Computer Science Journal of Moldova, vol.**29**, no.1(85), 2021

A Sublinear Sudoku Solution in cP Systems and its Formal Verification*

Yezhou Liu✉, Radu Nicolescu, Jing Sun, Alec Henderson

Abstract

Sudoku is known as a NP-complete combinatorial number-placement puzzle. In this study, we propose the first cP system solution to generalised Sudoku puzzles with $m \times m$ cells grouped in m blocks. By using a fixed constant number of rules, our cP system can solve all Sudoku puzzles in sublinear steps. We evaluate the cP system and discuss its formal verification.

Keywords: cP systems, P systems, Sudoku problem, NP-complete problem, Formal verification.

MSC 2020: 68Q07, 68N30.

1 Introduction

Sudoku is a famous number-placement puzzle designed for a single player, which has $m \times m$ cells divided into m blocks. A solvable Sudoku puzzle may have one or multiple solutions. In a valid Sudoku solution, each row, column and block contains exactly one of each number from 1 to m. For all the classic 9×9 Sudoku puzzles, there are approximately 6.67×10^{21} valid solutions [1].

Different algorithms can be used to solve Sudoku puzzles, which include: backtracking [2], stochastic search [3], evolutionary algorithms [4]–[7], propositional satisfiability inference techniques [8], constraint algorithms [9], [10] and rewriting rules [11]. Many existing solvers are designed to solve specific Sudoku instances, and their performance is related to the difficulties of these instances.

©2021 by CSJM; Yezhou Liu, Radu Nicolescu, Jing Sun, Alec Henderson

*This paper is based on our presentation at the 6th annual Rogojin lectures, November 15, 2019, Chisinau, Moldova.

Membrane computing systems (P systems) were inspired by the biological structure of living cells [12]. Major P system variants have theoretically unlimited computational power and memory, and can compute multiple tasks in a maximally parallel manner, which include but not limited to: P system with active membranes [13], tissue P systems [14], spiking neural P systems [15], kernel P systems [16], and P systems with complex objects (cP systems) [17], [18]. According to the membrane structure, these P system variants can be roughly classified into three categories: cell-like P systems, tissue-like P systems and neural-like P systems.

The first P study of Sudoku used a family of P systems containing enzymatic, dissolution and send-out rules [19]. The proposed P systems can either solve a Sudoku instance, or detect the drawback and stop the computation. Later, the work was extended by adding some solving strategies and a brute-force algorithm [20]. Another Sudoku solution used cell-like P systems with the rules of particle swarm optimization to solve Sudoku instances [21]. The authors classified the instances based on the difficulty, and evaluated their P systems on success rate and running time.

As a recently proposed P system variant, cP systems support complex symbols and generic rules, which can use a fixed constant number of rules to solve computationally hard problems efficiently. In this study, we provide a cP system solution to general Sudoku puzzles. Compared to other P system solutions, our cP system solution consists of only 16 rules (by using relations as special promoters), which can solve all $m \times m$ Sudoku puzzles in $3m + 7$ steps regardless of their difficulty. Considering an input size of m^2, our cP solution is sublinear. We use the model checker PAT3 [22] to formally verify the cP solution.

We organize this paper as follows. Section 2 reviews cP systems' syntax and the model checker PAT3. Section 3 introduces our Sudoku solution. Section 4 shows a worked example of the solution. Section 5 discusses its formal verification. Section 6 discusses existing P system Sudoku solutions. Section 7 concludes the work with future directions.

2 Background

cP systems share advantages of both traditional cell-like P systems [12] and tissue P systems [14]. One cP system may have one or multiple top-cells, where each top-cell can contain a number of nested sub-cells. Top-cells in cP systems have multiset rewriting rules, and sub-cells are only used to represent local data [18]. In previous studies, cP systems were used to solve multiple computationally hard problems, which include the subset sum problem [23], Hamiltonian cycle problem [24], travelling salesman problem [24], and quantified SAT problem [25].

To formally verify cP systems, we can apply model checking techniques. A model checker can simulate a finite-state system, and exhaustively search its state-space to check if the system can meet some given specifications. We use the model checker PAT3 to verify our cP Sudoku solution, where its rules are translated into CSP# descriptions, and its properties are specified in temporal logic.

2.1 cP system notation

The syntax of cP systems is shown in Fig. 1. Our basic vocabulary consists of *atoms* and *variables*, collectively known as *simple terms*. We use lowercase letters to represent atoms and uppercase letters to denote variables. For instance, a, b, c are atoms and X, Y, Z are variables. To represent numbers, we use a unity symbol *1*, which can be treated as a special atom. Underscores (_) are used to denote anonymous variables.

Compound terms are recursively built by labelled multisets of other compound or simple terms with *functors*, where functors are atoms. For instance, $a(b)$, $a(b(1)c(X))$, $f(g(a)b_)$ are compound terms. In cP systems, cells and their contents are represented as compound terms. For example, a cell with label a which contains two atoms b and c can be presented as $a(bc)$; a cell with label d which contains three unity symbols can be represented as $d(111)$ or $d(1^3)$. We can either represent numbers in unary as *1*, *11*, *111*..., 1^n; or in decimal as 1, 2, 3..., n.

Since all the terms in cP system are multiset-based, the order of their sub-terms does not matter. For example, to represent a cell la-

< *term* > ::= < *simple-term* > | < *compound-term* >
< *simple-term* > ::= < *atom* > | < *variable* >
< *compound-term* > ::= < *functor* > (< *argument* >)
< *functor* > ::= < *atom* >
< *state* > ::= < *l-state* > | < *r-state* >
< *l-state* > ::= < *atom* >
< *r-state* > ::= < *atom* >
< *argument* > ::= < *term* > ...
< *rule* > ::= < *lhs* >$\rightarrow_\alpha$< *rhs* >< *promoters* >
< *lhs* > ::= < *l-state* >< *term* > ...
< *rhs* > ::= < *r-state* >< *term* > ...
< *promoters* > ::= | < *term* > ...| < *relation* > ...

Figure 1. cP system syntax (lhs = left-hand-side, rhs = right-hand-side, α = rule application model)

belled as f which contains two simple terms g and h, we can either write $f(gh)$ or $f(hg)$, where the order of g and h does not matter. Similarly, the terms $a(bcd)$, $a(bdc)$ and $a(dcb)$ are identical.

