



Solving a PSPACE-complete problem with cP systems

Alec Henderson¹ · Radu Nicolescu¹ · Michael J. Dinneen¹

Received: 24 June 2020 / Accepted: 12 October 2020 / Published online: 24 November 2020
© Springer Nature Singapore Pte Ltd. 2020

Abstract

There have been a few NP-hard problems solved using cP systems including the travelling salesman problem. However, these problems are typically in NP rather than higher in the polynomial time hierarchy. In this paper, we solve QSAT (also known as TQBF), which is a well-known PSPACE-complete problem. Compared to other extant confluent P systems solutions, our deterministic cP solution only uses a small constant number of custom alphabet symbols (19), a small constant number of rules (10) and a small constant upper limit of membrane nesting depth (6), independent of the problem size.

Keywords Membrane computing · cP systems · Computational complexity

1 Introduction

There are many computational complexity classes from log space to exponential and beyond. Many of these classes are not known to be equal or not. One of the millennial prize problems is dedicated to knowing whether P equals NP. The class NP has many complete problems such as the maximum independence set, the travelling salesman, and satisfiability problems. NP is one of the lowest levels of the polynomial-time hierarchy which characterises many different classes within PSPACE. One of the most famous PSPACE-complete problems is the Quantified Boolean Satisfiability (QSAT) problem, also known as the True Quantified Boolean Formula (TQBF) problem.

P systems proposed in [1] are a parallel and distributed model of computation based on a membrane structure. Following this work, many different variants of P systems were proposed including P systems with active membranes [2], spiking neural P systems [3], tissue P systems [4] and P systems with compound terms (cP systems). A large number of hard problems have been shown to be able to be solved in polynomial time with P systems. For example, in [2], it was shown how P systems with active membranes are able to solve NP complete problems in linear time.

cP systems are a variant of P systems that use high-level rewriting rules, based on one-way first-order syntactic unification (which is similar to pattern matching in functional programming). cP systems usually have smaller alphabets and rulesets than other P systems; however, the rules are more complex. cP systems have been used: to solve NP-hard problems such as the travelling salesman problem [5], to model distributed problems [6], and with recent research focused on verification [7]. In this paper, we demonstrate that cP systems can not only solve NP-hard problems in linear time, but also PSPACE-complete problems in polynomial time, with QSAT (i.e. TQBF) being solvable in linear time.

Our solution is—as far as we know—the first using cP systems; it uses ten rules and a constant custom alphabet of size 19. Our solution is deterministic, so we do not compare it here with nonconfluent solutions, such as [8]. (This could be the topic of further investigations.) We note that our solution is not the first confluent solution to PSPACE-complete problems using P systems. Previous solutions exist that follow similar ideas, such as [9–12]. Our solution utilises partial evaluation when generating the possible candidate solutions to the problem, allowing our solution to minimise the number of clauses being used. As shown in Table 1, our solution substantially improves the extant results, on several criteria: alphabet size, number of rules and membrane nesting depth—all small constants, independent of the problem size.

Rule templates are groupings of similar rules, only differing by symbol indices. When counting rule templates

✉ Alec Henderson
ahen386@aucklanduni.ac.nz

¹ School of Computer Science, The University of Auckland, Auckland, New Zealand

Table 1 Comparison of our solution with pre-existing confluent P system solutions, where n is the number of variables and m the number of clauses

Solution (year)	# Rule templates	# Rules	# Custom alphabet symbols	Membrane nesting depth
Linear solution for QSAT (2006) [11]	40	$\mathcal{O}(m)$	$\mathcal{O}(nm)$	$\mathcal{O}(n)$
Uniform solution of QSAT (2007) [12]	33	$\mathcal{O}(mn)$	$\mathcal{O}(nm)$	$\mathcal{O}(n)$
Deterministic solution to QSAT (2010) [9]	20	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Solving QSAT in sublinear nesting depth (2018) [10]	27	$\mathcal{O}(mn)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n / \log n)$
QSAT cP system (2020)	10	10	19 (also 6 states)	6

and rules, we did not consider the numbers of repeated copies placed in different membranes/neurons. If we were to include such occurrence counts, the number of rules would increase drastically, for the other extant solutions. For example, the solution to QSAT in [9] would have $\mathcal{O}(2^{2n})$ rules, if we count the rules in every neuron. cP systems do not have such an exponential blow-up, and all these characteristics are small constants.

In Sect. 2, we discuss the background of this specific problem and how cP systems work. In Sect. 3, we present and discuss our ruleset to solve the QSAT problem.

2 Background

In this section, we cover cP systems and a brief explanation of the QSAT problem. For a more thorough introduction to cP systems, see [13].

2.1 QSAT

A Boolean formula is an expression involving Boolean variables and Boolean operations. For more information on Boolean formulae, see [14]. An NP-complete problem is the Boolean satisfiability problem (SAT), which determines if the variables of a given Boolean formula can be assigned Boolean values that evaluate the formula to true.

A Boolean formula is in conjunctive normal form (CNF) if it is expressed as a conjunction (\wedge) of clauses. A clause is a disjunction (\vee) of literals. A literal is a variable or its negation (here indicated by overbars). For example, the following Boolean formula is in CNF:

$$(x_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2).$$

The SAT formulae assume implicit existential quantifiers on all variables. The existential quantifier (\exists) results are true if one of the possible assignments of the variables allows the formula to be true. Thus, the above formula is interpreted as:

$$\exists x_1 \exists x_2 (x_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2).$$

A quantified Boolean formula is a Boolean formula where variables can be explicitly and independently quantified, with existential or universal quantifiers. The universal quantifier (\forall) results true if every possible assignment of the variables results in the formula being true.

