Membrane computing: The computational power of cP and water systems.

by

Alec S. Henderson

under the supervision of:

Dr. Radu Nicolescu and Dr. Michael J. Dinneen

Advised by:

Mr TN Chan

A Thesis Submitted in Partial Fulfilment of the
Requirements for the Degree of
DOCTOR OF PHILOSOPHY
in
Computer Science
The University of Auckland, 2021

University of Auckland

ABSTRACT

P systems/membrane computing is a parallel and distributed model of computation. This thesis focuses on two recently proposed P system variants: P systems with complex objects (cP systems) and water-based P system (wP system).

This thesis first investigates what can be computed utilising a single cP system cell. We show for the first time that cP systems can solve PSPACE complete problems in polynomial time (linear). We will also, demonstrate the advantages of our solution compared to other P system variants. Following this positive result, we demonstrate efficient solutions to NP-complete problems surpassing previous results.

The thesis then looks at a P system model of water computing. We prove that this model is Turing complete via the construction of μ recursive functions. Following this construction, we demonstrate how this model can be viewed as a restricted cP system. We then prove that this model can construct a PRAM machine. This construction proves that the model can be used as an efficient parallel machine.

Finally, the thesis looks at using cP systems for solving distributed computing problems. We first solve the Byzantine agreement problem using newly proposed actor based controls on the messages. We show that this new solution surpasses the previous solutions in terms of: number of cells, number of steps, rule set size, and many more. We then solve the Santa Claus problem demonstrating the usefulness of cP systems as a concurrency specification language. We demonstrate the reduction of program size of our cP solution compared to similar solutions implemented in modern programming languages.

ACKNOWLEDGEMENTS

First, I would like to thank my supervisors Dr. Radu Nicolescu and Dr. Michael J. Dinneen. The support and guidance throughout this study has been immeasurable.

I would also like to thank my advisor TN Chan you have provided guidance and support throughout my studies.

I would like to thank my co-authors of papers throughout my PhD not necessarily included in this thesis thanks: Dr. Radu Nicolescu, Dr. Michael J. Dinneen, Mr TN Chan, PD Dr.-Ing. habil. Thomas Hinze, Mr Hendrik Happe, Ms Ocean Wu and Mr Yezhou Liu

I would like to thank James Cooper and Yezhou Liu my fellow students working on membrane computing at Auckland we have had many interesting discussions.

I would like to thank Dr Sathiamoorthy Manoharan and Professor Cristian Calude who have both given words of wisdom and useful discussions during this research.

I would like to thank my partner Ms Rujia Liu for being there for me throughout my studies. She has accompanied and supported me throughout this meaningful experience.

Finally, I would like to thank my friends and family for the support during my studies.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 P systems

Whether P equals NP is unquestionably the most important unsolved problem in computational complexity theory. The problem has been studied extensively, with many practical problems found to be NP-complete. However, the currently best-known general solutions to NP-complete problems take prohibitively large amounts of time for large instances.

P systems are a parallel and distributed model of computing, first proposed by Gheorghe Păun in [54]. P systems are an abstract model of membrane systems, with many variants being proposed such as: P systems with active membranes [55], spiking neural P systems [32], tissue P systems [43], and P systems with compound terms (cP systems) [50]. These systems have been found to have efficient solutions to hard problems. However, as far as we know, these efficient solutions are still in theory and have not yet been practically realised.

As discussed in [68] P system research is largely broken into three domains:

- Theory: This area mainly focuses on computability and computational complexity theory of the different P system variants.
- Tools: simulations and verification of different P system variants.
- Applications: Investigating the different applications in which P systems may be advantageous compared to the current systems.

This thesis is mainly centred around theory, with Chapter 3 on computational complexity and Chapter 4 computability. Chapter 5 is based on applications.

1.2 Motivation

cP systems developed in [51] are a variant of P systems that use high level generic rewriting rules. cP systems usually have smaller alphabets and rule sets than other P systems; however, the rules are more complex. cP systems have previously been shown to solve NP-hard problems (NP-complete for the decision version) such as the travelling salesman [14], the subset sum [39] and the 3-colouring problem [13]. However, harder problems had not been solved by cP systems. In this work (Chapter 3), we present, to the best of our knowledge, the first proof that cP systems can solve PSPACE complete problems.

Although cP systems had solved NP-hard problems, each problem solved was a new algorithm not utilising the previous results. Reductions are a well-known technique of using one solution to solve multiple problems with a simple procedure to transfer an instance of the new problem into the old one. As far as we know, we are the first to utilise reductions in cP systems. Not only that, but we constructively give a constant cP reduction from k-colouring to k-sat. Using the reduced problem instance, we find, as far as we know, the fastest k-colouring cP solution even outperforming the previous algorithm specifically designed for k-colouring.

Although we have shown in theory what cP systems can solve efficiently, these have not been practically realised. Another area of research is utilising cP systems as a distributed specification language. Previously, other P system researchers have looked at this, such as [68, 36]. In this work, we further this direction of research in which we present an improved solution to the Byzantine agreement. As well as the byzantine agreement, we look at the Santa Claus problem and demonstrate the usefulness of cP systems as a parallel computing specification language.

1.3 Thesis outline

Chapter 2: Background This chapter introduces the concepts which are used in the later sections of the thesis. We first give definitions of the different computation models used in the thesis (mainly utilised in Chapter 4). We then discuss the different complexity classes which play an essential role in the constructions in Chapter 3. We finally discuss P and cP systems which are utilised throughout the thesis.

Chapter 3: Computational power of a single cell This chapter is dedicated to demonstrating constructively that polynomial-time cP systems can solve 'hard' problems. This chapter is broken into two parts :

1) We solve QSAT, 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. This is based on our work presented in [26].

2) We present a sublinear solution to k-SAT and demonstrate that k-colouring can be reduced to k-SAT in constant time. This work not only demonstrates that traditional reductions are efficient in cP systems, but also that they can produce more efficient solutions than the previous problem-specific solutions. This work is based on our work presented in [27].

Chapter 4: Computing with water This chapter is dedicated to a new P system originally proposed in [30]. The P system is a water-based computing device with no central control. This chapter is broken into two parts :

1) We further develop water computing as a variant of P systems. We propose an improved modular design, which duplicates the main water flows by associated control flows. We first solve the three open problems of the previous design by demonstrating: how functions can be stacked without a combinatorial explosion of valves; how termination of the system can be detected; and how to reset the system. We then prove that the system is Turing complete by modelling the construction of μ-recursive functions. The new system is based on directed acyclic graphs, where tanks are nodes and pipes are arc; there are no loops anymore, water falls strictly in a 'top down' direction. Finally, we demonstrate how our water tank system can be viewed as a restricted version of cP systems. This is based on our work presented in [28].

2) We further the work on a recently proposed membrane computing model which utilises decentralised water tanks interconnected by pipes with water flow controlled by valves. We demonstrate that such systems can construct 'efficiently': 1) A programmable sequential, random-access machine (RAM), which we then extend to construct: 2) a programmable exclusive read exclusive write (EREW) parallel random-access machine (PRAM).

Chapter 5: Applications in distributed computing This chapter is dedicated to multi-cell cP systems. This chapter is broken into two parts :

1) We propose a revised version of the previous best models for the Byzantine agreement problem—a famous problem in distributed algorithms, with non-trivial data structures and algorithms. The new actor-based solution uses a substantially shorter fixed-sized alphabet and ruleset, independent of the problem size. Moreover, in contrast to the previous models, additional helper/firewall cells are not anymore needed to ensure protection against Sybil attacks. Also, as with any standard distributed algorithm, the novel actor-based cP model uses exactly one top-level cell for each process in Byzantine agreement, thus solving another open problem. This is based on our work presented in [25].

2) We discuss the Santa Claus Problem, a classic and challenging concurrency problem, and propose a slightly tighter specification, that precludes a few odd scenarios. We provide an elegant solution to this refined problem using cP systems. Previously, cP systems have been successfully used as a specification language for a wide variety

of problems, such as NP-hard problems, graph algorithms, image processing, distributed algorithms (e.g. Byzantine agreement). We now evaluate, for the first time, our cP systems as a concurrency specification language. We compare our cP specification and similar solutions implemented in several modern languages (e.g. Go, C#, F#), with respect to program-size complexity and (where available) runtime performance. This is based on our work presented in [29].

Chapter 6: Conclusions In this chapter, we critically discuss the results and suggest some possible future directions.

Chapter 2

Background

In this chapter we introduce some of the necessary material for the rest of the thesis. We start with defining the models of computation which we use in Chapter 4. We then discuss the computational complexity classes used in Chapter 3. Finally we discuss cP systems which are used throughout the thesis. For readers familiar with the topics covered, this chapter can be skipped.

Saturation arithmetic. As discussed in [69], saturation arithmetic restricts operations to a fixed range. If an operation results in a number exceeding the upper bound, the upper bound is the result. Conversely, if the result of the operation goes below 0, then the result is 0. If the upper and lower bounds are $+\infty$ and $-\infty$ respectively, then saturation arithmetic is standard arithmetic. For example, in the range [0,100], $5 \times 30 = 100$ and $20 - 30 = 0$. We further denote saturating addition as $\oplus$, and saturating subtraction as $\ominus$.

2.1 Traditional computing models

In this section, we describe three traditional computational models: Random access machine (RAM), Parallel random access machine (PRAM), and general recursive functions (μ-recursive functions). The Church–Turing thesis states that a function on the natural numbers can be calculated effectively if and only if it is computable by a Turing machine. This means that if another computational model of computation can simulate a Turing machine, it can calculate all effectively calculable functions. RAM and μ-recursive functions have both been shown to be able to simulate TMs.

2.1.1 RAM

There have been many models of computation proposed, with the 'sequential machines' typically being the deterministic Turing machine and the random access machine [53](RAM). We focus on the RAM in this work as they usually have more

practical uses than the traditional Turing model. For example, implementing the GCD algorithm using a RAM as seen in Table 2.2 is straightforward whereas, for a Turing machine it is not.

A RAM consists of a finite program of m lines and a potentially infinite sequence of registers $r_1, r_2, ...r_n$. The program of the RAM will consist of a sequence of operation codes and parameters. Based on the definition in [12], a RAM has the following operations:

1. $r_i \leftarrow C$: assign a constant value C to register i.
2. $r_i \leftarrow r_j \oplus r_k$: add the value of two registers j and k and assign to register i.
3. $r_i \leftarrow r_j \ominus r_k$: subtract from register j the value stored in k and assign to register i.
4. $r_i \leftarrow r_{r_j}$: get the value y from register j, then get the value from register y and assign to register i
5. $r_{r_i} \leftarrow r_j$: get the value y from register j, then get the value x from register i and assign y to register x.
6. TRA m $r_i > 0$: go to program line m (control transferred to line m of the program) if r_i greater than 0, otherwise go to the next line.

Where we use the item number as the op code as seen in Table 2.1.

We assume that the output will be the values stored in the registers once the program has halted. A program halts when it has gone to a line number in the program which is not defined. For example, consider the following Euclidean algorithm pseudocode for positive integers:

```
1 function gcd(a,b)
2     while (a ≠ b)
3         if (a > b) then
4             a ← a ⊖ b
5         else
6             b ← b ⊖ a
7     return a
```

Table 2.1: Operations and there corresponding opcodes.

Operation	Opcode
$r_i \leftarrow C$	1 i C
$r_i \leftarrow r_j \oplus r_k$	2 i j k
$r_i \leftarrow r_j \ominus r_k$	3 i j k
$r_i \leftarrow r_{r_j}$	4 i j
$r_{r_i} \leftarrow r_j$	5 i j
TRA m $r_i > 0$	6 m i

This pseudocode translates into the RAM code presented in Table 2.2.

We note that our RAM model has a fixed size program with m lines. This can implement RAM programs of length less or equal m. Noting that the machine halts when it gets to line $m+1$. Hence, if a program had less than m lines, an additional line could be added to jump to line $m+1$.