Only top-cells in cP systems contain rewriting rules, each top-cell may have one or multiple rules. A rule consists of a lhs, a rhs and an *application model* α; both its lhs and rhs contain a state and zero or more terms. For example: $s_1\ a\ \ \rightarrow_1\ \ s_2\ bc$ is a cP rule, where s_1 is its *l-state*, s_2 is its *r-state*, a is a term of its lhs, b and c are terms of its rhs. The application model of this rule is 1, which means "exactly-once". The rule can consume a term a and produce two terms b and c – in other words, it can rewrite a as bc.

Every cP top-cell has a *state* alternatively named *system state*. States are atoms, to distinguish them with lhs and rhs terms in rules, we often use atom s with subscripts to represent states, for example: s_1, s_2 and s_3.

A rule is applicable if and only if its *l-state* matches the system state, and its lhs terms can be found in the system. After applying it, the system state will be changed to the rule's *r-state*. Suppose we have a cP system at state s_1, which has a term a and a rule $s_1\ a\ \ \rightarrow_1\ \ s_2\ bc$. Since the rule's *l-state* matches the system state, and the system does

contain a term a, the rule is applicable. After it is being applied, the term a will be consumed, two terms b and c will be generated, and the system state will be changed to s_2.

For generic rules with variable terms such as $a(X)$, $b(Y)$, a *one-way unification* (pattern matching) is supported in cP systems. Before applying a rule, its variable terms must be unified against terms in the system. Suppose a cP system at state s_1 that has two terms $a(1)$, $b(11)$ and a generic rule $s_1\ a(X)\ b(Y) \rightarrow_1 s_2\ c(XY)$. Variable terms $a(X)$ and $b(Y)$ will be unified against terms $a(1)$ and $b(11)$. In this example, we can get $X \mapsto 1$, $Y \mapsto 11$. So the rule will be unified as $s_1\ a(1)\ b(11) \rightarrow_1 s_2\ c(111)$. By applying it, the two terms $a(1)$ and $b(11)$ will be consumed, and a term $c(111)$ will be generated. The system state will be changed from s_1 to s_2.

Two major *application models* are supported in cP systems, which are "*exactly-once* (1)" and "*max-parallel* (+)". In the exactly-once model, a rule will only apply once. In the max-parallel model, a rule will apply to all possible terms simultaneously. Suppose a cP system at state s_1 that has three terms $a(1^2)$, $a(1^3)$, $a(1^3)$ and a rule $s_1\ a(1X) \rightarrow_\alpha s_2\ a(X)$. The rule can be unified to three ground rules (ground here means "without variables"), which are $s_1\ a(11) \rightarrow_\alpha s_2\ a(1)$, $s_1\ a(11^2) \rightarrow_\alpha s_2\ a(1^2)$ and $s_1\ a(11^2) \rightarrow_\alpha s_2\ a(1^2)$, where the variable X is unified as 1, 1^2 and 1^2, respectively. When the application model α of the rule is "1" (exactly-once model), the system will *non-deterministically* choose one unified rule to apply. The computation result will be $a(1)$, $a(1^3)$, $a(1^3)$ or $a(1^2)$, $a(1^2)$, $a(1^3)$. If α in the rule is "+" (max-parallel model), the system will apply all three unified rules, the computation result will be $a(1)$, $a(1^2)$, $a(1^2)$.

For a cP system with a max-parallel rule, the unified rules which can be applied together are called *compatible*. Suppose a cP system at state s_1 has four terms $a(c)$, $a(d)$, $b(e)$, $b(f)$, and a rule $s_1\ a(X)b(Y) \rightarrow_+ s_1\ g(XY)$. To unify the rule against the system terms, we can get the following unified rules: *r1*: $s_1\ a(c)b(e) \rightarrow_+ s_1\ g(ce)$, *r2*: $s_1\ a(c)b(f) \rightarrow_+ s_1\ g(cf)$, *r3*: $s_1\ a(d)b(e) \rightarrow_+ s_1\ g(de)$, and *r4*: $s_1\ a(d)b(f) \rightarrow_+ s_1\ g(df)$. When applicable, these unified rules will be non-deterministically chosen by the system. Suppose it chooses

r2 to apply, the system terms $a(c)$ and $b(f)$ will be "locked" by *r2*, which will be used to consume and to produce the term $g(cf)$ later. So they cannot be used by other unified rules any more. The only free terms in the system are $a(d)$ and $b(e)$, which can be used in *r3*. Thus, *r2* and *r3* can be applied together – they are compatible. Similarly, *r1* and *r4* are compatible. In the max-parallel model, the system will non-deterministically choose *r2*, *r3* OR *r1*, *r4* to apply.

cP systems apply rules following a *weak priority* order – i.e., rules are sequentially considered in the top-down order. The first applied rule commits the target state, any subsequent rule that indicates a different target state is then disabled. This way, the weak priority order can be used to simulate *if-then-else* structures of traditional programming.

Suppose a cP system at state s_1 that has two terms $a(c)$, $b(d)$ and three rules: *r1*: $s_1\ a(X) \rightarrow_1 s_2\ o(XX)$, *r2*: $s_1\ b(X) \rightarrow_1 s_3\ p(X)$, and *r3*: $s_1\ b(X) \rightarrow_1 s_2\ q(XXX)$. The system will first consider *r1* and find if it is applicable. Thus, the target state will be confirmed as $s2$. Since *r2* commits to a different target state s_3, it is not applicable. *r3* commits to s_2, and it is compatible with *r1*, so it will be applied with *r1* together in the same *step*. The computational result of the system will be $o(cc)$, $q(ddd)$.

In each step, new generated terms will be temporarily put into a "product membrane", which will not be available until the next step. For example, a cP system has two terms $a(c)$, $b(d)$ and two rules *r1*: $s_1\ a(X) \rightarrow_1 s_2\ b(X)$ and *r2*: $s_1\ b(X) \rightarrow_+ s_2\ c(X)$. *r1* and *r2* will be applied in the same step. The term $b(c)$ generated by *r1* will be sent to the product membrane, which will not be consumed by *r2* in the same step. After applying them, the system state will be changed to s_2, then none of them are applicable. The computational result of the system will be $b(c)$, $c(d)$.

To apply a rule with promoters, the promoters must exist in the system, and will not be consumed. Suppose a cP system has a rule $s_1 \rightarrow_1 s_2\ x(X) \mid y(XZ)\ z(Z)$, and two terms $y(6)$ and $z(4)$. The rule can be unified as $s_1 \rightarrow_1 s_2\ x(2) \mid y(6)\ z(4)$. By applying it, a term $x(2)$ will be generated. $y(6)$ and $z(4)$ are promoters, they will be checked by the rule, but will not be consumed.

For readability, we can use two kinds of delimiters in cP terms as needed, which are blank space “ ” and comma “,”. Adding delimiters to a term will not affect its meaning. For example, $a(bc)$, $a(b\ c)$ and $a(b, c)$ represent the same term.