Without loss of generality, we use a restricted version of quantified Boolean formulae that are assumed to be in fully quantified prenex normal form. Prenex normal form (PNF) means that the quantified variables are all factored out before the Boolean formula. Fully quantified means that every variable in the Boolean formula has a quantifier. This leads to the problem TQBF, as presented in [14]:

TQBF

$$= \{ \phi \mid \phi \text{ is a true fully quantified Boolean formula in PNF} \}.$$

As shown in [15], TQBF is a PSPACE-complete problem. Without loss of generality, here we only use Boolean formulae which are also in CNF form, a further restricted version which is still PSPACE-complete. This problem is usually referred to as QSAT, where [9–12] also make the same assumptions.

For example, the following two formulae are fully quantified Boolean formulae in CNF and PNF:

$$\forall x_1 \exists x_2 (x_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2). \tag{1}$$

$$\exists x_1 \forall x_2 (x_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2). \tag{2}$$

Solving QSAT for a given formula ϕ can be done, as shown in [14], with the recursive algorithm (pseudocode) presented here in Table 2, slightly adapted, where we separate ϕ in three components: q —the stack of quantifiers; p —the stack of variables; and f the Boolean expression itself (the unquantified matrix).

The given algorithm systematically explores all possible combinations of variable assignments and evaluates the formula according to the given quantifiers. In the top-down pass, expressions $f[x := 0]$, $f[x := 1]$ indicate substitutions in f of x by 0 (i.e. false), respectively, by 1 (i.e. true). In the bottom-up pass, (\exists) is associated with (\vee), and (\forall) with (\wedge), as straightforward arguments indicate.

Table 2 Recursive algorithm for QSAT: q = quantifiers; p = variables; f = unquantified Boolean expression

```

let rec QSAT  $q p f =$ 
  if  $q = ()$  then
    eval  $f$  // no more quantifiers in prefix, all variables assigned

  else
    let  $y, x = \mathbf{pop} q, \mathbf{pop} p$ 
    let  $v' = \text{QSAT } q p f[x := 0]$ 
    let  $v'' = \text{QSAT } q p f[x := 1]$ 
    let  $v = \mathbf{if } y = \forall \mathbf{ then } v' \wedge v'' \mathbf{ else } v' \vee v''$ 
     $v$ 

// sample calls, for Formulae (1, 2)
let  $v = \text{QSAT } (\forall, \exists) (x_1, x_2) (x_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2)$  // false
let  $v = \text{QSAT } (\exists, \forall) (x_1, x_2) (x_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2)$  // true
    
```

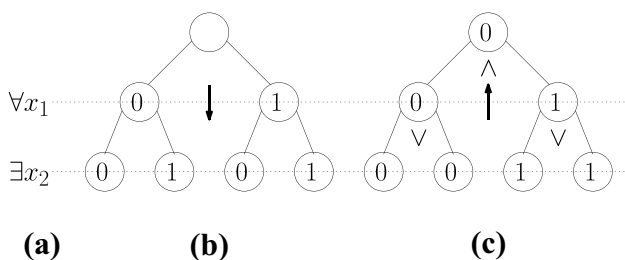


Fig. 1 QSAT tree for Formula (1): $\forall x_1 \exists x_2 (x_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2)$

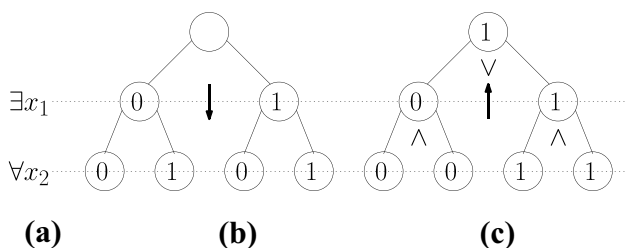


Fig. 2 QSAT tree for Formula (2): $\exists x_1 \forall x_2 (x_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2)$

The candidate solutions of Formulae (1, 2) can be visualised on the trees shown in Figures 1, 2: (a) lists the quantified variables, top-down, one per tree layer; (b) is the top-down construction of the tree, showing variable assignments; and (c) is the bottom-up evaluation of the tree, applying \vee for \exists and \wedge for \forall .

A sequential execution of the recursive solution makes a preorder traversal of the complete tree, using $\mathcal{O}(n)$ space and $\mathcal{O}(2^n)$ runtime steps, where n is the number of quantifiers (or variables) in the prefix, and also the number of tree levels below the root.

Note that Formulae (1, 2) only differ in quantifiers. Thus, Figs. 1, 2 differ only in their quantifiers lists (a) and evaluation results in otherwise isomorphic trees (c), while trees (b) are identical.

The recursion of the algorithm in Table 2 can be unrolled by straightforward techniques. The nonrecursive solution in Table 3 creates each layer of the tree successively, while implicitly discarding the previous layer. Variables in p are processed during the top-down pass, so p is simply successively popped. Quantifiers in q are required during the bottom-up pass, so, during the top-down pass, q is successively reversed into q' .

F is an ordered list of Boolean expressions, corresponding to the nodes of the corresponding layer in the underlying virtual tree. Initially, $F = (f)$, a singleton list containing the formula given by the problem. F changes $2n + 1$ times, by way of the higher-order function map: (i) during the n top-down steps, each expression $f \in F$ is replaced by the substitutions pair $f[x := 0], f[x := 1]$; (ii) at the leaves level, when all variables have been assigned, each f is replaced by its evaluated Boolean value; and (iii) during the n bottom-up steps, each consecutive pair of Boolean values is replaced by either an \wedge or \vee result, depending on the corresponding quantifier, \forall or \exists . (This quantifier was saved in q' during the top-down pass.)

Assuming that enough processing elements are available, a parallel execution of this nonrecursive solution trades space for time, running in $\mathcal{O}(n)$ time and using $\mathcal{O}(2^n)$ space. Our cP solution follows the same process as the nonrecursive solution in Table 3.

2.2 cP systems

P systems, also known as membrane computing, is a generic framework for designing computational models inspired by biology. Similar to many other P systems variants, cP systems (i) assume access to unbounded resources, such as space and computing power; (ii) organise top-level cells into digraph-like structures (Fig. 3); and (iii) evolve by applying formal multiset rewriting rules, with additional messaging primitives between top cells.