2.1.2 PRAM

As the RAM model we have discussed is inherently sequential, one can extend it to be parallel. A parallel RAM machine (PRAM) is one of the most well-known parallel computing models. As defined in [22, 23] a PRAM consists of a set of processors $p_1, p_2, ...$ and a set of shared registers $r_1, r_2, ...$Each processor can be viewed as an individual RAM with its own program and sequence of registers.

Processors can only communicate via the shared memory using read or write operations. The processors execute in a synchronous manner, with each step taking the same amount of time. Typically there are three models of PRAM discussed in the literature to model the read and write of the shared memory they are:

- Exclusive read exclusive write (EREW): only one processor can read or write to each shared register at a time.
- Concurrent read exclusive write (CREW): Any number of processors can read from the same shared register simultaneously, but only one can write to each shared memory location.
- Concurrent read concurrent write (CRCW): Any number of processors can read and write the same shared register simultaneously.

The CRCW PRAM is typically further defined based on different ways of handling concurrent writes; these include [20]:

Table 2.2: Euclidean algorithm implemented for RAM machine.

Line number	Operation	Opcode	Comment
1	$r_3 \leftarrow r_1 \ominus r_2$	3 3 1 2	
2	TRA 6 $r_3 > 0$	6 6 3	Go to line 6 if $a > b$
3	$r_3 \leftarrow r_2 \ominus r_1$	3 3 2 1	
4	TRA 8 $r_3 > 0$	6 8 3	Go to line 8 if $b > a$
5	TRA 10 $r_1 > 0$	6 10 1	Halt $a = b$ result is in r_1
6	$r_1 \leftarrow r_1 \ominus r_2$	3 1 1 2	$a \leftarrow a \ominus b$
7	TRA 1 $r_1 > 0$	6 1 1	Start the loop again
8	$r_2 \leftarrow r_2 \ominus r_1$	3 2 2 1	$b \leftarrow b \ominus a$
9	TRA 1 $r_2 > 0$	6 2 1	Start the loop again

- Collision: A collision symbol is written. This does not give details about which processors caused the collision or what they were attempting to write.
- Common: Successful write only if all processors writing to the same location are writing the same value.
- Arbitrary: Only one arbitrary attempt is successful, but only if all choices are confluent(end with the same final result).
- Priority: The processor with the lowest ID's write is successful.

2.1.3 μ Recursion

Although machine models of computation are closely related to the everyday computer we all use, it is not the only way to define a model of computation. One of the other popular models is μ recursion in which computation is defined via functions. In [9, 57, 60, 8] it was shown that the following base functions and closure operators are sufficient to construct the unary primitive recursive functions:

- **Successor function**: $S(x) = x + 1$
- **Subtraction function**: $B(x, y) = x - y$
- **Composition operator**: $C(h, g)(x) = h(g(x))$
- **Difference operator**: $D(f, g)(x) = g(x) - f(x)$
- **Primitive recursion operator**: $P(f) = p, p(0) = 0$, $p(x + 1) = f(p(x))$
- **Addition function**: $A(x, y) = x + y$
- μ **operator**: $\mu_y(f)(x) = \min_y\{f(x, y) = 0\}$

2.2 Computational complexity

In this section, we define and discuss the computational complexity classes which appear later in this thesis. This section will only briefly touch on these classes but give sufficient knowledge to understand our constructions later. We direct interested readers to the well-known texts [3, 61] for more information.

2.2.1 Polynomial time

The first complexity class we discuss is polynomial time (P). The class P is all languages that can be decided by a polynomial-time Turing machine. Of course, we have not discussed Turing machines in detail and so can utilise the sequential thesis, which states:

Thesis 2.2.1. *All reasonable sequential computational models are polynomially equivalent.*

Therefore, we can think of P as the class decided in polynomial time by the sequential RAM model previously discussed. Where polynomial time for a RAM means all computational paths on input x will are of length $O(|x|^y)$ where y is a fixed constant. One example of a polynomial computation is our algorithm for the GCD. We do note that we compute the GCD, so we are solving a function problem rather than a decision problem, however.

In this thesis, we are not focused on class P. We are focused on what can be computed by certain models in polynomial time. We do emphasise that although we prove our systems can compute hard problems, they are not sequential machines. This means that they do not show that these problems are in P.

2.2.2 NP-complete

NP-complete languages have been studied for decades, and knowing whether the complexity class is within P is one of the most important questions in theoretical computer science. The typical definition for a language to be NP-complete is that it is NP-hard and an element of NP. NP-hardness means that all languages in NP polynomially time reduce to it. A great deal of these problems have significant practical importance. There are practicality hundreds, if not thousands, of problems that have been found to be NP-complete. Of course, NP-completeness deals with decision problems; however, all NP-complete languages being self-reducible means we do not need to study optimisation versions as much [3].

SAT

The Boolean satisfiability problem (SAT) is one of the most famous NP-complete problems and also the first problem shown to be NP-complete [11]: given a Boolean formula, does there exists a satisfying assignment? Typically the problem considers formulas in conjunctive normal form (CNF). A Boolean formula is in 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:

$$(x_1 \vee x_2) \wedge (\bar{x_1} \vee \bar{x_2})$$

The k-SAT problem is a restricted version of SAT, where each clause contains at most k variables. This restricted version is also NP-complete for $k \geq 3$.

Polynomial time reductions

As defined in [61], given two languages $A, B \subseteq \Sigma^*$, A is polynomial time mapping reducible (also known as Karp reducible) to a language B ($A \leq_p B$) if a polynomially computable function $f : \Sigma^* \to \Sigma^*$ exists where for every w:

$$w \in A \iff f(w) \in B$$

The function f is called the polynomial-time reduction. The well-known proof by Cook [11] shows how all languages in NP have a polynomial-time reduction to SAT. Here, we describe the reduction from k-colouring to k-SAT [62].

Useful Boolean formulas As discussed in [62], we can make some useful formulas in CNF, which simplify the reductions to SAT. *at_most_one* is a formula which defines the property only one literal of the arguments is **true**:

$$at\_most\_one(l_1, l_2, \ldots, l_n) = \bigwedge_{1 \leq i < j \leq n} (\bar{l_i} \vee \bar{l_j})$$

Similarly we can define *at_least_one* meaning at least one variable is **true**:

$$at\_least\_one(l_1, l_2, \ldots, l_n) = (l_1 \vee l_2 \vee \ldots \vee l_n)$$

Combining these, we can also define *exactly_one* where exactly one of the variables will be **true**:

$$exactly\_one(l_1, l_2, \ldots, l_n) = at\_most\_one(l_1, l_2, \ldots, l_n) \wedge \ at\_least\_one(l_1, l_2, \ldots, l_n)$$

The formula given for *at_most_one* is not the most efficient and can be implemented in $O(n)$ rather than the $O(n^2)$ version we defined, as discussed in [62]. This more efficient implementation does, however, introduce $n - 2$ more variables. Throughout the rest of this thesis we assume the inefficient encoding is used.

k-colouring As defined in [62], given a graph G with vertices V and edges E, G is k-colourable if there exists a function f:

$$f : V \to \{1, 2, \ldots, k\} \text{ such that for all } \{u, v\} \in E, f(u) \neq f(v),$$

Here we show the reduction from an instance of k-colouring to SAT given in [62]. The set of variables is denoted X, formula as F, and a set K as $\{1, 2, \ldots, k\}$.

$$X = \{x_{v,i} : v \in V, i \in K\}$$

$$F = \bigwedge_{v \in V} exactly\_one(x_{v,i} : i \in K) \wedge \bigwedge_{\{u,v\} \in E} \bigwedge_{i \in K} (\bar{x}_{u,i} \vee \bar{x}_{v,i}) \tag{2.1}$$

Each vertex in the graph is represented in the formula by k variables. Where, each variable in the formula represents a vertex assigned to the colour i.

The first part of the formula ($\bigwedge_{v \in V} \mathit{exactly\_one}(x_{v,i} : i \in K)$) represents the requirement that a vertex must take exactly one colour. This will create $|V|\binom{k}{2}+|V|k$ clauses, i.e. $O(|V|k^2)$.

The second part of the formula ($\bigwedge_{\{u,v\} \in E}$ $\bigwedge_{i \in \kappa}(\bar{x}_{u,i} \vee \bar{x}_{v,i})$) ensures that each pair of vertices connected by an edge will have a different colour. This will create $O(|E|k)$ clauses, with each clause containing two variables.

Theorem 2.2.1. *The clause set for k-colourable is linear in the input size.*

Based on the previous analysis, we know that we create $O(|V|k^2)$ clauses for the first part of the formula. We also know that we create $O(|E|k)$ clauses for the second part, each being two variable length. Therefore the entire formula is $O(|V|k^2 + |E|k)$ characters. Due to k being a fixed constant, we know the length will be $O(|V|+|E|)$, which is linear in the input size.

2.2.3 PSPACE

QSAT

The SAT formulae discussed previously assumed *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:

$$\exists x_1 \; \exists x_2 \; (x_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{x}_2).$$

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 is 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 [61]:

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

As shown in [10] 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 [33, 38, 24, 2] also make the same assumptions.

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

$$\forall x_1 \; \exists x_2 \; (x_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2). \tag{2.2}$$

$$\exists x_1 \; \forall x_2 \; (x_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2). \tag{2.3}$$

2.3 cP systems

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 (Figure 2.1); and (iii) evolve by applying formal multiset rewriting rules, with additional messaging primitives between top-cells.

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 (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 their 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.

Using a BNF-like notation, Tables 2.3 and 2.4 describe basic structures of cP systems. The grammar presented in Table 2.3 describes the contents for top-cells and sub-cells, i.e., how data is stored in cP multisets. The grammar presented in Table 2.4 describes the high-level rewriting rules for cP systems.

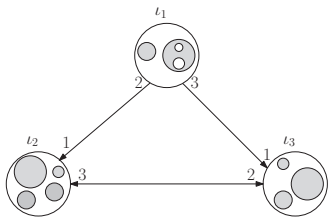


Figure 2.1: Bird's eye view of a sample cP system, with top-level cells and subcells.

Table 2.3: BNF grammar for cP top-cells.

```
1 <top-cell> ::= <state> <term> ...
2 <state> ::= <atom>
3 <term> ::= <atom> | <sub-cell>
4 <sub-cell> ::= <compound-term> ...
5 <compound-term> ::= <functor> <args> ...
6 <functor> ::= <atom>
7 <args> ::= '(' <term> ... ')'
```

Table 2.4: BNF grammar for cP rules, omitting features not used here.

```
1 <rule> ::= <lhs> →a <rhs> (<promoters> | <inhibitors>) ...
2 <lhs> ::= <state> (<term> | ?δ '{' <term> '}')...
3 <rhs> ::= <state> (<term> | !δ '{' <term> '}')...
4 <promoters> ::= ('|' <vterm-eq>) ...
5 <inhibitors> ::= ('¬' <vterm-eq>)...
6 <vterm> ::= <variable> | <atom> | <compound-vterm>
7 <compound-vterm> ::= <functor> <vargs> ...
8 <vterm-eq> ::= <vterm> | '('<vterm> '=' <vterm>')'
9 <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, with the exception of 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.

A cP system evolves through a sequence of configurations by changing its state and contents. These changes are driven by the high-level rewriting rules associated to its top-cell, which are constructed according to the grammar presented Table 2.4. Unlike similar cells in cell-like P systems, cP sub-cells are more restricted, by not having their own rules. Thus, sub-cells are just data storage facilities, and are acted upon by the top-cell's rules only. Unlike other P systems variants, rules in cP systems are generic templates, i.e., their var-objects may contain variables that must be instantiated before the rule application. Before a rule can apply:

- Its left hand side (lhs) state must match the current top-cell state.
- Its right hand side (rhs) state must match the already committed next state, if any, as further detailed below, in the section on weak priority order.
- The rule must be 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*; (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*.

Rules are applied in *state-based weak priority order* (see example 5 Section 2.3) with rules considered in the given top-down. The first lhs state is the state of the initial configuration. Once an applicable rule has been found, this commits to the next state, with subsequent rules committing to different states disabled. Rules going to the same state as the applicable rule, which can also be applied, will be applied in the same step. This state-based weak priority order supports a straightforward emulation of basic control flow (e.g. conditional, or loop structures).

Essentially, applying a rule:

- Commits to the next state (which may be the current state).
- Consumes (deletes) extant top-cell objects matching its lhs. Promoters must match and inhibitors not match extant top-cell objects, with both not being consumed by the rule.
- Creates new objects as indicated by its instantiated rhs. Newly created objects are temporary unavailable and become available after the end of the current step only, as in traditional P systems.

There are two rule application modes: exactly-once ($\rightarrow 1$) and max-parallel ($\rightarrow +$). An exactly-once rule will apply for one single matching (non-deterministically chosen). A max-parallel rule will apply it as many times as possible, conceptually all in the same step, but following a serialisation semantics, i.e., its effects must be identical to a sequential repetition of the same rule in the exactly-once mode (sequence non-deterministically chosen). Although, as just mentioned, the cP semantics allow non-deterministic computations, most of our work has focused on confluent evolutions, often deterministic; the solution proposed in this paper is deterministic.

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 *1*. We do not use lists here, so this topic is not discussed.

Terms with repeated arguments seem to require an ordering 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 the number of cells in the system. The cP semantics will be further clarified in the following subsection, by way of examples.

2.3.1 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):