To simplify the cP encoding of Sudoku, we can use the following abbreviation: $a(X)(Y) \equiv a(a_1(X)a_2(Y))$, when the sub-cell names a_1 and a_2 are unimportant. Similarly, we can use $b(X)(Y)(Z)$ to represent $b(b_1(X)b_2(Y)b_3(Z))$ as needed.

In this study, to make our rules more readable, we use logic *relations* – including *multiset inclusion* ($\subseteq$), *multiset NOT inclusion* ($\nsubseteq$), and *multiset inequality* ($\neq$) – as special promoters. This design can be translated into classical cP systems by adding a few more rules.

For example, to test the inclusion relation ($\subseteq$) – e.g., $aab \subseteq aabcd$, we can use a rule fragment (stateless) $\rightarrow_1 \quad \mathit{1} \mid aab$. If the system contains the multiset $aabcd$, then the rule fragment is applicable, and it will generate a symbol *1* to indicate that aab is a submultiset of $aabcd$. If the multiset which needs to be checked is not a supermultiset of aab, for instance $cdef$, the rule fragment will not be applicable, and the symbol *1* will not be generated.

Another example is to test the NOT inclusion relation ($\nsubseteq$) – e.g. $aab \nsubseteq abcd$, we can use two rules: *r1*: $s_1 \rightarrow_1 s_2 \mid aab$ and *r2*: $s_1 \rightarrow_1 s_3\ \mathit{1}$. Suppose the cP system starts at state s_1, and contains the multiset $abcd$. It will consider *r1* first, but it cannot find all *r1*'s promoters – thus *r1* is not applicable. Then the system will consider *r2*, which is applicable. *r2* will generate a symbol *1* to indicate that aab is not a submultiset of $abcd$. By using the same rules, if the multiset that needs to be checked is a supermultiset of aab, for instance $aabbc$, rule *r1* – which appears before *r2*, thus has a higher priority – is applicable, it will change the system state to s_2, then no more rules are applicable, and the symbol *1* will not be generated.

2.2 The model checker PAT3

PAT3 provides an extensible framework for simulating and verifying different systems in multiple application domains [22]. Previous research

showed that PAT3 can effectively verify cP models, and transformation guidelines from cP syntax to CSP# were proposed [23].

PAT3 supports several semantic models and modelling languages, which include Communicating Sequential Processes (CSP), Real-Time Systems (RTS), Labeled Transition Systems (LTS), and Timed Automata (TA). To improve PAT3's performance, the authors implemented a number of model checking algorithms, state reduction techniques and abstraction techniques in it.

A specification language called CSP# (Communicating Sequential Programs) is supported by PAT3. CSP#, an extension of CSP, combines high-level modeling operators (e.g. choices, interrupt, parallel composition, asynchronous message passing) and low-level constructs (e.g. data structures and conditional statements) together. CSP# is especially good at representing cP systems, which has great potential for simulating the cell communication among multiple top-cells.

To describe features of cP systems, we can either use Linear Temporal Logic (LTL) or Computation Tree Logic (CTL). In addition to specifying features by users, some commonly checked features are pre-implemented in PAT3.

Similar to other model checkers, PAT3 also has a simulator, which can be used to visualize its checking processes. Selecting different simulation engines, PAT3 can traverse the system's state-space using different heuristics.

3 Solving Sudoku in a cP system

Our strategy of solving Sudoku is to generate all possible solutions (matrices), eliminate invalid ones, and filter them by comparing them to the input puzzle. For an $m \times m$ Sudoku, the cP system will first generate all valid m-size row candidates, where each candidate is a permutation of $[1..m]$. Then the system will use these row candidates to build templates of $m \times m$ matrices. After getting all the matrix templates, the system will filter them by columns and blocks. After that, it will contain all the valid $m \times m$ Sudoku solutions. Then it can match these matrix templates to a particular instance, and find its

solutions.

The cP system starts at state s_1 with terms $p()$, $t()$, $s(S)$, $a(1)$, $a(2)$,..., $a(m)$, $n(n)$, $m(m)$, and $l(1)$. The term $p(_)$ is used to build and store the row candidates, $t(_)$ is used to store matrix templates, and $s(S)$ is the cP encoding of a Sudoku puzzle instance. Terms $a(1)$, $a(2)$ ..., $a(m)$ store the numbers from 1 to m, which can be used to fill the blank cells of the puzzle. $n(n)$ stores the block size and $m(m)$ stores the problem size of the puzzle, where $m = n^2$. The system uses $l(_)$ as a counter.

A simple Sudoku example ($m = 4$) is shown in Fig. 2. In its cP representation, we use two terms $m(4)$ and $n(2)$ to represent its problem size and block size. Four numbers $a(1)$, $a(2)$, $a(3)$ and $a(4)$ can be used to fill the puzzle. The puzzle instance is encoded as $s(r(1)(c(3)(2), c(4)(4)),\ r(2)(c(1)(2), c(2)(4), c(4)(3)),\ r(3)(c(2)(1)),$ $r(4)(c(3)(3)))$. In the encoding, the term s stores all the existing numbers of the puzzle, where sub-cell names r and c refer to "row" and "column" respectively. The sub-cell $r(2)(c(1)(2), c(2)(4), c(4)(3))$ can be interpreted as "the value in row 2 column 1 is 2, in row 2 column 2 is 4, and in row 2 column 4 is 3".

		2	4
2	4		3
	1		
		3	

Figure 2. A Sudoku puzzle, $m = 4$

3.1 Generating row candidates

To build valid solutions, the cP system first generates all the row candidates. Each row candidate contains all the numbers from 1 to m, and each number only appears once. A ruleset with four rules can be used to generate row candidates in a column by column manner (Fig. 3).

$s_1\ l(M1)$	$\rightarrow_1$	s_2	$l(1) \mid m(M)$	(1)
s_1	$\rightarrow_+$	s_1	$p(X, c(L)(V)) \mid l(L), a(V), p(X), (c(_)(V) \nsubseteq X)$	(2)
$s_1\ p(_)$	$\rightarrow_+$	s_1		(3)
$s_1\ l(L)$	$\rightarrow_1$	s_1	$l(L1)$	(4)

Figure 3. Ruleset (1): generating row candidates

Rule (1) uses a counter $l(_)$ to track the progress of generating row candidates. When the value of $l(M1)$ is greater than the puzzle size $m(M)$, it means all the row candidates have been successfully generated. Then the cP system resets the counter to $l(1)$, changes its state to s_2 and moves to the next ruleset.