Table 3 Nonrecursive pseudocode—layer by layer in both sequential and parallel modes: q = quantifiers; p = variables; F = list of unquantified Boolean expressions

```

let QSAT q p F =
  let q' = ()
  while q ≠ () do // top-down pass
    let x = pop p
    push q' (pop q) // reverse q onto q'
    let F = flatmap F (f → (f[x := 0], f[x := 1]))

    // e.g. F = (0 ∨ 0) ∧ (0 ∨ 0̄), (0 ∨ 1) ∧ (0 ∨ 1̄), (1 ∨ 0) ∧ (1 ∨ 0̄), (1 ∨ 1) ∧ (1 ∨ 1̄)

  let F = map F (f → eval f)

  // e.g. F = 0, 0, 1, 1

  while q' ≠ () do // bottom-up pass
    let y = pop q'
    let P = pairwise F
    // e.g. P = (0, 0), (1, 1)
    let F = map P ((v, v') → if y = ∀ then v' ∧ v'' else v' ∨ v'')
    // e.g. F = 0, 1

  singleton F // returned value

// sample calls, for Formulae (1, 2)
let v = QSAT (∀, ∃) (x1, x2) (x1 ∨ x2) ∧ (x1 ∨ x̄2) // false
let v = QSAT (∃, ∀) (x1, x2) (x1 ∨ x2) ∧ (x1 ∨ x̄2) // true
  
```

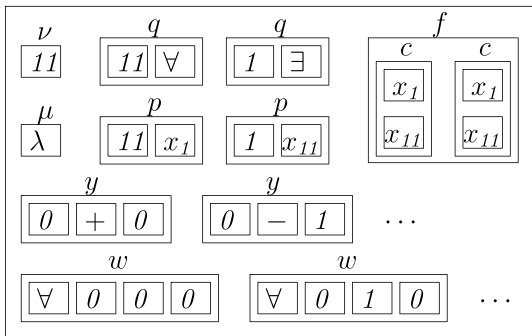


Fig. 3 High-level view of our top-level cell, as initialised for Formula (1) (Cf. Table 8)

cP top cells contain multisets of atoms and labelled sub-cells, which are compound objects similar to ground terms used in logic programming (Prolog). However, unlike Prolog terms, cP terms are strictly multiset based, thus totally

unordered and allowing repetitions. Collectively, cP cells (top cells and sub-cells) correspond to *cells* or membranes used by other P system variants.

cP rules are high level, supporting one-way first-order syntactic unification (which is similar to pattern matching in functional programming). Unlike other P systems variants, only cP top cells have rewriting rules. Sub-cells in cP systems do NOT have own rules and are only used to represent local data.

We now present a brief overview of cP systems, only focusing on the details needed here, specifically ignoring the inter-top-cell relations and messaging rules, as our solution here consists of one single top-cell.

Using a BNF-like notation, Tables 4 and 5 describe basic structures of cP systems, as used in this paper. The grammar presented in Table 4 describes the contents for top cells and sub-cells, i.e., how data are stored in cP multisets. The grammar presented in Table 5 describes the high-level rewriting rules for cP systems.

Table 4 BNF grammar for cP top cells

```

<top-cell> ::= <state> <term> ...
<state> ::= <atom>
<term> ::= <atom> | <sub-cell>
<sub-cell> ::= <compound-term> ...
<compound-term> ::= <functor> <args> ...
<functor> ::= <atom>
<args> ::= '( <term> ... ' )
  
```

Table 5 Restricted BNF grammar for cP rules, omitting inter-cell messaging and other features not used here

```

<rule> ::= <lhs>  $\rightarrow_a$  <rhs> <promoters>
<lhs> ::= <state> <vterm> ...
<rhs> ::= <state> <vterm> ...
<promoters> ::= ('|' <vterm>) ...
<vterm> ::= <variable> | <atom> | <compound-vterm>
<compound-vterm> ::= <functor> <vargs> ...
<vargs> ::= '(' <vterm> ... ') ' | '{' <vterm> ... '}'

```

We use standard conventions: the symbol λ denotes the empty multiset; dots ('...') represent zero or more repetitions; atoms are denoted by lower case characters (letters or other symbols); and variables are denoted by uppercase letters, except the special discard variable, denoted by an underscore (_).

Top cells have states and contain multisets of literal atoms and recursively nested compound terms called sub-cells. Functors are sub-cell labels, and their multiset arguments are enclosed in parentheses "(" Top-cell contents are all ground, i.e. cannot contain variables.

Rules are applied in state-based weak priority order (see example 5 in Sect. 2.3) and contain compound terms called vterms, which are similar to the sub-cell terms, but with one critical distinction: vterms may also include variables. State-based weak priority order enables simple but powerful application flow control, specifically branching (if then else) and various looping constructs.

As already mentioned, rules are applied using pattern matching unification between terms, and specifically unification between variables and other terms (atom or compound). Rules can be applied in two modes: in exactly-once (1) or max-parallel mode (+).

As usually, before a rule can be applied, it must match, by way of unification, all conditions specified by its left-hand side, and its promoter and inhibitor constraints. There are two cases: (i) vterm arguments enclosed in round parentheses ')' require complete match and (i) vterm arguments enclosed in curly braces '{}' require partial match, of only the specified contents. The second feature is not frequently needed, but enables partial sub-cell transformations similar to those of other P system variants, without locking the whole sub-cell; this is further described under the title microsurgery.

We emphasise that cP terms and vterms are strictly based on multisets. However, we can straightforwardly emulate other structures, such as numbers and even ordered lists. Essentially, numbers can be represented as multisets solely consisting of repeated occurrences of a designated unary digit, typically I . We do not use lists here, so this topic is not discussed.

Terms with repeated arguments seem to require an order concept. However, we consider that these are just convenient shorthands to nested multiset-based labelled terms.