```
1   s0  a  b  →1  s1  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:

```
1   s0  a  b  →+  s1  c  |  p
```

3. Change state from s_0 to s_1 and rewrite *all* a, b pairs into c's, in the max-parallel mode, provided that no p is present:

```
1   s0  a  b  →1  s1  c  ¬  p
```

4. Change state from s_0 to s_1, rewrite one compound term $a()$ by adding one *1* to its contents; variable X is unified to the actual contents of a.

```
1   s0  a(X)  →1  s1  a(X1)
```

   If the current a already has two copies of *1*, i.e. $a(11)$, then the result will be an updated copy with three *1*'s, i.e. $a(111)$—thereby *incrementing* its base 1 contents.

5. *Conditionally* change state from s_0 to s_1, rewrite one compound term $a()$ by removing one *1* from its contents, *if* there is at least one *1* among its contents.

```
1   s0  a(Y1)  →1  s1  a(Y)
```

   For example, if the current a already has three copies of *1*, i.e. $a(111)$, the result will be an updated copy with two *1*'s, i.e. $a(11)$—thereby *decrementing* its base 1 contents. The rule does NOT apply if the cell does not contain at least one *1*.

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

```
1   s0  a  →1  s1  e      (1)
2   s0  b  →1  s2  f      (2)
3   s0  c  →1  s1  g      (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.

7. **Microsurgery** is denoted by curly braces { } instead of round parentheses ( ) and enables processing of *parts* of the inner contents, *without locking* the rest [48]. Microsurgery allows us to use sub-cells in the same style as we use our top-cells, and also independent cells in other P systems variants. Without microsurgery, this will NOT be possible, because sub-cells do NOT have their own rules – instead, their contents need to be manipulated solely by the rules of their containing top-cells.

   For example, the rules:

```
1  s0 x{a} →+ x{b}
2  s0 x{c} →+ 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:

```
1  s0 a → s0 b ≡ s0 {a} → s0 {b},
```

2.3.2 Multi Cell

Here we utilise our revised version of cP systems, which includes a new Actor-like facility for controlling the message flow. Essentially, we have now two communication primitives:

- '!' : a *send* primitive, which can only appear on the the *rhs* of the rules, and sends messages over *outgoing* arcs;

- '?' : a *receive* primitive, which can only appear on the the *lhs* of the rules, and receives (accepts) messages from *incoming* arcs.

Sent messages which arrive at the target cell are NOT immediately inserted among the target's contents; instead, these messages are conceptually "enqueued" and there

is one message "queue" – in our case, read "multiset" – for each incoming arc. Then, these messages may be accepted by the receive primitive, '?', only at appropriate target steps. Moreover – like the send primitive '!' – the receive primitive '?' can also be associated with any term (including compound), which enables it to filter only queued messages meeting a specific pattern. Depending on the runtime, the designated receive and/or send location may be a cell id or an arc id. Throughout this thesis, we assume that they are cell ids.

For example, consider two top-level cells, ι_1 and ι_2, connected by a communication channel symbolised by the (directed) arc (ι_1, ι_2), which is labelled by 2 at its source and 1 at its target. Consider that cell ι_1 sends one a and then one b, via two steps, as shown by the following two rules:

S_0	$\rightarrow_1$	S_1	$!_2\,\{a\}$	ι_1 sends a to ι_2
S_1	$\rightarrow_1$	S_2	$!_2\,\{b\}$	ι_1 sends b to ι_2

The sent items, a and b, are not automatically inserted into the contents of ι_2. Consider that target cell ι_2 is in state S_0 and has the following rule:

S_0	$?_1\,\{b\}$	$\rightarrow_1$	S_1 $\quad c\ c$	ι_2 expects b from ι_1

Then, cell ι_2 does not accept the first sent item, a, which remains in the inbox multiset (and may be accepted later). Thus, ι_2 idles one step, until it receives the second item, b, which is transformed by the rule into two c's. More complex (and interesting) scenarios can be designed using compound terms and variables.

For simplicity, we include a send to all neighbours(broadcast) $!_\forall$.

2.4 Computing with water

Water has been used for information processing for over 2000 years [41]. Although water computers are not commonly used today, they have had many successful uses in the past. In 1901, water was used to calculate the nth root of a polynomial [19]. In the 1930s, water integrators were made to solve ordinary differential equations and were not surpassed by digital computers until the 1980s [65]. In the late 1940s, the first Phillips machine was built [56]. The Phillips machine was used to model macroeconomic theory. It was used in lectures for many years after the original prototype of the 1940s [58]. In the 1960s, water was used to implement logic gates such as AND, OR and NOT [45]. In the early 2000s, a fluid-based bilateral system was proposed and used to solve the satisfiability problem [4]. Recent work has investigated the possibility of using a microfluidic biochip as a way of implementing spiking neural P systems [34]. For a more detailed history see [1]. As our work is theoretical, we note that physical implementations of water-based machines are still being developed, such as in [64].

In [30] a new model of water computing was proposed. The model worked by having

a set of tanks interconnected using pipes. Each pipe is controlled by a set of valves that allow water to pass through it if and only if all valves on the pipe are open. The system would terminate after an arbitrary given amount of time, making it undecidable to determine if the system had completed its computation. Further work was also required to reset the system for the subsequent evaluation or combine a set of functions into a directed acyclic graph without an exponential explosion of valves.

The tank system presented in [30], as well as our system, both use saturation arithmetic where the lower bound value is 0 (no water), and the upper bound is the capacity of the result tank (sometimes unbounded). Saturation arithmetic produces more 'natural' results than the usual modular arithmetic used by current computers.

Water tank system. Tanks are displayed as open rectangles, the pipes as lines between the tanks and valves as lines crossing the pipe it belongs to. We denote water going to the infinite sink by a black arrow at the end of the pipe. If the pipe starts with an empty rectangle, then the water comes from the infinite source.

Figure 2.2 and Figure 2.3 illustrate two tank systems that only contain value tanks (and no control tanks).

1. **Subtraction**($\ominus$): Basic saturated subtraction can be achieved using three tanks, input tanks x and y, and the output tank z as shown in Figure 2.2. The system drains from both x and y until y is empty. Once y is empty, x corresponds to the result to be stored in z.
2. **Addition**($\oplus$): Basic saturated addition is achieved using three tanks: the two inputs and the output total as shown in Figure 2.3. The system does not contain valves, and the values of x and y go directly to the result. Once x and y are empty, z contains the final result.

As discussed in [30], although we can intuitively model simple stand-alone gates – such as basic addition and subtraction - several essential open problems remain:

- Termination detection: a tank system that self-determines when it has completed, with a control tank becoming full when the system has finished.
- System reset: a way to re-evaluate the system, possibly on other data.

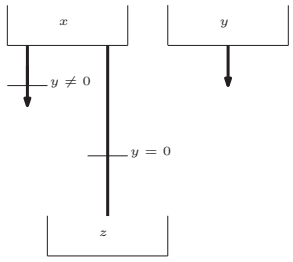


Figure 2.2: A diagram representing basic subtraction $z = x \ominus y$.

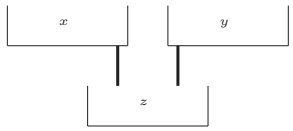


Figure 2.3: A diagram representing basic addition $z = x \oplus y$.

- Simplifying the control valves, to avoid a combinatorial explosion in the number of tanks – essential for building more complex systems.
- Limiting the system structure to be a directed acyclic graph – the previous paper [30] used loops for implementing more complex arithmetic, such as multiplication and division. Where a loop requires a 'pump' to move the water against the natural flow (gravitational gradient).

In Chapter 4 we solve all these problems, by extending the basic approach with a parallel support network of control tanks. It, also proves that this extended water system is Turing complete (by way of μ-recursive functions).

Chapter 3

Efficient single cell cP solutions to hard problems.

In this chapter, we investigate the power of a single cell cP system. In the first part of this chapter, we constructively prove that cP systems can compute PSPACE hard problems in polynomial time. In the second part, we show that cP systems can solve NP-hard problems in sublinear time. With this sublinear solution to K-SAT, we show how reductions can be applied to produce very efficient solutions surpassing the problem-specific solutions presented in the literature.

3.1 Solving PSPACE complete problem

In this section, we demonstrate that cP systems can solve not only 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 10 rules and a constant custom alphabet of size 19. Our solution is *deterministic*, so we do not compare it here with *non-confluent* solutions, such as [37] (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 [33, 38, 24, 2]. 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 3.1, 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. We did not consider the number of repeated copies placed in different membranes/neurons when counting rule templates and rules. If we were to include such occurrence counts,

Table 3.1: 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) [24]	40	$\mathcal{O}(m)$	$\mathcal{O}(nm)$	$\mathcal{O}(n)$
Uniform solution of QSAT (2007) [2]	33	$\mathcal{O}(mn)$	$\mathcal{O}(nm)$	$\mathcal{O}(n)$
Deterministic solution to QSAT (2010) [33]	20	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Solving QSAT in sublinear nesting depth (2018) [38]	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

the number of rules would increase drastically, for the other extant solutions. For example, the solution to QSAT in [33] 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, all these characteristics are small constants.

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

QSAT

Solving QSAT for a given formula ϕ can be done, as shown in [61], with the *recursive* algorithm (pseudocode) presented here in Table 3.2, slightly adapted, where we separate ϕ in three components: q – the stack of *quantifiers*; p – the stack of *variables*; 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.

The candidate solutions of Formulae (2.2, 2.3) can be visualised on the *trees* shown in Figures 3.1, 3.2: (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 $\forall$.

Table 3.2: Recursive algorithm for QSAT: q = quantifiers; p = variables; f = unquantified Boolean expression.

```
1  let rec QSAT q p f =
2    if q = () then
3      eval f // no more quantifiers in prefix, all variables assigned
4
5    else
6      let y, x = pop q, pop p
7      let v' = QSAT q p f[x := 0]
8      let v'' = QSAT q p f[x := 1]
9      let v = if y = ∀ then v' ∧ v'' else v' ∨ v''
10     v
11
12 // sample calls, for Formulae (2.2, 2.3)
13 let v = QSAT (∀, ∃) (x1, x2) (x1 ∨ x2) ∧ (x1 ∨ x̄2) // false
14 let v = QSAT (∃, ∀) (x1, x2) (x1 ∨ x2) ∧ (x1 ∨ x̄2) // true
```

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 the number of tree levels below the root.

Note that Formulae (2.2, 2.3) only differ in quantifiers. Thus Figures 3.1, 3.2 differ only in their quantifiers lists (a) and evaluation results in otherwise isomorphic tress (c); while trees (b) are identical.