Rule (2) works in the max-parallel model, so all the compatible unified rules will be applied (all solution will be generated). When rule (2) is applied, it adds a number V at column L to each row candidate $p(X)$. The relation $c(_)(V) \nsubseteq X$ guarantees the number V has not been used in the same row candidate. At the beginning of the computation, $p()$ was empty. By applying rule (2) once, the system creates m different terms, which are $p(c(1)(1))$, $p(c(1)(2))$..., $p(c(1)(m))$. By applying it again, the system will generate $m \times (m-1)$ terms including $p(c(1)(1), c(2)(2))$, $p(c(1)(1), c(2)(3))$..., $p(c(1)(m), c(2)(m-1))$. After applying it m times, the cP system will generate all the $m!$ row candidates.

Rule (3) is another max-parallel rule, which cleans the out-of-date $p(_)$ terms in the system. As mentioned, rule (2), (3) and (4) commit to the same target state, so if applicable, they will be applied in one step. Thus $p(_)$ terms generated by rule (2) will not be immediately consumed by rule (3) in the same step.

Rule (4) increases the counter $l(_)$ by 1 in every step, by consuming the existing counter $l(L)$, and producing a new counter $l(L1)$.

To generate all the row candidates for a $m \times m$ Sudoku, the cP system needs to apply the ruleset $m+1$ times, and the state of the system will be changed from s_1 to s_2.

3.2 Generating matrix templates

The ruleset to build matrix templates is shown in Fig. 4. Using the row candidates generated by ruleset (1), the cP system builds matrix templates row by row.

$s_2\ l(M1)$	$\rightarrow_1$	s_3	$l(1) \mid m(M)$	(5)
s_2	$\rightarrow_+$	s_2	$t(X, r(L)(P)) \mid l(L), p(P), t(X), (r(_)(P) \notin X)$	(6)
$s_2\ t(_)$	$\rightarrow_+$	s_2		(7)
$s_2\ l(L)$	$\rightarrow_1$	s_2	$l(L1)$	(8)

Figure 4. Ruleset (2): generating matrix templates

The cP system uses the counter $l(_)$ to track the working row. Once all the m rows of matrix templates are filled with row candidates – when $l(M1)$ is greater than $m(M)$ – rule (5) is applicable. It changes the system state to s_3, and resets the counter to $l(1)$.

Rule (6) generates matrix templates row by row. In every step, the system adds exactly one row candidate $p(P)$ at row L to each matrix template $t(X)$. After m steps, the system will finish generating all $m!!/(m! - m)!$ matrix templates. Rule (7) cleans out-of-date $t(_)$ terms, and rule (8) increments the counter $l(_)$ by 1 in each step.

Ruleset (2) takes $m + 1$ steps in total. The matrix templates generated by ruleset (2) do not have any number conflicts in each row, since every row candidate is a permutation of [1..m]; but they may have number conflicts in columns and blocks (Fig. 5).

1	3	2	4
2	4	1	3
2	1	3	4
4	2	3	1

Figure 5. Number conflicts in a matrix template

3.3 Filtering matrix templates by columns

To delete the matrix templates with number conflicts in columns, the cP system only needs one max-parallel rule (Fig. 6). The rule works as a filter, which is applied to all the matrix templates simultaneously. In a matrix template $t(_)$, if there are two cells in the same column – row A column C and row B column C – share the same value V, the template will be consumed (deleted). Ruleset (3) only needs 1 step to run. After applying it, all the matrix templates that remain in the cP system do not have any number conflicts in rows and columns.

s_3	$t(r(A)(c(C)(V)_),\ r(B)(c(C)(V)_),\ _)$	$\rightarrow_+$	s_3	(9)

Figure 6. Ruleset (3): filtering matrix templates by columns

3.4 Filtering matrix templates by blocks

To check if matrix templates have number conflicts in blocks, we need to create some supporting terms to indicate the relationship among rows, columns and blocks. For example, when $m = 9$, we can build terms $b(1)(1)$, $b(2)(1)$, $b(3)(1)$, $b(4)(2)$, $b(5)(2)$, $b(6)(2)$, $b(7)(3)$, $b(8)(3)$ and $b(9)(3)$ in the cP system. To check if two Sudoku cells are in the same block, we only need to compare their rows and columns with the supporting terms. Suppose we want to check if two cells – row 4 column 3 and row 6 column 1 – are in the same block (Fig. 7). First, we check terms $b(4)(A)$ and $b(6)(B)$ in the supporting terms, we can find $A = 2$ and $B = 2$. Then we check $b(3)(X)$ and $b(1)(Y)$, find $X = 1$ and $Y = 1$. If $A = B$ and $X = Y$, the two cells are in the same block; otherwise they are not. In this example, row 4 column 3 and row 6 column 1 are in the same block.

The ruleset we use to build the supporting terms is shown in Fig. 8. Rule (10) creates two terms $v(1)$ and $k(N)$, and changes the state to s_4. Term $v(V)$ holds a value to fill in current supporting term $b(_)(_)$, and $k(K)$ tracks the boundary of blocks. Rule (11) monitors the progress of building supporting terms. When $l(M1)$ is greater than $m(M)$, the

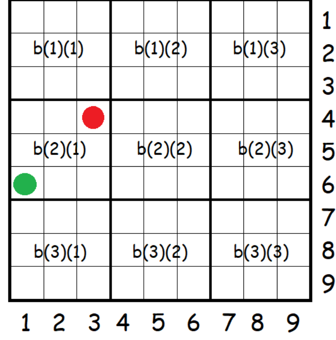

Figure 7. Checking if two cells are in the same block

system has finished creating supporting terms, it changes the state to s_5. Rule (12) creates a supporting term $b(L)(V)$ based on the counter $l(L)$ and value $v(V)$. Rule (13) updates terms $v(_)$ and $k(_)$ after the counter $l(_)$ moved to the next block. Rule (14) increases the counter $l(L)$ by 1 in each step.

s_3	$\rightarrow_1$	s_4	$v(1),\ k(N) \mid n(N)$	(10)
$s_4\ l(M1)$	$\rightarrow_1$	s_5	$\mid m(M)$	(11)
s_4	$\rightarrow_1$	s_4	$b(L)(V) \mid l(L),\ v(V)$	(12)
$s_4\ k(K),\ v(V)$	$\rightarrow_1$	s_4	$k(KN),\ v(V1) \mid n(N),\ l(K)$	(13)
$s_4\ l(L)$	$\rightarrow_1$	s_4	$l(L1)$	(14)

Figure 8. Ruleset (4a): creating block checking supporting terms

After having the supporting terms in the system, rule (15) can filter the matrix templates by blocks (Fig. 9). If the system detects a matrix template $t(_)$ has two cells (row X column A and row Y column B) in the same block that share the same value V, it will consume the template. Rule (15) runs in the max-parallel model, which can filter all the matrix templates in 1 step.