For example, the term $a(bc)(de)$ is actually a shorthand for $a(bc \cdot (de))$, where the dot functor (\cdot) is system provided. Thus, if a is a sub-cell at nesting depth 1, then b and d are at nesting depths 2 and 3, respectively.

We conclude this subsection by noting that, unlike most other P system variants, cP terms and rules allow crisp algorithm descriptions, with constant-size alphabets, constant-size rulesets and bounded membrane nesting, independent of the size of the problem and number of cells in the system. The cP semantics will be further clarified in the following subsection, by way of examples.

2.3 Examples of cP rules

We now present a few simple but typical rules for cP systems.

1. Change state from s_0 to s_1 and rewrite one pair of a and b into one c , provided that at least one p is present (and will stay unchanged in the cell):

$$s_0 a b \rightarrow_1 s_1 c | p$$

2. Change state from s_0 to s_1 and rewrite all a, b pairs into c 's, in the max-parallel mode, provided that at least one p is present:

$$s_0 a b \rightarrow_+ s_1 c | p$$

3. Change state from s_0 to s_1 , rewrite one compound term $a()$ by adding one I to its contents; variable X is unified to the actual contents of a .

$$s_0 a(X) \rightarrow_1 s_1 a(XI)$$

If the current a already has two copies of I , i.e. $a(II)$, then the result will be an updated copy with three I 's, i.e. $a(III)$ —thereby incrementing its base I contents.

4. Conditionally change state from s_0 to s_1 , rewrite one compound term $a()$ by removing one I from its contents, if there is at least one I among its contents.

$$s_0 a(YI) \rightarrow_1 s_1 a(Y)$$

For example, if the current a already has three copies of I , i.e. $a(III)$, the result will be an updated copy with two I 's, i.e. $a(II)$ —thereby decrementing its base I contents. The rule does NOT apply if the cell does not contain at least one I .

5. A complex operation, highlighting the weak priority order, with resulting state depending on the current cell contents.

$$s_0 a \rightarrow_1 s_1 e \quad (1)$$

$$s_0 b \rightarrow_1 s_2 f \quad (2)$$

$$s_0 c \rightarrow_1 s_1 g \quad (3)$$

- (a) If the cell contains a and c , then rules (1) and (3) apply; new state: s_1 , new contents: e and g .
- (b) If the cell contains b and c , then only rule (2) applies; new state: s_2 , new contents: f and c . Rule (3) is NOT applicable, because rule (2) has already set the target state to s_2 .
- (c) If the cell contains a , b and c , then only rules (1) and (3) apply; new state: s_1 , new contents: e , b and g . Rule (2) is NOT applicable, because rule (1) has already set the target state to s_1 .

6. Microsurgery is denoted by curly braces $\{ \}$ instead of round parentheses $()$ and enables processing of parts of the inner contents, without locking the rest [16]. Microsurgery allows us to use sub-cells in the same style as we use our own top cells, and also independent cells in other P systems variants. Without microsurgery, this will NOT be possible, because sub-cells do NOT have own rules—instead, their contents need to be manipulated solely by rules of their containing top cells.

For example, the rules:

$$s_0 x\{a\} \rightarrow_+ x\{b\}$$

$$s_0 x\{c\} \rightarrow_+ x\{d\}$$

applied to the term $x(a a c c c e)$ will in one single step result in $x(b b d d d e)$. Without microsurgery, this requires more steps and more complex rules.

Note that microsurgical applications are already the default for top cells, where we do apply partial matching, without locking all the contents. However, for simplicity, we do not use explicit curly braces for the outermost top-cell. For example, these two rules would in fact be equivalent:

$$s_0 a \rightarrow s_0 b \equiv s_0 a \rightarrow s_0 b,$$

3 cP Solution and examples

In this section, we discuss our cP system for solving $QSAT$ for $n \geq 1$, and we illustrate its evolution on Formulae (1, 2), recalled here:

$$\forall x_1 \exists x_2 (x_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2) \quad (1)$$

$$\exists x_1 \forall x_2 (x_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2) \quad (2)$$

We only use one single top-level cell, and we closely follow the parallel pseudocode algorithm listed in Table 3: a layer-by-layer sweep over a virtual tree, in two passes—first top-down and then bottom-up.

Table 6 Cells y form a lookup table for Boolean identity and negation operations: $V = \text{if } S = + \text{then } K \text{ else } \bar{K}$

K	S	V	y cells contents
0	+	0	$y(0)(+)(0)$
0	-	1	$y(0)(-)(1)$
1	+	1	$y(1)(+)(1)$
1	-	0	$y(1)(-)(0)$

Table 7 Cells w form a lookup table for Boolean \vee and \wedge operations: $V = \text{if } Q = \forall \text{ then } V' \wedge V'' \text{ else } V' \vee V''$

Q	V'	V''	V	w cells contents
\forall	0	0	0	$w(\forall)(0)(0)(0)$
\forall	0	1	0	$w(\forall)(0)(1)(0)$
\forall	1	0	0	$w(\forall)(1)(0)(0)$
\forall	1	1	1	$w(\forall)(1)(1)(1)$
\exists	0	0	0	$w(\exists)(0)(0)(0)$
\exists	0	1	1	$w(\exists)(0)(1)(1)$
\exists	1	0	1	$w(\exists)(1)(0)(1)$
\exists	1	1	1	$w(\exists)(1)(1)(1)$

We use six states, $\{s_1, s_2, \dots, s_6\}$, where s_1 is the initial state, and s_6 is the final. The ruleset is shown in two listings: the top-down pass in Table 9 and the bottom-up pass in Table 11. The evolution corresponding to Formula (1) is illustrated in the following tables: Table 8 shows the initial cell contents and state; then, Table 10 traces the top-down evolution, and Table 12 traces the bottom-up evolution. Table 13 traces the bottom-up evolution of the slightly different Formula (2).

3.1 Lookup tables

For efficiency, our cP solution uses two read-only “tables”. Four sub-cells $y()()()$ form a lookup table for Boolean identity and negation operations. Table 6 shows their contents and their interpretation.