The recursion of the algorithm in Table 3.2 can be *unrolled* by straightforward techniques. The *non-recursive* solution in Table 3.3 creates each layer of the tree successively, whilst 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

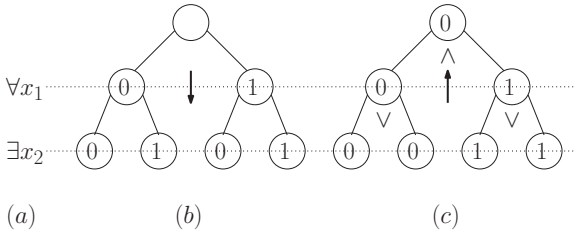


Figure 3.1: QSAT tree for Formula (2.2): $\forall x_1\ \exists x_2\ (x_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2)$.

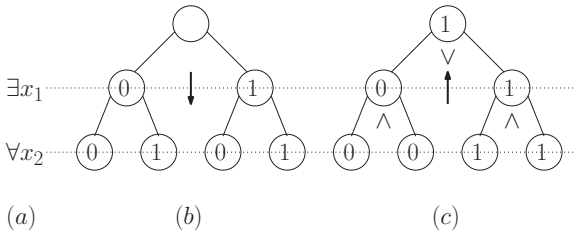


Figure 3.2: QSAT tree for Formula (2.3): $\exists x_1\ \forall x_2\ (x_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2)$.

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; (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, parallel execution of this non-recursive 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 non-recursive solution in Table 3.3.

3.1.1 cP Solution and Examples

In this subsection we discuss our cP system for solving $QSAT$ for $n \geq 1$, and we illustrate its evolution on Formulae (2.2, 2.3), recalled here:

```
1   ∀x1 ∃ x2 (x1 ∨ x2) ∧ (x1 ∨ x̄2)      (1)
2   ∃x1 ∀ x2 (x1 ∨ x2) ∧ (x1 ∨ x̄2)      (2)
```

We only use *one single top-level cell*, and we closely follow the parallel pseudocode algorithm listed in Table 3.3: a layer-by-layer sweep over a virtual tree, in two passes – first top-down, then bottom-up.

We use *six states*, $\{s_1, s_2, ..., s_6\}$, where s_1 is the initial state, and s_6 is the final. The rule set is shown in two listings: the *top-down pass* in Table 3.7, and the *bottom-up pass* in Table 3.9. The evolution corresponding to Formula (2.2) is illustrated in the following tables: Table 3.6 shows the *initial cell contents and state*; then Table 3.8 traces the *top-down* evolution; and Table 3.10 traces the *bottom-up* evolution. Table 3.11 traces the *bottom-up* evolution of the slightly different Formula (2.3).

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 3.4 shows their contents and their interpretation.

Table 3.3: Non-recursive pseudocode – layer-by-layer in both sequential and parallel mode: q = quantifiers; p = variables; F = list of unquantified Boolean expressions.

```
1  let QSAT q p F =
2    let q' = ( )
3    while q ≠ ( ) do // top-down pass
4      let x = pop p
5      push q' (pop q) // reverse q onto q'
6      let F = flatmap F (f → (f[x := 0], f[x := 1]))
7
8    // e.g. F = (0 ∨ 0) ∧ (0 ∨ 0̄), (0 ∨ 1) ∧ (0 ∨ 1̄), (1 ∨ 0) ∧ (1 ∨ 0̄), (1 ∨ 1) ∧ (1 ∨ 1̄)
9
10   let F = map F (f → |f)
11
12   // e.g. F = 0, 0, 1, 1
13
14   while q' ≠ ( ) do // bottom-up pass
15     let y = pop q'
16     let P = pairwise F
17     // e.g. P = (0, 0), (1, 1)
18     let F = map P ((v, v') → if y = ∀ then v' ∧ v'' else v' ∨ v'')
19     // e.g. F = 0, 1
20
21   singleton F  // returned value
22
23 // sample calls, for Formulae (2.2, 2.3)
24 let v = QSAT (∀, ∃) (x1, x2) (x1 ∨ x2) ∧ (x1 ∨ x̄2) // false
25 let v = QSAT (∃, ∀) (x1, x2) (x1 ∨ x2) ∧ (x1 ∨ x̄2) // true
```

Eight sub-cells $w()()()()$ form a lookup table for Boolean $\wedge$ and $\vee$ operations, the actual operation being selected on the corresponding quantifier. Table 3.5 shows their contents and their interpretation.

Prefixes and tree levels

During the top-down pass, sub-cell $\nu()$ is a counter that indicates the tree level depth; initially $\nu(n)$, 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 $\nu()$. During the top-down pass, the top elements of $p()()$ are temporarily popped into $h()$, and $q()()$ is 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 the Table 3.3. Together with their associated counters, $\nu()$ and $\mu()$, these play the role of *global* variables controlling the two passes.

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:

```
1  x_1 ≡ x(1)(+)
2  x̄_2 ≡ x(11)(−)
```

This is just a notation convenience; our rules 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:

```
1  (x_1 ∨ x̄_2) ≡ c(x_1 x̄_2) ≡ c(x(1)(+) x(11)(−))
```

Table 3.4: Cells y form a lookup table for Boolean identity and negation operations: $V =$ **if** $S{=}+$ **then** K **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 3.5: Cells w form a lookup table for Boolean $\vee$ and $\wedge$ operations:
V = **if** Q=$\forall$ **then** $V' \wedge V''$ **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	$w(\exists)(1)(1)(1)$

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 (2.2) is encoded as:

1 $f(c(x_1\ x_2)\ c(x_1\ \bar{x}_2))$

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 to the node on left-most path 01 looks like this:

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

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, although usually is. Thus, the above $a()$'s may indicate the tree node for $x_1 = 0, x_2 = 1$, (Cf. Figures 3.1, 3.2).

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 $v()$ sub-cells. For example, under the above mentioned sample assumptions:

1 $f(v(0)...\ a(11)(1)\ a(1)(0))$
2 $\iff (x_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2)[x_1 := 0, x_2 := 1] \equiv (0 \vee 1) \wedge (0 \vee \bar{1}) \equiv 0$

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 the Table 3.3.

Top-down pass

The rules for the top-down pass are listed in Table 3.7. Essentially, we have a loop consisting of three steps ($s_1 \to s_2 \to s_3 \to s_1$) that is repeated n times and a subsequent one step evaluation ($s_1 \to s_4$). These steps closely follow the top-down pass of the parallel algorithm presented in Table 3.3.

Rules (1, 2, 3) form an **if then else** construct. If we haven't yet processed all quantifiers and variables, condition detected by a non-empty $\nu()$ counter, then rule (1) applies, resetting our global control variables and starting one more loop iteration ($s_1 \to s_2$). 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 $v(0)$ sub-cell. The other formulae, which are **true**, are tagged by one $v(1)$ sub-cell.

Together, rules (4,5,6) form the main body of the top-down loop ($s_2 \to s_3 \to s_1$). 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

Table 3.6: Contents of our top-level cell, as initialised for Formula (2.2): $\forall x_1\ \exists x_2\ (x_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2)$.

1 $s_1\ \ \nu(11)\ \ \mu()$
2 $p(11)(x_1)\ \ p(1)(x_2)\ \ h()$
3 $q(11)(\forall)\ \ q(1)(\exists)$
4 $f(c(x_1\ x_2)\ c(x_1\ \bar{x}_2))$
5
6 $y(0)(+)(0)\ \ y(0)(-)(1)\ \ y(1)(+)(1)\ \ y(1)(-)(0)\ \ //\ \ y\ \ lookup\ table$
7
8 $w(\forall)(0)(0)(0)\ \ w(\forall)(0)(1)(0)\ \ w(\forall)(1)(0)(0)\ \ w(\forall)(1)(1)(1)\ \ //\ \ w\ \ lookup\ table$
9 $w(\exists)(0)(0)(0)\ \ w(\exists)(0)(1)(1)\ \ w(\exists)(1)(0)(1)\ \ w(\exists)(1)(1)(1)$

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 3.8 illustrates this top-down pass by traces for Formula (2.2), starting from the initial state shown in Table 3.6.

Bottom-up evaluation

The rules for the top-down traversal pass are listed in Table 3.9. Essentially, we have a one step transition from the top-down pass ($s_4 \to s_5$), followed by a one step loop ($s_5 \to s_5$) that is repeated n times, and a one step exit to the final state ($s_5 \to s_6$). These steps closely follow the bottom-up pass of the parallel algorithm presented in Table 3.3.

Rule (7) runs in max parallel mode and performs a clean up step ($s_4 \to s_5$), 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, $s_5 \to s_5$, 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 $f()$ are 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 3.3. 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 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, a condition detected by a non-empty counter μ, then rule (9) resumes the loop, $s_5 \to s_5$. Otherwise, rule (10) applies and exits, cleaning all remaining stuff, and recording the final value in $v()$.

Table 3.10 illustrates this bottom-up pass by traces for Formula (2.2, $\forall\ \exists$), starting from the end state shown in Table 3.10. The evolution for the related Formula (2.3, $\exists\ \forall$) are only marginally different, but still significant in the bottom-up pass, when we actually use quantifiers; its bottom-up traces are shown in Table 3.11.

Table 3.7: Top-down rules. The three steps loop $s_1 \to s_2 \to s_3 \to s_1$ is repeated n times, followed by the exit $s_1 \to s_4$.

$s_1\ \nu(1N)\ h(\_)\ p(1N)(X)$ $\quad q(1N)(Q)\ \mu(M)$	$\to_1$	$s_2\ \nu(N)\ h(X)\ \mu(1M)\ q'(1M)(Q)$	(1)
$s_1\ f(c()\ Z)$	$\to_+$	$s_4\ f(v(0)\ Z)$	(2)
$s_1\ f(c(1)\ Z)$	$\to_+$	$s_4\ f(v(1)\ Z)$	(3)
$s_2\ f(Z)$	$\to_+$	$s_3\ f(0)(Z\ a(M)(0))$ $\quad f(1)(Z\ a(M)(1))$ $\mid \mu(M)$	(4)
$s_3\ f(K)\ \{c(x(I)(S)\ Y)\}$	$\to_+$	$s_1\ f\{c(Y)\}$ $\mid h(x(I)(+))$ $\mid y(K)(S)(0)$	(5)
$s_3\ f(\_)\ \{c(x(I)(S)\ Y)\}$	$\to_+$	$s_1\ f\{c(1)\}$ $\mid h(x(I)(+))$ $\mid y(K)(S)(1)$	(6)

Table 3.8: Top-down traces for Formula (2.2, $\forall\ \exists$). Continued from Table 3.6. (Cf. tree (b) in Figure 3.1).

s_1
$\quad \nu(11)\ \ h(\_)\ \ p(11)(x_1)\ \ p(1)(x_2)\ \ q(11)(\forall)\ \ q(1)(\exists)\ \ \mu()$
$\quad\quad f(c(x_1\ x_2)\ c(x_1\ \bar{x}_2))$

$(1) \Rightarrow\ s_2$
$\quad \nu(1)\ \ h(x_1)\ \ p(1)(x_2)\ \ q(1)(\exists)\ \ \mu(1)\ \ q'(1)(\forall)$

$(4) \Rightarrow\ s_3$
$\quad\quad f(0)(c(x_1\ x_2)\ c(x_1\ \bar{x}_2)\ a(1)(0))\ \ f(1)(c(x_1\ x_2)\ c(x_1\ \bar{x}_2)\ a(1)(1))$

$(5) \Rightarrow\ s_1$
$\quad\quad f(c(x_2)\ c(\bar{x}_2)\ a(1)(0))\ \ f(c(1)\ c(1)\ a(1)(1))$

$(1) \Rightarrow\ s_2$
$\quad \nu()\ \ h(x_2)\ \ \mu(11)\ \ q'(11)(\exists)\ \ q'(1)(\forall)$

$(4) \Rightarrow\ s_3$
$\quad\quad f(0)(c(x_2)\ c(\bar{x}_2)\ a(11)(0)\ a(1)(0))\ \ f(1)(c(x_2)\ c(\bar{x}_2)\ a(11)(1)\ a(1)(0))$
$\quad\quad f(0)(c(x_2)\ c(\bar{x}_2)\ a(11)(0)\ a(1)(1))\ \ f(1)(c(x_2)\ c(\bar{x}_2)\ a(11)(1)\ a(1)(1))$

$(5) \Rightarrow\ s_1$
$\quad\quad f(c()\ \ c(1)\ a(11)(0)\ a(1)(0))\ \ f(c(1)\ c()\ \ a(11)(1)\ a(1)(0))$
$\quad\quad f(c(1)\ c(1)\ a(11)(0)\ a(1)(1))\ \ f(c(1)\ c(1)\ a(11)(1)\ a(1)(1))$

$(2,3) \Rightarrow\ s_4$
$\quad\quad f(v(0)\ c(1)\ a(11)(0)\ a(1)(0))\ \ f(v(0)\ c(1)\ a(11)(1)\ a(1)(0))$
$\quad\quad f(v(1)\ c(1)\ a(11)(0)\ a(1)(1))\ \ f(v(1)\ c(1)\ a(11)(1)\ a(1)(1))$

Table 3.9: Bottom-up rules. One step loop $s_5 \to s_5$ repeated n times.