Rulesets (4a) and (4b) take $m + 3$ steps in total. After applying them, all the matrix templates that remain in the cP system are valid Sudoku solutions; and all the valid solutions are contained in the

$$s_5 \quad t(r(X)(c(A)(V)_),\ r(Y)(c(B)(V)_),\ _) \quad \rightarrow_+ \quad s_5 \quad \mid b(X)(W), b(Y)(W), b(A)(C), b(B)(C) \qquad (15)$$

Figure 9. Ruleset (4b): filtering matrix templates by blocks

cP system!

3.5 Matching matrix templates to a Sudoku instance

One max-parallel rule can be used to match the matrix templates to a Sudoku instance (Fig. 10). Rule (16) compares all matrix templates $t(_)$ to the instance $s(S)$. If the system finds any conflicts between a matrix template $t(_)$ and $s(S)$, it deletes the template. Rule (16) takes 1 step. After applying it, the $t(T)$ terms in the system are solutions to the instance. A Sudoku instance may have multiple valid solutions, the cP system is guaranteed to find all of them at the same step.

$$s_5 \quad t(r(R)(c(C)(V)_)_) \quad \rightarrow_+ \quad s_5 \quad \mid s(r(R)(c(C)(U)_)_),\ U \neq V \qquad (16)$$

Figure 10. Ruleset (5): matching matrix templates to a Sudoku instance

The cP solution consists of 16 rules, which can solve any $m \times m$ Sudoku instances in $3m + 7$ steps. Considering the size of the input of a Sudoku puzzle is m^2, the complexity of the solution is sublinear (square root time).

4 A worked example

In this section we use the Sudoku puzzle ($m = 4$, $n = 2$) shown in Fig. 2 to illustrate how our cP system works. To represent the puzzle, we create 10 terms in the cP system (Table 1). The initial state of the system is s_1.

Table 1. Initial terms in the cP system

$m(4),\ n(2),\ a(1),\ a(2),\ a(3),\ a(4),\ l(1),\ p(),\ t(),$ $s(r(1)(c(3)(2), c(4)(4)),\ r(2)(c(1)(2), c(2)(4), c(4)(3)),\ r(3)(c(2)(1)),\ r(4)(c(3)(3))).$

The cP system uses ruleset (1) to generate all valid row candidates. A hand simulation can be found in Table 2. Since there are many terms in the system, the table only shows terms directly related to the ruleset. In each step, the system adds exactly one number into each $p(_)$ term. After $m + 1 = 5$ steps, the cP system will successfully generate all the $m! = 24$ row candidates, which are the permutations of [1..4]. Then the system will reset the counter to $l(1)$, and change its state to s_2.

Ruleset (2) will be applied when the system is at state s_2. It uses row candidates to build matrix templates row by row. After $m+1 = 5$ steps, the system will generate all $m!!/(m! - m)! = 255024$ matrix templates, and change its state to s_3.

To filter the matrix templates by columns, the cP system applies ruleset (3). If it finds one value that appears twice in a column of a template, it consumes that template. Ruleset (3) takes one max-parallel step to apply to all matrix templates in the system, and does not change the system state. After applying it, all the matrix templates with number conflicts in their columns will be deleted from the cP system.

The cP system uses rulesets (4a) and (4b) to check and delete matrix templates with number conflicts in blocks. The hand simulation of building supporting terms (ruleset (4a)) can be found in Table 4. System terms which are related to ruleset (4a) are shown in the table.

By applying rule (10), the system will generate terms $v(1)$ and $k(2)$. Rule (11) compares terms $l(L)$ and $m(4)$, it is only applied when L is greater than 4. Rule (12) keeps building $b(_)$ terms in every step. Rule (13) updates $k(_)$ and $v(_)$ values when the counter $l(_)$ moves to a new block. Rule (14) increases $l(_)$ by 1 in each step. Because rules (12), (13) and (14) are committing to the same state s_4, the cP system will apply them in the same step if possible.

Terms $b(1)(1)$, $b(2)(1)$, $b(3)(2)$, $b(4)(2)$ show the relationship among

Table 2. Generating row candidates

Step	State	Available terms	Generated terms	Rule to apply
0	s_1	$m(4)$, $l(1)$, $p()$.		(2)
1	s_1	$m(4)$, $l(1)$.	$p(c(1)(1))$, $p(c(1)(2))$, $p(c(1)(3))$, $p(c(1)(4))$.	(4)
1	s_1	$m(4)$.	$p(c(1)(1))$, $p(c(1)(2))$, $p(c(1)(3))$, $p(c(1)(4))$, $l(2)$.	(2)
2	s_1	$m(4)$, $p(c(1)(1))$, $p(c(1)(2))$, $p(c(1)(3))$, $p(c(1)(4))$, $l(2)$.	$p(c(1)(1), c(2)(2))$, $p(c(1)(1), c(2)(3))$,..., $p(c(1)(4), c(2)(3))$.	(3)
2	s_1	$m(4)$, $l(2)$.	$p(c(1)(1), c(2)(2))$, $p(c(1)(1), c(2)(3))$,..., $p(c(1)(4), c(2)(3))$.	(4)
2	s_1	$m(4)$.	$p(c(1)(1), c(2)(2))$, $p(c(1)(1), c(2)(3))$,..., $p(c(1)(4), c(2)(3))$, $l(3)$.	(2)
3	s_1	$m(4)$, $p(c(1)(1), c(2)(2))$, $p(c(1)(1), c(2)(3))$,..., $p(c(1)(4), c(2)(3))$, $l(3)$.	$p(c(1)(1), c(2)(2), c(3)(3))$, $p(c(1)(1), c(2)(2), c(3)(4))$,..., $p(c(1)(4), c(2)(3), c(3)(2))$.	(3)
3	s_1	$m(4)$, $l(3)$.	$p(c(1)(1), c(2)(2), c(3)(3))$, $p(c(1)(1), c(2)(2), c(3)(4))$,..., $p(c(1)(4), c(2)(3), c(3)(2))$.	(4)
3	s_1	$m(4)$.	$p(c(1)(1), c(2)(2), c(3)(3))$, $p(c(1)(1), c(2)(2), c(3)(4))$,..., $p(c(1)(4), c(2)(3), c(3)(2))$, $l(4)$.	(2)
4	s_1	$m(4)$, $p(c(1)(1), c(2)(2), c(3)(3))$, $p(c(1)(1), c(2)(2), c(3)(4))$,..., $p(c(1)(4), c(2)(3), c(3)(2))$, $l(4)$.	$p(c(1)(1), c(2)(2), c(3)(3), c(4)(4))$, $p(c(1)(1), c(2)(2), c(3)(4)), c(4)(3))$,..., $p(c(1)(4), c(2)(3), c(3)(2), c(4)(1))$.	(3)
4	s_1	$m(4)$, $l(4)$.	$p(c(1)(1), c(2)(2), c(3)(3), c(4)(4))$, $p(c(1)(1), c(2)(2), c(3)(4)), c(4)(3))$,..., $p(c(1)(4), c(2)(3), c(3)(2), c(4)(1))$.	(4)
4	s_1	$m(4)$.	$p(c(1)(1), c(2)(2), c(3)(3), c(4)(4))$, $p(c(1)(1), c(2)(2), c(3)(4)), c(4)(3))$,..., $p(c(1)(4), c(2)(3), c(3)(2), c(4)(1))$, $l(5)$.	(1)
5	s_2	$m(4)$, $p(c(1)(1), c(2)(2), c(3)(3), c(4)(4))$, $p(c(1)(1), c(2)(2), c(3)(4)), c(4)(3))$,..., $p(c(1)(4), c(2)(3), c(3)(2), c(4)(1))$.	$l(1)$.	