Eight sub-cells $w()()()()$ form a lookup table for Boolean \wedge and \vee operations, the actual operation being selected on the corresponding quantifier. Table 7 shows their contents and their interpretation.

3.2 Prefixes and tree levels

During the top-down pass, sub-cell $v()$ is a counter that indicates the tree-level depth, initially $v(n)$, where n is the actual number of quantifiers (or variables).

Sub-cells $p()()$, $q()()$ form 1-based associative arrays that encode the given prefix: $p()()$ contains variables, $q()()$ contains quantifiers. These sub-cells are used as “horizontal” (not nested) stacks, where the top is indicated by the current value of counter $v()$. During the top-down pass, the top elements of $p()()$ are temporarily popped into $h()$, and $q()()$ is

Table 8 Contents of our top-level cell, as initialised for Formula (1): $\forall x_1 \exists x_2 (x_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2)$

$s_1 \nu(11) \mu()$ $p(11)(x_1) p(1)(x_2) h()$ $q(11)(\forall) q(1)(\exists)$ $f(c(x_1 x_2) c(x_1 \bar{x}_2))$ $y(0)(+)(0) y(0)(-)(1) y(1)(+)(1) y(1)(-)(0) // y \text{ lookup table}$ $w(\forall)(0)(0)(0) w(\forall)(0)(1)(0) w(\forall)(1)(0)(0) w(\forall)(1)(1)(1) // w \text{ lookup table}$ $w(\exists)(0)(0)(0) w(\exists)(0)(1)(1) w(\exists)(1)(0)(1) w(\exists)(1)(1)(1)$
--

Table 9 Top-down rules. The three-step loop $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_1$ is repeated n times, followed by the exit $s_1 \rightarrow s_4$

$s_1 \nu(1N) h(_) p(1N)(X)$ $q(1N)(Q) \mu(M)$	\rightarrow_1	$s_2 \nu(N) h(X) \mu(1M) q'(1M)(Q)$	(1)
$s_1 f(c(_) Z)$	\rightarrow_+	$s_4 f(v(0) Z)$	(2)
$s_1 f(c(1) Z)$	\rightarrow_+	$s_4 f(v(1) Z)$	(3)
$s_2 f(Z)$	\rightarrow_+	$s_3 f(0)(Z a(M)(0))$ $f(1)(Z a(M)(1))$ $ \mu(M)$	(4)
$s_3 f(K) \{c(x(I)(S) Y)\}$	\rightarrow_+	$s_1 f\{c(Y)\}$ $ \ h(x(I)(+))$ $ \ y(K)(S)(0)$	(5)
$s_3 f(_) \{c(x(I)(S) Y)\}$	\rightarrow_+	$s_1 f\{c(1)\}$ $ \ h(x(I)(+))$ $ \ y(K)(S)(1)$	(6)

reversed into a similar “horizontal” stack, $q'()$, whose top is indicated by counter $\mu()$ —stack $q'()$ will be used in the bottom-up pass.

Stacks $p()$, $q()$, $q'()$ closely match their namesake variables used in Table 3. Together with their associated counters, $\nu()$ and $\mu()$, these play the role of global variables controlling the two passes.

3.3 Literals encoding and formula sub-cells

Formula literals, i.e. variables and their negations, are given via sub-cells $x()$. We use shorthand notations that closely match the mathematical expression and keep our expression crisp:

$$x_1 \equiv x(1)(+)$$

$$\bar{x}_2 \equiv x(11)(-)$$

This is just a notation convenience, and our rules actually assume the longer version when being matched.

Clauses are given via sub-cells $c()$, having literals as contents, with implicitly assumed Boolean or’s. For example:

$$(x_1 \vee \bar{x}_2) \equiv c(x_1 \bar{x}_2) \equiv c(x(1)(+) x(11)(-))$$

The contents of $c()$ ’s are multisets; thus, the order of literals is irrelevant, but we usually keep it in our listings, for more readability.

Unquantified formulae are given via sub-cells $f()$. Initially, sub-cells $f()$ contain just multisets of clauses. For example, at the root of the virtual tree, the unquantified part of Formula (1) is encoded as:

$$f(c(x_1 x_2) c(x_1 \bar{x}_2))$$

The contained clauses are partially evaluated during the top-down pass, and new contents appear in $f()$ that indicate the path to the root and the final value.

Sub-cells $a()$ form a 1-based associative array that indicates a complete path to the root and are used as “horizontal” (not-nested) stacks, with the top indicated by the contents of counter $\nu()$ —similar to the above-mentioned global $q'()$ sub-cells. For example, ignoring its other contents, a formula associated with the node on leftmost path 01 looks like this:

$$f(\dots a(11)(1) a(1)(0))$$

Table 10 Top-down traces for Formula (1, $\forall \exists$). Continued from Table 8. (Cf. tree (b) in Figure 1)