$s_4\ f\{\ c(\_)\ \}$	$\to_+$	$s_5\ f\{\ \}$	(7)
$s_5\ f(v(V')\ a(M)(0)\ A)$ $\quad f(v(V'')\ a(M)(1)\ A)$	$\to_+$	$s_5\ f(v(V)\ A)$ $\mid \mu(M)$ $\mid q'(M)(Q)$ $\mid w(Q)(V')(V'')(V)$	(8)
$s_5\ \mu(1M)\ q(1M)(\_)$	$\to_1$	$s_5\ \mu(M)$	(9)
$s_5\ f(v(V))\ \mu()\ \nu()\ h(\_)$	$\to_1$	$s_6\ v(V)$	(10)

Table 3.10: Bottom-up traces for Formula (2.2, $\forall\,\exists$). Continued from Table 3.8. Final result is **false**. (Cf. tree (c) in Figure 3.1).

1	s_4
2	$\quad \mu(11)\ \ q'(11)(\exists)\ \ q'(1)(\forall)\ \ //\ \ \nu()\ \ h(x_2)$
3	$\quad\quad\quad f(v(0)\ c(1)\ a(11)(0)\ a(1)(0))\ \ f(v(0)\ c(1)\ a(11)(1)\ a(1)(0))$
4	$\quad\quad\quad f(v(1)\ c(1)\ a(11)(0)\ a(1)(1))\ \ f(v(1)\ c(1)\ a(11)(1)\ a(1)(1))$
5	
6	$(7) \Rightarrow\ s_5$
7	$\quad\quad\quad f(v(0)\ a(11)(0)\ a(1)(0))\ \ f(v(0)\ a(11)(1)\ a(1)(0))$
8	$\quad\quad\quad f(v(1)\ a(11)(0)\ a(1)(1))\ \ f(v(1)\ a(11)(1)\ a(1)(1))$
9	
10	$(8,9) \Rightarrow\ s_5$
11	$\quad\quad\quad f(v(0)\ a(1)(0))$
12	$\quad\quad\quad f(v(1)\ a(1)(1))$
13	$\quad \mu(1)\ \ q'(1)(\forall)\ \ //\ \ \nu()\ \ h(x_2)$
14	
15	$(8,9) \Rightarrow\ s_5$
16	$\quad\quad\quad f(v(0))$
17	$\quad //\ \ \mu()\ \ \nu()\ \ h(x_2)$
18	
19	$(10) \Rightarrow\ s_6$
20	$\quad\quad\quad v(0)\ \ //\ \ \mathbf{false}$

Table 3.11: Bottom-up traces for Formula (2.3, $\exists\,\forall$). Continued from Table 3.8, with *different* $q'()()$'s. Final result is **true**. (Cf. tree (c) in Figure 3.2).

1	s_4
2	$\quad \mu(11)\ \ q'(11)(\forall)\ \ q'(1)(\exists)\ \ //\ \ \nu()\ \ h(x_2)$
3	$\quad\quad\quad f(v(0)\ c(1)\ a(11)(0)\ a(1)(0))\ \ f(v(0)\ c(1)\ a(11)(1)\ a(1)(0))$
4	$\quad\quad\quad f(v(1)\ c(1)\ a(11)(0)\ a(1)(1))\ \ f(v(1)\ c(1)\ a(11)(1)\ a(1)(1))$
5	
6	$(7) \Rightarrow\ s_5$
7	$\quad\quad\quad f(v(0)\ a(11)(0)\ a(1)(0))\ \ f(v(0)\ a(11)(1)\ a(1)(0))$
8	$\quad\quad\quad f(v(1)\ a(11)(0)\ a(1)(1))\ \ f(v(1)\ a(11)(1)\ a(1)(1))$
9	
10	$(8,9) \Rightarrow\ s_5$
11	$\quad\quad\quad f(v(0)\ a(1)(0))$
12	$\quad\quad\quad f(v(1)\ a(1)(1))$
13	$\quad \mu(1)\ \ q'(1)(\exists)\ \ //\ \ \nu()\ \ h(x_2)$
14	
15	$(8,9) \Rightarrow\ s_5$
16	$\quad\quad\quad f(v(1))$
17	$\quad //\ \ \mu()\ \ \nu()\ \ h(x_2)$
18	
19	$(10) \Rightarrow\ s_6$
20	$\quad\quad\quad v(1)\ \ //\ \ \mathbf{true}$

Analysis

Proposition 3.1.1. Our solution uses a alphabet size of 25.

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

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

Proposition 3.1.2. Our solution uses a ruleset containing 10 rules.

Top-down pass uses 6 rules (Table 3.7) and bottom-up pass uses 4 (Table 3.9), making a total of 10 rules. □

Proposition 3.1.3. Total runtime is $4n + 3$.

Top-down runtime (Table 3.7): The top-down loop $s_1 \to s_2 \to s_3 \to s_1$ runs n times. The transition $s_1 \to s_4$ runs once, making this pass take $3n + 1$ steps.

Bottom-up runtime (Table 3.9): The transitions $s_4 \to s_5$ and $s_5 \to s_6$ run once. The bottom-up loop $s_5 \to s_5$ runs n times, making this pass take $n + 2$ steps.

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

Proposition 3.1.4. The evolution of our ruleset is totally deterministic.

Rules applicable exactly once ($\to_1$) use singleton terms and do not allow any possible choice. Rules applicable in the max parallel mode ($\to_+$) make the same multiset transformations, regardless of any hypothetical application order. □

Proposition 3.1.5. The maximum membrane nesting depth is 6.

The largest nesting depth in Table 3.8 occurs in:

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

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. □

Theorem 3.1.6. *Single cell cP systems can deterministically solve QSAT in time $\mathcal{O}(n)$ with 10 rules and an alphabet size of 25.*

The proof for this is contained in propositions 3.1.1-3.1.5.

3.1.2 Conclusions

We have presented an efficient *deterministic* cP solution to QSAT, that runs in $4n + 3 = \mathcal{O}(n)$ steps, the 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.

3.2 Sublinear solutions to NP complete problems

In this section we demonstrate that cP systems can solve k-SAT in sublinear time $(\sqrt{n})$. We also show that a polynomial reduction from k-colouring to k-SAT can be done in a constant number of steps. This shows that our reduced solution as far as we know is the fastest P system solution for k-colouring.

3.2.1 Rule set

Here we assume k-SAT to be a formula in CNF with variables $x_0, ..., x_{n-1}$ where each clause contains at most k variables. To solve k-SAT in square root time, we break it up into three steps: first step is generating assignment templates, second is generating the assignments, and finally evaluating the entire formula. Figure 3.3 shows a state diagram of the entire system broken down into the three main parts.

Initial configuration

During the execution of the algorithm subcell $m()$ is used to determine when loops have finished; initially set to $m(1)$. Subcell $j()$ is used to store the *branch number* of allocations of variables where, the *branch number* is a index of the paths from root to leaf starting at 0 for the left-most leaf (see Figure 3.4b); initially we have $j(0)$ $j(1)$ $j(2)$ $j(3)$ because we assume that we have already allocated x_0 and have the next level of branches ready to assign.(no matter what the value of n we assume that we have only allocated the first variable).

Another way of looking at branch numbers is a bijection between integers and allocations. Algorithms to go between these representations are given in the appendix.

Subcell $a(i)(v)(j)$ states that x_i has been assigned the value v and is of the allocation for branch number j. Subcells a initially have x_0 assigned ($a(0)(0)(0)$ $a(0)(1)(1)$).

As described in [26] we can simplify the rules to evaluate a variable using lookup tables. The lookup table y has three parts: the assigned value of the variable; the 'sign' of the variable (whether negated or not) in a clause; and the value when you apply this sign to the value of the assigned variable.

The subcell $k()$ contains the value k for k-SAT. The subcell l contains $\sqrt{n}$ where n is the number of variables. The formula that is being tested is encoded as subcells $c(x(i)(s), ...)$, where i denotes which x_i is being referred to and s whether it is negated in the clause.

For example, consider the following formula ($n = 4$, $k = 2$, and $\rho = \sqrt{n} = 2$):

$$(x_0 \vee x_1) \wedge (x_2 \vee \bar{x_3}) \wedge (\bar{x_2} \vee x_1). \tag{3.1}$$

Table 3.12 contains the fixed values, and Figure 3.4a the variables that change during the evolution of the system.

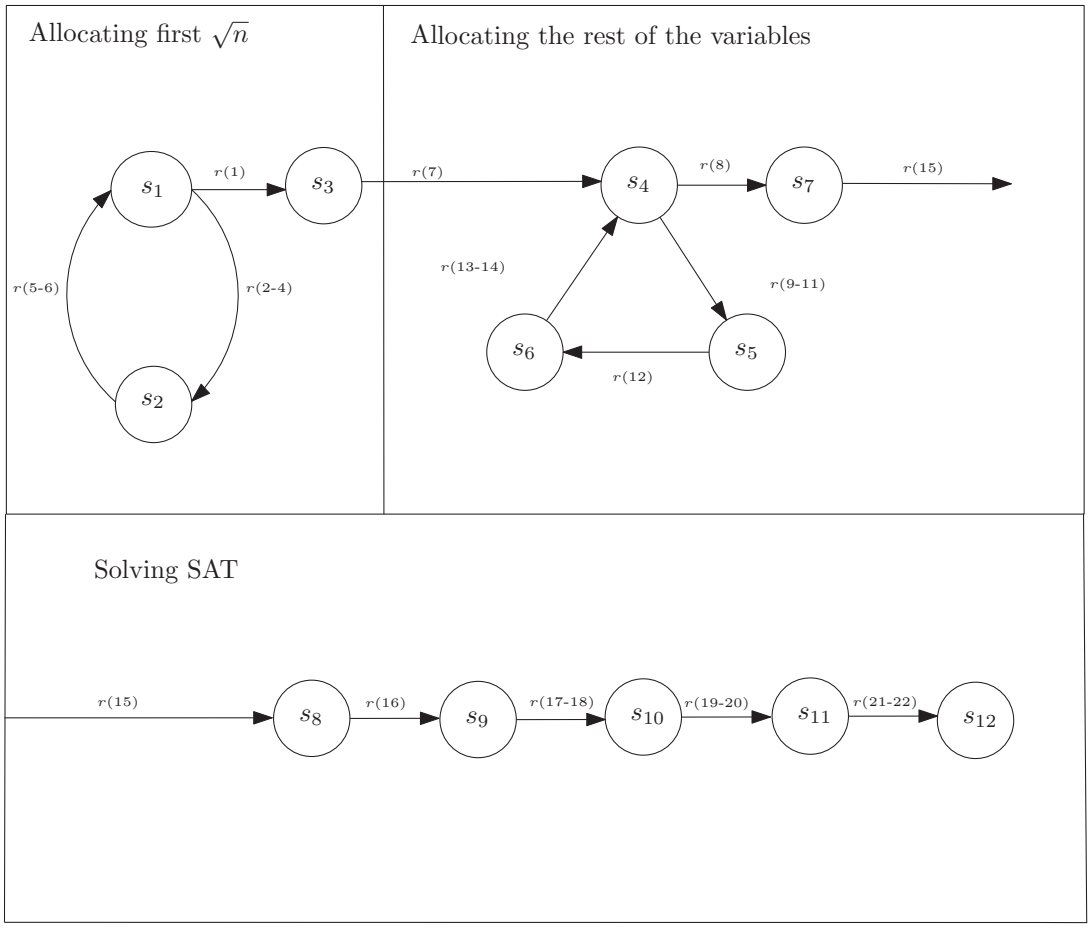


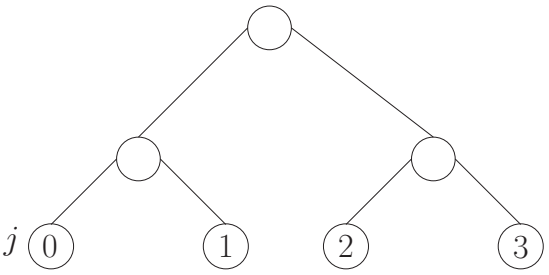
Figure 3.3: State diagram broken up into the three parts of the algorithm.