rows, columns and blocks. To check if two cells row 1 column 2 and row 2 column 3 are in the same block, we need to compare their rows

Table 3. A matrix template with number conflicts (cP representation)

$t(r(1)(c(1)(1), c(2)(3), c(3)(2), c(4)(4)),$
$r(2)(c(1)(2), c(2)(4), c(3)(1), c(4)(3)),$
$r(3)(c(1)(2), c(2)(1), c(3)(4), c(4)(3)),$
$r(4)(c(1)(4), c(2)(1), c(3)(3), c(4)(2))).$

Table 4. Building supporting terms (for block check)

Step	State	Available terms	Generated terms	Rule to apply
0	s_3	$n(2)$, $m(4)$, $l(1)$.		(10)
1	s_4	$n(2)$, $m(4)$, $l(1)$.	$v(1)$, $k(2)$.	(12)
2	s_4	$n(2)$, $m(4)$, $l(1)$, $v(1)$, $k(2)$.	$b(1)(1)$.	(14)
2	s_4	$n(2)$, $m(4)$, $v(1)$, $k(2)$.	$b(1)(1)$, $l(2)$.	(12)
3	s_4	$n(2)$, $m(4)$, $v(1)$, $k(2)$, $b(1)(1)$, $l(2)$.	$b(2)(1)$.	(13)
3	s_4	$n(2)$, $m(4)$, $b(1)(1)$, $l(2)$.	$b(2)(1)$, $v(2)$, $k(4)$.	(14)
3	s_4	$n(2)$, $m(4)$, $b(1)(1)$.	$b(2)(1)$, $v(2)$, $k(4)$, $l(3)$.	(12)
4	s_4	$n(2)$, $m(4)$, $b(1)(1)$, $b(2)(1)$, $v(2)$, $k(4)$, $l(3)$.	$b(3)(2)$.	(14)
4	s_4	$n(2)$, $m(4)$, $b(1)(1)$, $b(2)(1)$, $v(2)$, $k(4)$.	$b(3)(2)$, $l(4)$	(12)
5	s_4	$n(2)$, $m(4)$, $b(1)(1)$, $b(2)(1)$, $v(2)$, $k(4)$, $b(3)(2)$, $l(4)$.	$b(4)(2)$.	(13)
5	s_4	$n(2)$, $m(4)$, $b(1)(1)$, $b(2)(1)$, $b(3)(2)$, $l(4)$.	$b(4)(2)$, $v(3)$, $k(6)$.	(14)
5	s_4	$n(2)$, $m(4)$, $b(1)(1)$, $b(2)(1)$, $b(3)(2)$.	$b(4)(2)$, $v(3)$, $k(6)$, $l(5)$.	(11)
6	s_5	$n(2)$, $m(4)$, $b(1)(1)$, $b(2)(1)$, $b(3)(2)$, $b(4)(2)$, $v(3)$, $k(6)$.		

and columns separately. To check their rows, we need to find terms $b(1)(A)$ and $b(2)(B)$ in the cP system – which are $b(1)(1)$ and $b(2)(1)$, thus $A = B = 1$. To check their columns, we search for $b(2)(X)$ and $b(3)(Y)$ in the system, and we will get $b(2)(1)$, $b(3)(2)$, thus we have $X = 1$, $Y = 2$, where $X \neq Y$. Then we know that row 1 column 2 and row 2 column 3 are not in the same block.

Ruleset (4a) takes $m + 2 = 6$ steps. After building the support-

ing terms, the system will take one max-parallel step to delete matrix templates with number conflicts in blocks (ruleset (4b)), then all the matrix templates left in the cP system are valid solutions to different Sudoku puzzles.

By applying ruleset (5) – which also takes one max-parallel step – the system can find the solutions to the given Sudoku instance (Table 5), by deleting other matrix templates in it. The entire solution takes $3m + 7 = 19$ steps.

Table 5. The solution of the example puzzle in Fig. 2

$t(r(1)(c(1)(1), c(2)(3), c(3)(2), c(4)(4)),$
$r(2)(c(1)(2), c(2)(4), c(3)(1), c(4)(3)),$
$r(3)(c(1)(3), c(2)(1), c(3)(4), c(4)(2)),$
$r(4)(c(1)(4), c(2)(2), c(3)(3), c(4)(1))).$

5 Verifying the cP Sudoku solution

We verified the two core rulesets (1 and 2) of our solution in PAT3. To model our cP system in CSP#, we mainly followed the transformation guidelines proposed in our previous work [23]. A translation example from cP syntax of ruleset (1) to CSP# is shown in Figure 11.

Our cP system's properties including deadlock-freeness (safety, weak-liveness), terminating (safety, liveness), divergence-freeness (safety) and non-deterministic (fairness) were formally verified. The CSP# code of this study can be found in *https://github.com/YezhouLiu/cP-Sudoku*.