s_1	$\nu(11) \ h(_) \ p(11)(x_1) \ p(1)(x_2) \ q(11)(\forall) \ q(1)(\exists) \ \mu()$ $f(c(x_1 \ x_2) \ c(x_1 \ \bar{x}_2))$
$(1) \Rightarrow \ s_2$	$\nu(1) \ h(x_1) \ p(1)(x_2) \ q(1)(\exists) \ \mu(1) \ q'(1)(\forall)$
$(4) \Rightarrow \ s_3$	$f(0)(c(x_1 \ x_2) \ c(x_1 \ \bar{x}_2) \ a(1)(0)) \ f(1)(c(x_1 \ x_2) \ c(x_1 \ \bar{x}_2) \ a(1)(1))$
$(5) \Rightarrow \ s_1$	$f(c(x_2) \ c(\bar{x}_2) \ a(1)(0)) \ f(c(1) \ c(1) \ a(1)(1))$
$(1) \Rightarrow \ s_2$	$\nu() \ h(x_2) \ \mu(11) \ q'(11)(\exists) \ q'(1)(\forall)$
$(4) \Rightarrow \ s_3$	$f(0)(c(x_2) \ c(\bar{x}_2) \ a(11)(0) \ a(1)(0)) \ f(1)(c(x_2) \ c(\bar{x}_2) \ a(11)(1) \ a(1)(0))$ $f(0)(c(x_2) \ c(\bar{x}_2) \ a(11)(0) \ a(1)(1)) \ f(1)(c(x_2) \ c(\bar{x}_2) \ a(11)(1) \ a(1)(1))$
$(5) \Rightarrow \ s_1$	$f(c() \ c(1) \ a(11)(0) \ a(1)(0)) \ f(c(1) \ c() \ a(11)(1) \ a(1)(0))$ $f(c(1) \ c(1) \ a(11)(0) \ a(1)(1)) \ f(c(1) \ c(1) \ a(11)(1) \ a(1)(1))$
$(2, 3) \Rightarrow \ s_4$	$f(v(0) \ c(1) \ a(11)(0) \ a(1)(0)) \ f(v(0) \ c(1) \ a(11)(1) \ a(1)(0))$ $f(v(1) \ c(1) \ a(11)(0) \ a(1)(1)) \ f(v(1) \ c(1) \ a(11)(1) \ a(1)(1))$

Table 11 Bottom-up rules. One-step loop $s_5 \rightarrow s_5$ repeated n times

$s_4 \ f\{ \ c(_) \ }$	\rightarrow_+	$s_5 \ f\{ \}$	(7)
$s_5 \ f(v(V') \ a(M)(0) \ A)$ $f(v(V'') \ a(M)(1) \ A)$	\rightarrow_+	$s_5 \ f(v(V) \ A)$ $\mid \ \mu(M)$ $\mid \ q'(M)(Q)$ $\mid \ w(Q)(V')(V'')(V)$	(8)
$s_5 \ \mu(1M) \ q(1M)(_) \$	\rightarrow_1	$s_5 \ \mu(M)$	(9)
$s_5 \ f(v(V)) \ \mu() \ \nu() \ h(_) \$	\rightarrow_1	$s_6 \ v(V)$	(10)

The contents inside the first parentheses indicate the depths (here 2 and 1), while the content inside the second parentheses indicates the assigned values (here 1 and 0).

Note that the layers are processed in the order of variables given by the prefix—this will be discussed shortly. This need not be in increasing order, but usually is. Thus, the above $a()$'s may indicate the tree node for $x_1 = 0, x_2 = 1$, (Cf. Figures 1, 2).

The $a()()$'s are created during the top-down pass and effectively used during the bottom-up pass, to properly match sibling nodes.

At the tree leaves level, the formulae are completely evaluated and their values are stored in $\nu()$ sub-cells. For example, under the above-mentioned sample assumptions:

$$f(v(0)\dots a(11)(1) a(1)(0)) \iff (x_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2)[x_1 := 0, x_2 := 1] \equiv (0 \vee 1) \wedge (0 \vee \bar{1}) \equiv 0$$

To help an efficient top-down formula substitution split, such as $[x_i = 0]$ vs. $[x_i := 1]$, we also use temporary variants of f with two distinct arguments, $f()()$.

Cells $f()$ closely match their namesake variables used in Table 3.

Table 12 Bottom-up traces for Formula (1, $\forall \exists$). Continued from Table 10. Final result is **false**. (Cf. tree (c) in Figure 1)

s_4	$\mu(11) \quad q'(11)(\exists) \quad q'(1)(\forall) \quad // \quad \nu() \quad h(x_2)$ $f(v(0) \ c(1) \ a(11)(0) \ a(1)(0)) \quad f(v(0) \ c(1) \ a(11)(1) \ a(1)(0))$ $f(v(1) \ c(1) \ a(11)(0) \ a(1)(1)) \quad f(v(1) \ c(1) \ a(11)(1) \ a(1)(1))$
(7) \Rightarrow s_5	$f(v(0) \ a(11)(0) \ a(1)(0)) \quad f(v(0) \ a(11)(1) \ a(1)(0))$ $f(v(1) \ a(11)(0) \ a(1)(1)) \quad f(v(1) \ a(11)(1) \ a(1)(1))$
(8, 9) \Rightarrow s_5	$f(v(0) \ a(1)(0))$ $f(v(1) \ a(1)(1))$ $\mu(1) \quad q'(1)(\forall) \quad // \quad \nu() \quad h(x_2)$
(8, 9) \Rightarrow s_5	$f(v(0))$ $// \quad \mu() \quad \nu() \quad h(x_2)$
(10) \Rightarrow s_6	$v(0) \quad // \quad \mathbf{false}$

Table 13 Bottom-up traces for Formula (2, $\exists \forall$). Continued from Table 10, with *different* $q'()$'s. Final result is **true**. (Cf. tree (c) in Figure 2)

s_4	$\mu(11) \quad q'(11)(\forall) \quad q'(1)(\exists) \quad // \quad \nu() \quad h(x_2)$ $f(v(0) \ c(1) \ a(11)(0) \ a(1)(0)) \quad f(v(0) \ c(1) \ a(11)(1) \ a(1)(0))$ $f(v(1) \ c(1) \ a(11)(0) \ a(1)(1)) \quad f(v(1) \ c(1) \ a(11)(1) \ a(1)(1))$
(7) \Rightarrow s_5	$f(v(0) \ a(11)(0) \ a(1)(0)) \quad f(v(0) \ a(11)(1) \ a(1)(0))$ $f(v(1) \ a(11)(0) \ a(1)(1)) \quad f(v(1) \ a(11)(1) \ a(1)(1))$
(8, 9) \Rightarrow s_5	$f(v(0) \ a(1)(0))$ $f(v(1) \ a(1)(1))$ $\mu(1) \quad q'(1)(\exists) \quad // \quad \nu() \quad h(x_2)$
(8, 9) \Rightarrow s_5	$f(v(1))$ $// \quad \mu() \quad \nu() \quad h(x_2)$
(10) \Rightarrow s_6	$v(1) \quad // \quad \mathbf{true}$

3.4 Top-down pass

The rules for the top-down pass are listed in Table 9. Essentially, we have a loop consisting of three steps ($s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_1$) that is repeated n times and a subsequent one-step evaluation ($s_1 \rightarrow s_4$). These steps closely follow the top-down pass of the parallel algorithm presented in Table 3.