Table 3.12: Initial state of the variables that *do not* change for Formula 3.1

Table representation			cP system representation
y			
0	+	0	$y(0)(+)(0)$
1	+	1	$y(1)(+)(1)$
0	−	1	$y(0)(-)(1)$
1	−	0	$y(1)(-)(0)$
c			
x_0	x_1		$c(x(0)(+)\ x(1)(+))$
x_2	$\bar{x_3}$		$c(x(2)(+)\ x(3)(-))$
$\bar{x_2}$	x_1		$c(x(2)(-)\ x(1)(+))$
			$k(2)$
			$l(2)$

i	x_i	j	cP representation
0	0	0	$a(0)(0)(0)$
0	1	1	$a(0)(1)(1)$
			$m(1)$
			$j(0)\ j(1)\ j(2)\ j(3)$

(a) Table representation.



(b) Graph representation

Figure 3.4: cP (a table) and binary tree representation of initial branch numbers.

Allocating first $\sqrt{n}$ variables

First, we allocate the first $\sqrt{n}$ variables, which is then used as a lookup table. A traditional programming approach to creating these allocations can be seen in Table 3.13. The outer loop is used to reference the next variable being assigned and the first inner loop creating two new allocations from the previous ones. The second inner loop allocates the next variable for these newly created allocations (a 0 if the branch is even and a 1 if odd).

Our cP system closely models that of the sequential algorithm presented in Table 3.13 with the ruleset presented in Table 3.14. Rules 1 and 6 form the outer loop, with rule 6 being the increment and rule 1 being the termination condition. Rule 2 creates the copies and changes their branch numbers. Rules 3 and 4 add the next variable to the allocations. The outer loop formed by rules 1 and 6 run $\sqrt{n}$ times. The inner loop runs in parallel for all allocations at once. Rules 2-4 run in parallel, taking 1 step total for each loop. Rules 5 and 6 also run in parallel making the total running time $2\sqrt{n}+1$ alternatively $O(\sqrt{n})$.

As an example, the values in Figure 3.4a will change to the values in Table 3.15 after the execution of rules 2-6.

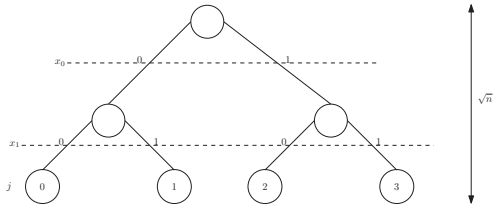
```
1  a = {(0,0,0),(0,1,1)}
2  j = {0,1,2,3}
3  for m = 1; m < √n; m++
4      p = {}
5      for (i,v,j) in a
6          p = p ∪ (i,v,j*2) ∪ (i,v,j*2+1)
7      t = {}
8      for z in j
9          if z%2 == 0:
10             p = p ∪ (m,0,z)
11         else:
12             p = p ∪ (m,1,z)
13         t = t ∪ 2*z ∪ 2*z+1
14     a = p
15     j = t
```

Table 3.13: Sequential algorithm for creating first $\sqrt{n}$ allocations.

Table 3.14: Rules to allocate first $\sqrt{n}$ variables.

$s_1\ m(I)$	$\rightarrow_1$	$s_3\ m(I)$ $\mid l(I)$	(1)
$s_1\ a(X)(Y)(Z)$	$\rightarrow_+$	$s_2\ a(X)(Y)(ZZ)$ $a(X)(Y)(ZZ1)$	(2)
$s_1\ j(ZZ)$	$\rightarrow_+$	$s_2\ j(ZZ)\ a(Y)(0)(ZZ)$ $\mid m(Y)$	(3)
$s_1\ j(ZZ1)$	$\rightarrow_+$	$s_2\ j(ZZ1)\ a(Y)(1)(ZZ1)$ $\mid m(Y)$	(4)
$s_2\ j(Z)$	$\rightarrow_+$	$s_1\ j(ZZ)$ $j(ZZ1)$	(5)
$s_2\ m(I)$	$\rightarrow_1$	$s_1\ m(I1)$	(6)

Table 3.15: First step of the algorithm for solving Formula 3.1.

Table representation				Graph representation
i	x_i	j	cP system representation	
0	0	0	$a(0)(0)(0)$	
1	0	0	$a(1)(0)(0)$	
0	0	1	$a(0)(0)(1)$	
1	1	1	$a(1)(1)(1)$	
0	1	2	$a(0)(1)(2)$	
1	0	2	$a(1)(0)(2)$	
0	1	3	$a(0)(1)(3)$	
1	1	3	$a(1)(1)(3)$	
			$m(2)$	
			$j(0)$ $j(1)$ $j(2)$ $j(3)$	
			$j(4)$ $j(5)$ $j(6)$ $j(7)$	

Allocating all other variables

To allocate the rest of the variables, we use the templates that were previously created for the first $\sqrt{n}$ variables. Using the templates, we loop $\sqrt{n}$ times, where on each loop we do a Cartesian product between the previously allocated variables and the template (the templates variables get incremented by $\sqrt{n}$ before each Cartesian product). Alternatively, this operation can be viewed as taking the allocation tree in Table 3.15, copying it and placing the tree at all of the leaves, as shown in Figure 3.5.

A sequential version of this algorithm can be seen in Table 3.16. First, we make a copy of the template a with all variables incremented by $\sqrt{n}$. Then we do an outer loop from $\sqrt{n}$ to n incrementing by $\sqrt{n}$. Inside this loop, a Cartesian product is made looping over each allocation in b and in a. The branch number for the combined allocation is denoted recursively as $\alpha(j)(i)$, with j being the branch number from a, and i being the branch number from b. For example, for 9 variables, one of the branch numbers created is $\alpha(\alpha(0)(1))(2)$.

The rules to allocate the remaining variables are in Table 3.17. The state diagram giving the state transitions of allocating the variables is shown in Figure 3.3.

Rule 7 acts as allocating the original b value. Rules 8 and 11 form the outer loop presented in the sequential algorithm, with rule 8 the termination condition and rule 11 the increment. Rules 9 and 10 apply the Cartesian product. Rule 13 increments b (last line of the sequential algorithm), and rules 12 and 14 reassign a (line 15 of the sequential algorithm).

The rules 9-14 form a loop which runs $\sqrt{n}$ times, with rules 9-11 running in parallel as well as 13 and 14. The loop takes $3\sqrt{n}$ steps and rules 7 and 8 each take one step,

```
1  a = {(0, 0, 0), ...}
2  b = map (i, v, j) => (i + √n, v, j), a
3  p = {}
4  for m = √n; m < n; m += √n
5      d = {}
6      for (i, v, j) in a:
7          for (y, q, x) in b
8              if i + m == y
9                  d = d ∪ (i, v, (j, x))
10                 d = d ∪ (y, v, (j, x))
11             for z in p
12                 if i + z == y
13                     d = d ∪ (i, v, (j, x))
14     p = p ∪ m
15     a = d
16     b = map (i, v, j) => (i + √n, v, j), b
```

Table 3.16: Sequential algorithm for creating the rest of the allocations

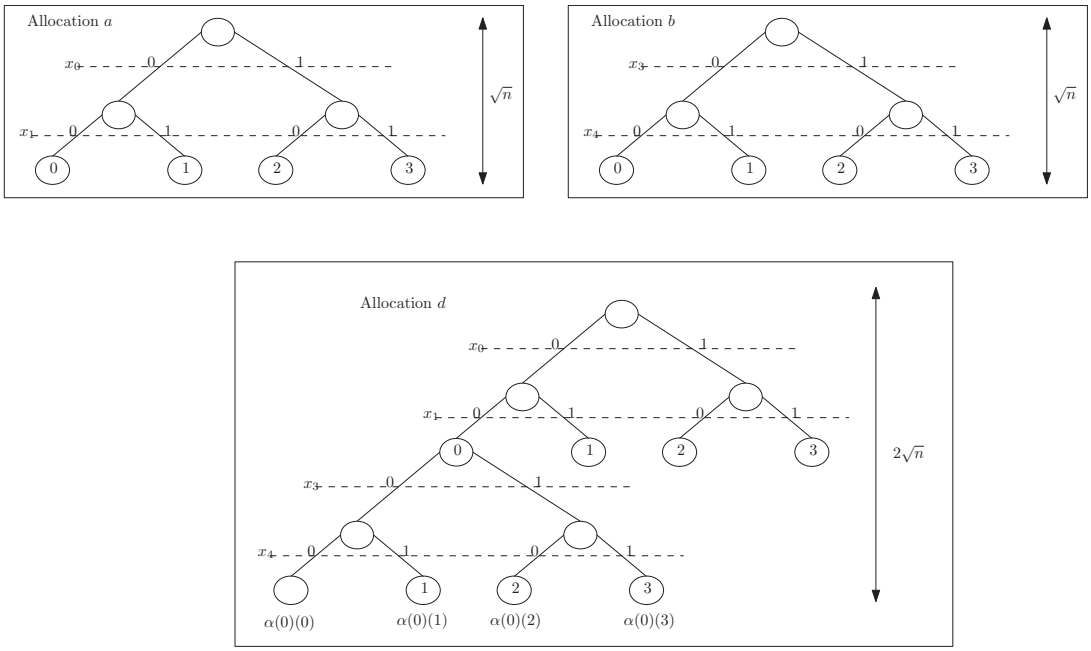


Figure 3.5: Diagram representing the Cartesian product showing for $\alpha = 0$ and $n = 4$.

making the running time for the rules presented in Table 3.17 $3\sqrt{n} + 2$ steps.

For example, rule 7 will create the b subcells displayed in Table 3.17. Where we note the only difference between the a and b is the first parameter with b subcells have $\sqrt{n}$ added. Rules 9 and 10 will create subcells d, which, as seen in Table 3.18 can be viewed as taking a Cartesian product of a and b. The d cells will become the a cells and loop until all allocations have been completed.

