," in *New Advances in Information Systems and Technologies*, Á. Rocha, A. M. Correia, H. Adeli, L. P. Reis, and M. M. Teixeira, Eds., vol. 444 of *Advances in Intelligent Systems and Computing*, pp. 707–715, Springer International Publishing, Cham, Swizerland, 2016.

[16] G. M. Oliveira, P. P. Oliveira, and N. Omar, "Definition and application of a five-parameter characterization of one-dimensional cellular automata rule space," *Artificial Life*, vol. 7, no. 3, pp. 277–301, 2001.

[17] N. Boccara and H. Fuks, "Number-conserving cellular automaton rules," *Fundamenta Informaticae*, vol. 52, no. 1-3, pp. 1–13, 2002.

[18] B. Durand, E. Formenti, and Z. Róka, "Number-conserving cellular automata I: decidability," *Theoretical Computer Science*, vol. 299, no. 1-3, pp. 523–535, 2003.