Rules (1, 2, 3) form an if then else construct. If we have not yet processed all quantifiers and variables, condition detected by a nonempty $\nu()$ counter, then rule (1) applies,

resetting our global control variables and starting one more loop iteration ($s_1 \rightarrow s_2$). Sub-cell $h()$ is updated to the current variable to be substituted, say $h(x_i)$, and its associated quantifier is popped into stack $q'()$, to be used in the bottom-up pass.

Otherwise, if the quantifiers and variables stacks are empty, we exit the loop via rules (2,3), applied in max-parallel mode. Formulae $f()$ that after partial evaluations are false, detected by at least one empty $c()$ clause are tagged by one $\nu(0)$ sub-cell. The other formulae, which are true, are tagged by one $\nu(1)$ sub-cell.

Together, rules (4,5,6) form the main body of the top–down loop ($s_2 \rightarrow s_3 \rightarrow s_1$). They run in max-parallel mode and create the next level down the tree, discarding the current level. Each formula $f()$ is split into two children formulae, by two substitutions, $x_i := 0, x_i := 1$, and new $a()$ sub-cells are created, to record the corresponding tree paths. These paths tags $a()$ will be essentially used during the bottom–up pass, when these two children will be recognised as siblings and merged together (despite being here thrown into an unordered multiset).

Using the lookup table $y()()$, rules (5,6) also perform straightforward partial evaluations, based on the values that are assigned to variable x_i .

Table 10 illustrates this top–down pass by traces for Formula (1), starting from the initial state shown in Table 8.

3.5 Bottom–up evaluation

The rules for the top–down traversal pass are listed in Table 11. Essentially, we have a one-step transition from the top–down pass ($s_4 \rightarrow s_5$), followed by a one-step loop ($s_5 \rightarrow s_5$) that is repeated n times, and a one-step exit to the final state ($s_5 \rightarrow s_6$). These steps closely follow the bottom–up pass of the parallel algorithm presented in Table 3.

Rule (7) runs in max-parallel mode and performs a clean-up step ($s_4 \rightarrow s_5$), removing unwanted material from all sub-cells $f()$.

Rules (8,9,10) form a repeat until bottom–up loop, with the exit condition checked by rules (9,10). This works, as we assume that $n \geq 1$.

Rule (8) forms the main body of this bottom–up loop, $s_5 \rightarrow s_5$, that is repeated n times and runs in max parallel mode. This rule creates the next level up the tree, discarding the current level.

Each pair of sibling formulae $f()$ is merged and evaluated, using the corresponding quantifier from stack $q'()$ (which was saved during the top–down pass). Because we use multisets, we cannot a sequence-based pairing, as in Table 3. From all $f()()$'s in the current multiset content, siblings are grouped together according to their path to root records, given by their contained $a()()$'s. The evaluation is performed with the help from the look-up table $w()()()$.

Rules (9,10) form an if then else loop end check. If we are not yet at the root level, condition detected by a nonempty counter μ , then rule (9) resumes the loop, $s_5 \rightarrow s_5$. Otherwise, rule (10) applies and exits, cleaning all remaining stuff and recording the final value in $v()$.

Table 12 illustrates this bottom–up pass by traces for Formula (1, $\forall \exists$), starting from the end state shown in Table 12. The evolution for the related Formula (2, $\exists \forall$) is only marginally different, but still significantly in the bottom-up pass, when we actually use quantifiers; its bottom-up traces are shown in Table 13.

3.6 Analysis

Proposition 1 *Our solution uses a custom alphabet size of 19.*

The state alphabet is $\{s_1, s_2, s_3, s_4, s_5, s_6\}$.

Our custom alphabet is $\{\exists, \forall, +, -, 0, 1, f, c, q, q', p, x, h, y, w, v, \mu, a, v\}$. \square

Proposition 2 *Our solution uses a ruleset containing ten rules.*

Top–down pass uses six rules (Table 9), and bottom–up pass uses four (Table 11), making a total of ten rules. \square

Proposition 3 *Total runtime is $4n + 3$.*

Top–down runtime (Table 9): The top–down loop $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_1$ runs n times. The transition $s_1 \rightarrow s_4$ runs once, making this pass take $3n + 1$ steps.

Bottom–up runtime (Table 11): The transitions $s_4 \rightarrow s_5$ and $s_5 \rightarrow s_6$ run once. The bottom–up loop $s_5 \rightarrow s_5$ runs n times, making this pass take $n + 2$ steps.

Thus, the total runtime is $\mathcal{O}(n) = 4n + 3$. \square

Proposition 4 *The evolution of our ruleset is totally deterministic.*

Rules that are applicable exactly once (\rightarrow_1) use singleton terms and do not allow any possible choice. Rules that are applicable in the max-parallel mode (\rightarrow_+) make the same multiset transformations, regardless of any hypothetical application order. \square

Proposition 5 *Maximum membrane nesting depth is 6.*

The largest nesting depth in Table 10 occurs in:

$$\begin{aligned} &f(0)(c(x_2) c(\bar{x}_2) a(11)(0) a(1)(0)) \\ &\equiv f(0)(c(x(11)(+)) c(x(11)(-)) a(11)(0) a(1)(0)) \end{aligned}$$

and other similar cells. Denoting nesting depth by δ , we have: $\delta(f) = 1$, $\delta(c) = 3$, $\delta(x) = 4$, $\delta(+)$ = 6. This example is for $n = 2$; however, for larger n , the nesting depth will NOT increase, but rather the “horizontal” number of cells at existing levels. \square

4 Conclusion and future work

We have presented an efficient deterministic cP solution to QSAT that runs in $4n + 3 = \mathcal{O}(n)$ steps, at same order of magnitude as the other P system solutions. However, in contrast to other confluent P system solutions, our cP solution uses a small constant alphabet size (19), a small constant number of rules (10) and very small constant membrane nesting depth (6), independent on the problem size.