Table 3.17: Rules to create allocations

$s_3\ a(X)(Y)(Z)$	$\rightarrow_+$	$s_4\ a(X)(Y)(Z)$ $\quad b(XQ)(Y)(Z)$ $\mid l(Q)$ $\mid n(XQI)$	(7)
$s_4\ m(QX)$	$\rightarrow_1$	$s_7\ m(1)$ $\mid n(Q)$	(8)
s_4	$\rightarrow_+$	$s_5\ d(X)(Y)(\alpha(Z)(W))$ $\quad d(XP)(V)(\alpha(Z)(W))$ $\mid a(X)(Y)(Z)$ $\mid b(XP)(V)(W)$ $\mid m(P)$	(9)
s_4	$\rightarrow_+$	$s_5\ d(X)(Y)(\alpha(Z)(W))$ $\mid a(X)(Y)(Z)$ $\mid b(XP)(V)(W)$ $\mid p(P)$	(9)
$s_4\ m(I)$	$\rightarrow_1$	$s_5\ p(I)\ m(IQ)$ $\mid l(Q)$	(11)
$s_5\ a(X)$	$\rightarrow_+$	s_6	(12)
$s_6\ b(X)(Y)(Z)$	$\rightarrow_+$	$s_4\ b(XQ)(Y)(Z)$ $\mid L(Q)$ $\mid n(XQI)$	(13)
$s_6\ d(X)$	$\rightarrow_+$	$s_4\ a(X)$	(14)

Table 3.18: Creating the next $\sqrt{n}$ variables for Formula 3.1

a

i	x_i	j	cP representation
0	0	0	$a(0)(0)(0)$
1	0	0	$a(1)(0)(0)$
0	0	1	$a(0)(0)(1)$
1	1	1	$a(1)(1)(1)$
0	1	2	$a(0)(1)(2)$
1	0	2	$a(1)(0)(2)$
0	1	3	$a(0)(1)(3)$
1	1	3	$a(1)(1)(3)$

b

i	x_i	j	cP representation
2	0	0	$b(2)(0)(0)$
3	0	0	$b(3)(0)(0)$
2	0	1	$b(2)(0)(1)$
3	1	1	$b(3)(1)(1)$
2	1	2	$b(2)(1)(2)$
3	0	2	$b(3)(0)(2)$
2	1	3	$b(2)(1)(3)$
3	1	3	$b(3)(1)(3)$

d

i	x_i	j	cP representaion
0	0	$\alpha(0)(0)$	$d(0)(0)(\alpha(0)(0))$
1	0	$\alpha(0)(0)$	$d(1)(0)(\alpha(0)(0))$
2	0	$\alpha(0)(0)$	$d(2)(0)(\alpha(0)(0))$
3	0	$\alpha(0)(0)$	$d(3)(0)(\alpha(0)(0))$

Solving SAT

The rules discussed previously are just a way of allocating all of the variables. For each allocation, the formula is evaluated and checked whether an allocation exists that satisfies the formula. A sequential algorithm describing the steps taken by our cP system can be found in Table 3.19. A state diagram of this final part of the algorithm is shown in Figure 3.3.

The rule set presented in Table 3.20 is the cP system equivalent of the algorithm presented in Table 3.19. Rules 15 and 16 are used to copy the formula for each of the different allocations. Rule 17 checks if any of the allocated variables makes the clause true if none exist, rule 18 sets the clause to false (they apply the **or** operation $\vee$). Rules 19 and 20 apply the and operation ($\wedge$) between the clauses. Rules 21 and 22 determine if there exists a satisfying assignment, outputting $r(1)$ if one existed, and $r(0)$ otherwise.

Rules 15 and 16 run once and are independent of each other(2 steps). The pairs of rules (17, 18) (19, 20), and (21, 22) each run once, with each pair taking 1 step (total is 3 steps). Making the running time of the rule set presented in Table 3.20 5 steps.

For example, if we have the allocation displayed in Table 3.21 rule 16 will create a clause for each of the allocations resulting in subcells denoted κ with a branch number as seen in Table 3.22. Once created, these clauses are evaluated, using rules

```
1  a = {(0, 0, (α(...)(0))), ...}
2  c = {((i, s), (j, t), ...), ...}
3  // a set of k tuples with each k tuple item being a pair.
4  γ = {}
5  for (i, v, j) in a
6      γ = γ ∪ j
7  κ = {}
8  for d in c
9      for j in γ
10         κ = κ ∪ (d, j)
11         //(d, j)(((i, s), (j, t), ...), (α(...)(0)))
12 t = {}
13 for (d, j) in κ
14     p = 0
15     for (i, s) in d
16         for (x, v, y) in a
17             if x == i and j == y and y(v, s) == 1
18                 p = 1
19     t = t ∪ (p, j)
20 κ = t
21 f = {}
22 for y in γ
23     v = 1
24     for (p, j) in κ
25         if p == 0 and j = y
26             v = 0
27     f = f ∪ v
28 r = 0
29 for v in f
30     if v == 1
31         r = 1
```

Table 3.19: Sequential algorithm for solving sat given all the allocations

Table 3.20: Rules to solve using the allocations.

$s_7\ a(0)(Y)(Z)$	$\rightarrow_+$	$s_8\ a(0)(Y)(Z)\ \gamma(Z)$	(15)
s_8	$\rightarrow_+$	$s_9\ \kappa(X\ j(Y))$ $\mid c(X)$ $\mid \gamma(Y)$	(16)
$s_9\ \kappa(x(I)(S)\ \_\ j(Y))$	$\rightarrow_+$	$s_{10}\ \kappa(1)(Y)$ $\mid a(I)(V)(Y)$ $\mid y(V)(S)(1)$	(17)
$s_9\ \kappa(\_\ j(Y))$	$\rightarrow_+$	$s_{10}\ \kappa(0)(Y)$	(18)
$s_{10}\ \kappa(0)(Y)\ \gamma(Y)$	$\rightarrow_+$	$s_{11}\ f(0)(Y)$	(19)
$s_{10}\ \kappa(1)(Y)\ \gamma(Y)$	$\rightarrow_+$	$s_{11}\ f(1)(Y)$	(20)
$s_{11}\ f(1)(\_)\ m(1)$	$\rightarrow_+$	$s_{12}\ r(1)$	(21)
$s_{11}\ f(0)(\_)\ m(1)$	$\rightarrow_+$	$s_{12}\ r(0)$	(22)

17 and 18 as shown in Table 3.23. After the evaluation the clauses with matching j are combined using an **and** operation $\wedge$ (rules 22 and 23) as shown in Table 3.24. Finally, the system checks if there is any f subcell containing a one; if there is, then there exists a satisfying assignment.

Theorem 3.2.1. *k-SAT is solvable in $O(\sqrt{n})$ using 22 rules and alphabet of size 26.*

The rule sets presented in Tables 3.14, 3.17 and 3.20 solve k-SAT with the running times being $\sqrt{n}+1$, $3\sqrt{n}+2$ and 5 making the total time $4\sqrt{n}+8$ therefore, $O(\sqrt{n})$. The alphabet symbols are $s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{11}, s_{12}, \kappa, \gamma, y, n, f, r, m, a, b, j, d, \alpha, l, p$

Table 3.21: Allocating variables for solving Formula 3.1 where we only list $\alpha(0)$ and $\beta(0)$ for brevity.

Table representation			cP system representation
i	x_i	j	
0	0	$\alpha(0)(0)$	$a(0)(0)(\alpha(0)(0))$
1	0	$\alpha(0)(0)$	$a(1)(0)(\alpha(0)(0))$
2	0	$\alpha(0)(0)$	$a(2)(0)(\alpha(0)(0))$
3	0	$\alpha(0)(0)$	$a(3)(0)(\alpha(0)(0))$

Table 3.22: The clauses for each allocation of Formula 3.1 where we only list $\alpha(0)$ and $\beta(0)$ for brevity.

Table representation			cP system representation
$x:i,s$	$x:i,s$	j	
$0,+$	$1,+$	$\alpha(0)\ \beta(0)$	$\kappa(x(0)(+)\ x(1)(+)\ j(\alpha(0)(0)))$
$2,+$	$3,-$	$\alpha(0)\ \beta(0)$	$\kappa(x(0)(+)\ x(1)(-)\ j(\alpha(0)(0)))$
$2,-$	$1,+$	$\alpha(0)\ \beta(0)$	$\kappa(x(0)(-)\ x(1)(+)\ j(\alpha(0)(0)))$

Table 3.23: The clause with all variables assigned for Formula 3.1 where we only list $\alpha(0)$ and $\beta(0)$ for brevity.

Table representation		cP system representation
v	j	
0	$\alpha(0)(0)$	$\kappa(0)(\alpha(0)(0))$
1	$\alpha(0)(0)$	$\kappa(1)(\alpha(0)(0))$
1	$\alpha(0)(0)$	$\kappa(1)(\alpha(0)(0))$

Table 3.24: The clause with all variables assigned for Formula 3.1 where we only list $\alpha(0)$ and $\beta(0)$ for brevity.

Table representation		cP system representation
v	j	
0	$\alpha(0)\ \beta(0)$	$f(0)(\alpha(0)(0))$

3.2.2 cP reductions for k-colouring

To make reductions simpler we first demonstrate how to use cP rules to make the formulas *at_most_one* and *at_least_one* following the encoding we used for our solution to k-SAT. Assuming we are given $x(0), ..., x(i)$ and a number i we make the formulae using the rule :

$$\begin{array}{lll} s_1 & \rightarrow_+ & s_2\ c(x(X)(-)\ x(XY1)(-)) \\ & & \mid x(X) \\ & & \mid x(XY1) \end{array} \quad (1)$$

At least one requires a loop in which we create the clause:

$$\begin{array}{lll} s_1\ i(0) & \rightarrow_+ & s_2 & (1) \\ s_1\ c(Y) & \rightarrow_+ & s_1\ c(Y\ x(X)(+)) & (2) \\ & & \mid x(X) & \\ & & \mid i(X) & \\ s_1\ i(X) & \rightarrow_+ & s_2\ i(1X) & (3) \end{array}$$

k-colouring

As discussed in subsection 2, the k-colouring problem is given a graph G determine if we can assign one of the k colours to each vertex such that no neighbours have the same colour. We saw that this can be solved using the formula:

$$\begin{aligned} F = & \bigwedge_{v \in V} exactly\_one(x_{v,i} : i \in \kappa) \\ & \wedge \bigwedge_{\{u,v\} \in E} \bigwedge_{i \in \kappa} at\_most\_one(x_{u,i}, x_{v,i}) \end{aligned} \quad (3.2)$$

cP encoding To encode the problem k-colouring we shall use the following:

- Vertices of the graph: $v_1, ..., v_n$ encoded as $v(1)...v(n)$
- Edges of the graph: $e_{\alpha,\beta}, ..., e_{i,j}$ encoded as $e(\alpha)(\beta)...e(i)(j)$
- The number k encoded as $\kappa(k)$
- The number n encoded as $\eta(n)$
- $\sqrt{n}$ encoded as $l(\sqrt{n})$

To construct the group of new variables $x_0, ..., x_{k(n-1)}$ we use the following rules:

$$
\begin{array}{lll}
s_1 & \to_1 & s_2 \mid i(X)\ k(X) \qquad (1)\\
s_1\ j(X)\ i(Y) & \to_+ & s_1\ j(NX)\ m(X)\ i(1Y) \mid \eta(N) \qquad (2)\\
s_2\ v(I) & \to_+ & s_3\ v(I)\ x(IX) \mid m(X) \qquad (3)
\end{array}
$$

The rules are a loop where on the ith iteration, a set of variables is created which encodes the vertices with the ith colour. To encode the ith vertex with the jth colour it is encoded as $x_{n\times j+i}$. The running time is $k+2$ as it loops k times on rule 2 and runs rules 1 and 3 once.

To ensure that the colours of vertices joined by an edge are not the same the following rules are used(creating a *at_most_one*):

$$
\begin{array}{lll}
s_3 & \to_+ & s_4\ c(x(XY)(-)\ x(XY1Z)s(-)) \qquad (4)\\
 & & \mid e(X)(X1Z)\\
 & & \mid m(Y)
\end{array}
$$

The rule uses the m to denote the gap between the different colours (multiples of n) and constructs the clauses such that at most, one of the variables in an edge contain that colour. The running time is 1 step.