Table 6 shows the model checking result. The cP system is deadlockfree, divergencefree, terminating and non-deterministic. In model checking, a *deadlock* state is a state with no further move (except expected final states). The "state" here refers to the state in the Kripke structure in the model checker, which is different from states in cP systems. Deadlockfree can be written as $A\Box(E \bigcirc (true))$ in CTL, which means "a non-deadlock state must have at least one successor". The symbol $\Box$ refers to *globally* and $\bigcirc$ is the *next operator*. In cP systems,

```
#define M 9; //problem size
var a = [1,2,3,4,5,6,7,8,9];     //term a
var c[M];        //index: L-1, value: V
var l = 1; //term l
var state = 1;
var promoter_valid = true;

R1() = rule1{
        if(state == 1 && l == M + 1){
                l = 1;
                state = 2;
        }
} -> if (state == 2){Skip} else {R2()};

R2() = []i:{0..(M-1)}@ rule2{
        if (state == 1){
                promoter_valid = true;
                var j = 0;
                while (j < M){
                        if (c[j] == a[i]){
                                promoter_valid = false;
                        }
                        j++;
                }
                if(promoter_valid){c[l-1] = a[i];}
                state = 1;
        }
} -> if (promoter_valid){R3()} else{Skip};

R3() = rule3{
        if (state == 1){state = 1;}
} -> R4();

R4() = rule4{
        if(state == 1){
                l = l + 1;
                state = 1;
        }
} -> R1();
```

Figure 11. The CSP# representation of ruleset (1)

a deadlock indicates that the system terminates somewhere unexpectedly. For example, if a rule's $r - state$ is miswritten as another state, which is unused anywhere else, after applying the rule, the system may

encounter a deadlock. The deadlock check can tell us if the cP system will get stuck.

Divergence in PAT3 means a process may perform internal transitions forever, which is often undesirable. The divergence check can be used to detect badly designed rules in cP systems (e.g. unnecessary self-looping rules). *Terminating* means the system will eventually halt, which can often be written as a reachability property. For example, a termination of ruleset (1) can be defined as: $A\Diamond(at_{s_2})$, where $\Diamond$ means "eventually". When a cP system applies ruleset (1), after generating all the row candidates, the system state will be changed to s_2. Then ruleset (1) terminates, because none of its rules are applicable at s_2. Performing the terminating check, we can find out if a cP system will run forever, or eventually halt at an expected state.

As a fairness property, *deterministic* means for a state there is no more than one out-going transition. cP systems are often non-deterministic because of unification. When designing a non-deterministic cP system, we often need to make it *confluent* – i.e. all its evolutions need to yield the same result. For example, when building a row candidate, the cP system will randomly choose a number to fill its working column. All the possible numbers have equal chance to be selected, none of the numbers will be neglected by the system. The system will finally generate $m!$ row candidates which represent all the permutations of $[1..m]$.

Table 6. Model checking result of the cP Sudoku rulesets

Ruleset	**Problem Size**	**Deadlock-free**	**Divergence-free**	**Terminating**	**Deterministic**
(1)	4	True	True	True	False
(1)	9	True	True	True	False
(2)	4	True	True	True	False

A major limitation of applying model checking to our cP system is state explosion. We verified the system's properties with $m = 4$ and $m = 9$. When the problem size $m = 9$, PAT3 encounters a memory explosion issue when verifying ruleset (2). To model check ruleset (1), PAT3 generated $9! = 362880$ row candidates and successfully checked

its state-space. To verify ruleset (2), PAT3 needs to simulate the generation of the $9!!/(9!-9)!$ matrix templates, which is impossible in practice. Even though PAT3 implements abstraction algorithms and can generate its state-space on the fly (without keeping the entire state-space in memory), it still cannot check that many states.

Because of the combinatorial explosion and limited languages features supported by existing general purpose model checkers, max-parallel filtering cP rulesets including ruleset (3), (4a), (4b), and (5) are not suitable to be verified via model checking. There is no straight forward way to manually release terms' memory in model checkers such as PAT3, so it is hard to emulate the term consumption in cP systems.

6 Related work

The first P system study on solving the Sudoku problem was published in 2010 [19]. To solve Sudoku puzzles, the authors used a family of P systems with enzymatic rules, dissolution rules, and send-out rules. In the P solution, 13 rulesets of $O(n^6)$ rules were used, where n is the block size of the puzzle. In most P system variants, to solve computationally hard problems, a number of P rules related to the problem size are needed. Instead of solving Sudoku in a brute-force manner, it used a human-style strategy. It is not guaranteed that the P system can solve all the Sudoku instances, but the system will indicate failure if it cannot solve an instance.

Later, the P system in [19] was extended by other researchers [20]. Additional algorithms including pruning were added to the solution. The authors also introduced a brute-force algorithm to the system. In this extended version of the P system, if no valid Sudoku solution can be found by the human-style strategy, the system will try the brute-force algorithm. $O(n^6)$ rules were used in the extended version of P system with the brute-force approach. The authors tested their solution in the P-Lingua simulator [26], but only for small-scale problems ($n = 2$).

Another P study used particle swarm optimization incorporated with P systems to solve Sudoku puzzles [21]. No P system rule was predefined in the system. The system generated the evolution rules

and communication rules by using particle swarm optimization. The authors also proposed another P system solution to solve Sudoku [27], where some evolution and communication rules were used. The ruleset of the P solution was not specified in the paper, too. These solutions cannot guarantee to solve all the Sudoku puzzles. The two algorithms designed by the authors only borrowed some shallow ideas from the cell-like structure of P systems, which did not make use of the theoretical unlimited computational power and memory of membrane computing models.

7 Conclusion

Although membrane computing models are suitable for solving computationally hard problems, to model practical problems in P system models is often non-trivial. In this study, we solved one of the most famous NP-complete puzzles – Sudoku – in cP systems. Our cP system only contains 16 rules, which can solve all $m \times m$ Sudoku puzzles in $3m + 7$ steps by using relations as special promoters. Considering the input size of a Sudoku puzzle is m^2, the cP solution is sublinear. To the best of our knowledge, our solution is the most efficient P system solution to the Sudoku problem in time complexity. We formally verified the core rulesets of our solution using the PAT3 model checker, and checked their safety, liveness and fairness properties.

Similar to other P system solutions, practical implementations of our cP solution will encounter a memory explosion issue. The search for finding efficient software or hardware implementations of P systems is one of the most important challenges in membrane computing.

In addition to the brute force algorithm, future work may include the design of practically useful cP solutions, based on practical Sudoku solving strategies. Although it is not guaranteed that the practical strategies can solve all Sudoku instances, they usually can find solutions for real-life Sudoku puzzles quickly.

To solve the existing issues on simulating and verifying cP systems, we also plan to design and implement a cP model checker, which can effectively simulate cP systems (including our cP Sudoku solution) and

automatically check their properties.

References

[1] D. Berend, "On the number of Sudoku squares," *Discrete Mathematics*, vol. 341, no. 11, pp. 3241–3248, 2018.