We conclude by noting the following open problems, which may need further investigations: Is it possible to design an equivalent cP system, still deterministic, “with an even smaller nesting depth bound, e.g. three or even two?”

How would non-confluent designs affect the power of cP systems? Are cP system inhibitors really needed? We note that our solution does not use any, and the presence of inhibitors will likely negatively affect the simulation runtime on existing hardware and software platforms.

Is a sub-linear runtime solution possible, using cP systems and/or another P systems variant? We note this can be viewed as seeing whether cP systems agree with the parallel computation thesis [15]. This also leads to the question on whether or not cP systems are polynomial equivalent to the other P system variants.

Compliance with ethical standards

Conflict of interest On behalf of all authors, the corresponding author states that there is no conflict of interest.

References

- Păun, G. (2000). Computing with membranes. *Journal of Computer and System Sciences*, 61(1), 108–143.
- Păun, G. (2001). P systems with active membranes: Attacking NP-complete problems. *Journal of Automata, Languages and Combinatorics*, 6(1), 75–90.
- Ionescu, M., Păun, G., & Yokomori, T. (2006). Spiking neural P systems. *Fundamenta Informaticae* 71(2):279–308.
- Martín-Vide, C., Păun, G., Pazos, J., & Rodríguez-Patón, A. (2003). Tissue P systems. *Theoretical Computer Science*, 296(2), 295–326.
- Cooper, J., & Nicolescu, R. (2019). The Hamiltonian cycle and travelling salesman problems in cP systems. *Fundamenta Informaticae*, 164(2–3), 157–180.
- Henderson, A., & Nicolescu, R. (2019). “Actor-like cP Systems,” in *Membrane Computing*, vol. 11399 of *Lecture Notes in Computer Science*, pp. 160–187, Springer.
- Liu, Y., Nicolescu, R., & Sun, J. (2020). Formal verification of cP systems using PAT3 and ProB. *Journal of Membrane Computing*, 2(2), 80–94.
- Leporati, A., Manzoni, L., Mauri, G., Porreca, A., & Zandron, C. (2019). Characterizing PSPACE with shallow non-confluent P systems. *Journal of Membrane Computing*, 1(2), 75–84.
- Ishdorj, T.-O., Leporati, A., Pan, L., Zeng, X., & Zhang, X. (2010). Deterministic solutions to QSAT and Q3SAT by spiking neural P systems with pre-computed resources. *Theoretical Computer Science*, 411(25), 2345–2358.
- Leporati, A., Manzoni, L., Mauri, G., Porreca, A. E., & Zandron, C. (2019). “Solving QSAT in sublinear depth,” in *International Conference on Membrane Computing* (T. Hinze, G. Rozenberg, A. Salomaa, and C. Zandron, eds.), vol. 11399 of *Lecture Notes in Computer Science*, pp. 188–201, Springer.
- Gutiérrez-Naranjo, M. A., Pérez-Jiménez, M. J., & Romero-Campero, F. J. (2006). “A Linear Solution for QSAT with Membrane Creation,” in *Membrane Computing* (R. Freund, G. Păun, G. Rozenberg, and A. Salomaa, eds.), vol. 3850 of *Lecture Notes in Computer Science*, pp. 241–252, Springer.
- Alhazov, A., & Pérez-Jiménez, M. J. (2007). “Uniform solution of QSAT using polarizationless active membranes,” in *International Conference on Machines, Computations, and Universality* (J. Durand-Lose and M. Margenstern, eds.), vol. 4664 of *Lecture Notes in Computer Science*, pp. 122–133, Springer.
- Nicolescu, R., & Henderson, A. (2018) “An introduction to cP Systems,” in *Enjoying Natural Computing: Essays Dedicated to Mario de Jesús Pérez-Jiménez on the Occasion of His 70th Birthday* (C. Graciani, A. Riscos-Núñez, G. Păun, G. Rozenberg, and A. Salomaa, eds.), vol. 11270 of *Lecture Notes in Computer Science*, pp. 204–227, Springer.
- Sipser, M. (2012). *Introduction to the Theory of Computation*. Boston: Cengage Learning.
- Chandra, A. K., Kozen, D. C., & Stockmeyer, L. J. (1981). Alternation. *Journal of ACM*, 28, 114–133.
- Nicolescu, R. (2014). “Parallel thinning with complex objects and actors,” in *International Conference on Membrane Computing* (M. Gheorghie, G. Rozenberg, A. Salomaa, P. Sosik, and C. Zandron, eds.), vol. 8961 of *Lecture Notes in Computer Science*, pp. 330–354, Springer.

Publisher’s Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Alec Henderson is a PhD student at the University of Auckland (New Zealand). His main interests are in distributed computing, artificial intelligence, astrophysics and alternative models of computation.



Radu Nicolescu is a senior lecturer in the School of Computer Science at the University of Auckland (New Zealand). He holds a PhD in mathematics from the University of Bucharest (Romania). Before joining the University of Auckland, he first worked at the University of Bucharest, and then 5 years on industrial optimization and robotics projects (Austria and Germany). His research interests include discrete mathematical models, information coding and complexity, compiler construction,

logical and functional programming, models for parallel and distributed computing.



Michael J. Dinneen received his PhD from the University of Victoria, Canada, in 1996, and is currently a Senior Lecturer at the University of Auckland. In the 1990s, he worked for several years at the Los Alamos National Laboratory in New Mexico, USA, working on grand challenge combinatorial search and optimization problems using supercomputers, such as those developed by Cray Research. Besides his specialty of graph theory and algorithms, he does research on unconventional models

of computation such as (adiabatic) quantum computing and membrane computing, culminating in over 100 research papers.