[2] D. E. Knuth, "Dancing links," *arXiv preprint cs/0011047*, 2000.

[3] R. Lewis, "Metaheuristics can solve Sudoku puzzles," *Journal of Heuristics*, vol. 13, no. 4, pp. 387–401, 2007.

[4] A. Moraglio, J. Togelius, and S. Lucas, "Product geometric crossover for the Sudoku puzzle," in *2006 IEEE International Conference on Evolutionary Computation*. IEEE, 2006, pp. 470–476.

[5] A. Moraglio and J. Togelius, "Geometric particle swarm optimization for the Sudoku puzzle," in *GECCO*, vol. 7, 2007, pp. 118–125.

[6] T. Mantere and J. Koljonen, "Solving, rating and generating Sudoku puzzles with GA," in *2007 IEEE Congress on Evolutionary Computation*. IEEE, 2007, pp. 1382–1389.

[7] Z. W. Geem, "Harmony search algorithm for solving Sudoku," in *International Conference on Knowledge-Based and Intelligent Information and Engineering Systems*. Springer, 2007, pp. 371–378.

[8] I. Lynce and J. Ouaknine, "Sudoku as a SAT problem." in *ISAIM*, 2006.

[9] H. Simonis, "Sudoku as a Constraint Problem," *Modelling and Reformulating Constraint Satisfaction Problems*, p. 13, 2005.

[10] T. K. Moon, J. H. Gunther, and J. J. Kupin, "Sinkhorn solves Sudoku," *IEEE Transactions on Information Theory*, vol. 55, no. 4, pp. 1741–1746, 2009.

[11] G. Santos-García and M. Palomino, "Solving Sudoku puzzles with rewriting rules," *Electronic Notes in Theoretical Computer Science*, vol. 176, no. 4, pp. 79–93, 2007.

[12] G. Păun, "Computing with membranes," *Journal of Computer and System Sciences*, vol. 61, no. 1, pp. 108–143, 2000.

[13] A. Păun, "On P systems with active membranes," in *Unconventional Models of Computation, UMC'2K*. Springer, 2001, pp. 187–201.

[14] C. Martín-Vide, G. Păun, J. Pazos, and A. Rodríguez-Patón, "Tissue P systems," *Theoretical Computer Science*, vol. 296, no. 2, pp. 295–326, 2003.

[15] M. Ionescu, G. Păun, and T. Yokomori, "Spiking neural P systems," *Fundamenta informaticae*, vol. 71, no. 2, 3, pp. 279–308, 2006.

[16] M. Gheorgue, F. Ipate, C. Dragomir, L. Mierla, L. Valencia Cabrera, M. García Quismondo, and M. d. J. Pérez Jiménez, "Kernel P systems-version 1," *Proceedings of the Eleventh Brainstorming Week on Membrane Computing, 97-124. Sevilla, ETS de Ingeniería Informática, 4-8 de Febrero, 2013,*, 2013.

[17] R. Nicolescu, F. Ipate, and H. Wu, "Programming P systems with complex objects," in *International Conference on Membrane Computing*. Springer, 2013, pp. 280–300.

[18] R. Nicolescu and A. Henderson, "An introduction to cP systems," in *Enjoying natural computing*. Springer, 2018, pp. 204–227.

[19] D. Díaz-Pernil, C. M. Fernández-Mírquez, M. García-Quismondo, M. A. Gutiérrez-Naranjo, and M. A. Martínez-del Amor, "Solving Sudoku with membrane computing," in *2010 IEEE Fifth International Conference on Bio-inspired Computing: Theories and Applications (BIC-TA)*. IEEE, 2010, pp. 610–615.

[20] D. Deodhare, S. Sonone, and A. Gupta, "A generic membrane computing-based Sudoku solver," in *2014 International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT).* IEEE, 2014, pp. 89–99.

[21] G. Singh and K. Deep, "A new membrane algorithm using the rules of Particle Swarm Optimization incorporated within the framework of cell-like P-systems to solve Sudoku," *Applied Soft Computing*, vol. 45, pp. 27–39, 2016.

[22] Y. Liu, J. Sun, and J. S. Dong, "Pat 3: An extensible architecture for building multi-domain model checkers," in *2011 IEEE 22nd International Symposium on Software Reliability Engineering.* IEEE, 2011, pp. 190–199.

[23] Y. Liu, R. Nicolescu, and J. Sun, "Formal verification of cP systems using PAT3 and ProB," *Journal of Membrane Computing*, pp. 1–15, 2020.

[24] J. Cooper and R. Nicolescu, "The Hamiltonian cycle and travelling salesman problems in cP systems," *Fundamenta Informaticae*, vol. 164, no. 2-3, pp. 157–180, 2019.

[25] A. Henderson, R. Nicolescu, and M. J. Dinneen, "Solving a PSPACE-complete problem with cP systems," *Journal of Membrane Computing*, pp. 1–12, 2020.

[26] I. Pérez-Hurtado, D. Orellana-Martín, G. Zhang, and M. J. Pérez-Jiménez, "P-lingua in two steps: flexibility and efficiency," *Journal of Membrane Computing*, vol. 1, no. 2, pp. 93–102, 2019.

[27] G. Singh and K. Deep, "Cell-like P-systems using deterministic update rules to solve Sudoku," *International Journal of System Assurance Engineering and Management*, vol. 8, no. 2, pp. 857–866, 2017.

Yezhou Liu, Radu Nicolescu,
Jing Sun, Alec Henderson

Received October 22, 2020
Revised December 22, 2020

Yezhou Liu
Institution: The University of Auckland
Address: SCIENCE CENTRE 303S - Bldg 303S, Level 5, Room 596, 38 PRINCES ST, AUCKLAND, New Zealand 1010
E–mail: yliu442@aucklanduni.ac.nz

Radu Nicolescu
Institution: The University of Auckland
Address: SCIENCE CENTRE 303S - Bldg 303S, Level 5, Room 587, 38 PRINCES ST, AUCKLAND, New Zealand 1010
E–mail: r.nicolescu@auckland.ac.nz

Jing Sun
Institution: The University of Auckland
Address: SCIENCE CENTRE 303 - Bldg 303, Level 5, Room 522, 38 PRINCES ST, AUCKLAND, New Zealand 1010
E–mail: jing.sun@auckland.ac.nz

Alec Henderson
Institution: The University of Auckland
Address: SCIENCE CENTRE 303S - Bldg 303S, Level 5, Room 596, 38 PRINCES ST, AUCKLAND, New Zealand 1010
E–mail: ahen386@aucklanduni.ac.nz