The rules to ensure that exactly one colour is chosen for each vertex can be broken into two steps. The first being that each vertex must take at most one of the colours which is given by:

$$
\begin{array}{lll}
s_3 & \to_+ & s_4\ c((XY)(-)\ x(XY1Z)s(-)) \qquad (5)\\
 & & \mid m(Y)\\
 & & \mid m(1YZ)
\end{array}
$$

This rule is practically the same as that given for at most one edge being the same colour. In fact the two rules can work in parallel so this has running time 1 (when totalling rules 4 and 5 is total 1 step). The second step being that at least one colour is taken by each vertex which is given by:

$$s_3\ v(X)\ x(X)\ i(1Y) \quad \rightarrow_+ \quad s_4\ c(x(X)s(+))\ i(Y) \tag{6}$$

$$s_4\ i(0) \quad \rightarrow_+ \quad s_5 \tag{7}$$

$$s_4\ c(Y\ x(X)(+))\ x(XZ) \quad \rightarrow_+ \quad s_4\ c(Y\ x(X)(+)\ x(XZ)(+)) \mid m(Z) \tag{8}$$

$$s_4\ i(1Y) \quad \rightarrow_+ \quad s_4\ i(Y) \tag{9}$$

These rules work in a loop over the colours. Whereat the ith iteration, the ith colour is added to the clause. The looping rules, 8 and 9, run k times (they run in parallel with each other) and rules 6 and 7 once. Hence, the time taken is $k + 2$.

Theorem 3.2.2. *k-colouring $\leq_p$ k-SAT in constant time.*

As demonstrated, our rules to change an instance of k-colouring to k-SAT took a time of $2k + 5$. Due to k being a fixed constant and not part of the problems input.

Corollary 3.2.3. *3-colouring is solvable in $O(\sqrt{n})$ steps.*

As 3-colouring is the instance of k-colouring for $k = 3$. We know the Karp reduction from 3-colouring takes 11 steps and solving the instance of 3-SAT takes $4\sqrt{n}+8$ steps hence the total number of steps is $O(\sqrt{n})$.

3.2.3 Discussion

SAT is one of the most famous problems to be known to be NP-complete, with many study's using theoretical molecular computing devices to solve it [42]. As discussed in [46], many solutions to SAT have been found running in linear time using P systems. A previous solution using cP systems also was found running in linear time [52]. However, as far as we know, our solution is the first P system solution to run in sublinear time.

We note that as discussed in [52], many of these solutions use a variable number of rules and alphabet symbols. Our solution uses a *constant* sized alphabet and ruleset. We do, however, note that our solution uses more rules than presented in [52].

As with SAT the 3-colouring problem has been the subject of many studies using P system variants including: cP systems [13], tissue P systems [15, 67], and kernel P systems [67] . However, as far as we know, no other solution using P systems runs in sublinear time.

3.2.4 Conclusions

We have presented a sublinear solution to the k-SAT and k-colouring problems and, as far as we know, the most efficient P system solution. Our solution to 3-colouring demonstrates that at least some of the traditional polynomial reductions can be done in a constant number of steps using cP systems. We also note that the strategies used to generate our allocations can be utilised to extend our solution to k-SAT to solving QSAT (a PSPACE complete problem).

3.3 Conclusions and Future Work

In this chapter we have shown how cP systems are able to solve a PSPACE hard problem. As shown in Table 3.1 our solution compared to other extant confluent P systems solutions, 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.

We then showed how we can develop efficient solutions to NP complete problems. With our solution, as far as we know, the most efficient P system solution to both k-SAT and k-colouring. Our solution used a fixed sized alphabet and fixed number of rules. We showed that at least the reduction from k-colouring to k-SAT can be done in a constant number of steps.

Future work includes model checking these solutions. We note that although cP systems have been used for model checking in the past [39] they have had the issue of memory explosion. However, if we are just model checking a reduction this should not occur and may enable much larger instances to be model checked.

Another problem is how many other problems can be efficiently reduced. The overarching problem being, can we find a significantly more efficient reduction using cP systems than the traditional Turing machine reduction.

Chapter 4

Computing with water

In this chapter we further develop water computing as a variant of P systems. We propose an improved modular design, which duplicates the main water flows by associated control flows. We first solve the three open problems of the previous design by demonstrating: how functions can be stacked without a combinatorial explosion of valves, how termination of the system can be detected and how to reset the system. We then prove that the system is Turing complete by modelling the construction of μ-recursive functions. The new system is based on directed acyclic graphs, where tanks are nodes and pipes are arcs; there are no loops anymore, water falls strictly in a 'top down' direction. We then demonstrate how our water tank system can be viewed as a restricted version of cP systems. Finally, we demonstrate that such systems can construct 'efficiently': 1) A programmable sequential, random-access machine (RAM), which we then extend to construct: 2) a programmable exclusive read exclusive write (EREW) parallel random-access machine (PRAM).

4.1 Turing completeness of water computing

In this section, we present an alternative definition for the water tank system based on rules, which more closely aligns with cP systems [50] (but also other P system variants). Our definition removes the timed termination of the previous model by replacing it with a set of control tanks that tightly control the execution of the actual operation. Our model also removes pipes having different flows as we assume that each pipe will move one unit of water at each time step. Our model specifies a separate *infinite source* and *infinite sink*. Together the source and sink form a two-compartment 'environment': a read-only source and write only sink. However, joining them takes away from the water moving in one direction, which is likely important for physical realisations, where our source is inspired by the beauty of naturally occurring waterfalls.

Our system has two types of tanks: value tanks, which correspond to the tanks

presented in [30]; and control tanks, which are essentially Boolean values. Each input and output value tank of the system has a corresponding control tank. If a control tank is full, then the corresponding value tank has been filled. This means that an operation has terminated if and only if all of its output control tanks are full. Value tanks have capacities in $\mathbb{N}_+ \cup \{\infty\}$: infinite ($\infty$) for unbounded tanks, and finite ($\mathbb{N}_+$) for bounded tanks. We assume that all bounded tanks have overflow pipes, which cleanly drain any possible overflow down to an infinite sink; but, while still there, overflow pipes are not represented in our diagrams (to keep these simple). Most theoretical results, such as μ-recursivity, assume that all value tanks are unbounded (similar to how a RAM has an unbounded number of registers); while, obviously, all practical implementations need bounded tanks.

In subsection 2, we briefly describe the basics of water computing systems. In subsection 3, we describe how a control tank can be constructed and how to reset a system. In subsection 4, we define formally our new model. In subsection 5, we prove that the system is Turing complete and that the tank system can be viewed as a directed acyclic graph. In subsection 6, we describe how the valves are viewed as a set of membrane computing rules and establish the water tank system as a restricted version of cP systems.

4.1.1 Modularisation and control tanks

In this subsection, we describe the use of control tanks for termination detection. We also discuss how these allow for modularisation and composition of functions.

Constructing complex expression such as $h(x, y, u, v) = (x \oplus y) \ominus (u \ominus v)$, from the functions in Figures 2.2 and 2.3 requires additional valves to be added. If one combines these without the addition of valves, then the result may be unexpected because of the synchronisation requirements of the subtraction operator. To overcome this issue, we use modularisation to allow for easier composition of operations.

Thus, the following restrictions are placed on all functions. A function $f(x_1, ..., x_n) = (y_1, \ldots, y_m)$ has n and m accompanying controls on the inputs $(x'_1, \ldots, x'_n)$ and outputs $(y'_1, \ldots, y'_m)$ respectively . The function executes when all input control tanks are filled. Each output of the function has finished being computed, when its corresponding control tank has been filled. For example, a function f with n inputs and m outputs is presented in Figure 4.1. We note that inner control tanks initially are empty, for example, q' in Figure 4.2.

Saturating subtraction following these conventions becomes the system presented in Figure 4.2. Although the system looks more complicated, it is much easier to *combine* the system into a complex system. Any function relying on the results from another function only needs a relationship with its result and control tanks. Thus a function can be viewed as a black box, cf. Figure 4.1, where the inputs are passed to the function and the outputs are a control and result tank. The inner workings can be ignored as before, and after execution, they are the same. The Appendix presents

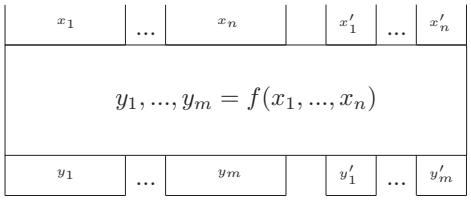


Figure 4.1: A diagram representing a function f following the conventions $y_1, ..., y_m = f(x_1, ..., x_n)$.

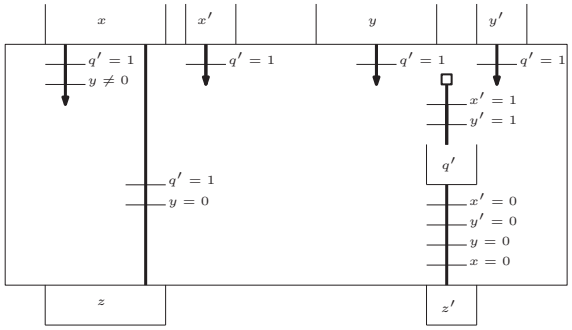


Figure 4.2: A diagram representing the controlled saturating subtraction operator $z = x \ominus y$.

an example trace of the controlled subtraction via images and cell contents of a cP representation.

Complex expressions can be built using a high level design. For example, if we have the addition function $f(x, y) = x \oplus y$ as seen in Figure 4.3, then we can create the complex expression $h(x, y, u, v) = (x \oplus y) \ominus (u \ominus v)$ by simply using the outputs and output controls as the inputs to the subtraction, as seen in Figure 4.4.

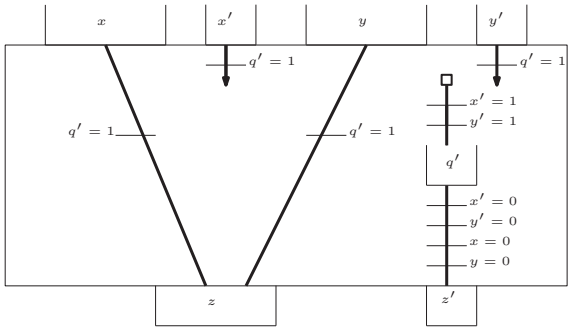


Figure 4.3: A diagram representing the controlled saturating addition operator $z = x \oplus y$.

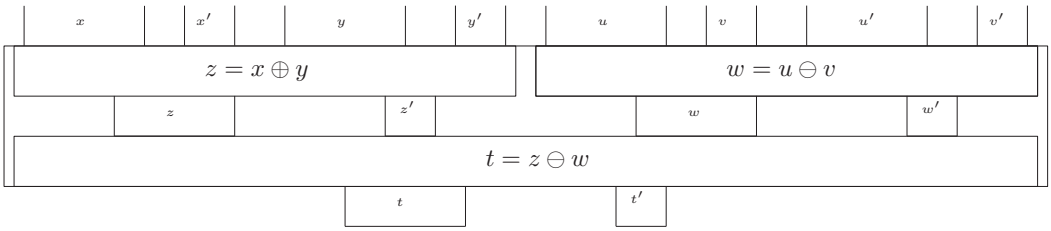


Figure 4.4: A diagram showing how complex expressions can be created.
$t = (x \oplus y) \ominus (u \ominus v)$.

4.1.2 New model

With the addition of control tanks, we formalise our model based on the work presented in [30]. Our water-based system:

$$\Pi = (T, T', F, e, r, L, C, V, S, P)$$

With its components:

- T finite set of tank identifiers.
- $T' \subset T$ finite set of control tank identifiers.
- $F \subset T'$ set of control tanks that when full indicate termination of the system.
- $e \in T \setminus T'$ the unique infinite sink of the system.
- $r \in T \setminus T'$ the unique infinite source in the system.
- $L : T \to \mathbb{N}_+$ The level at which the tanks are built, the lower the number, the conceptually higher the tank. Water can only flow from a tank with a lower number to one with a higher number. $L(r) = 0$ and $L(e) = \infty$.
- $C : T \to \mathbb{N}_+ \cup \{\infty\}$ capacity of the tanks. Where we assume that value tanks can be unbounded. Of course, for practical cases, all value tanks will need to have finite capacity. Control tanks all have capacity 1 (they act as Boolean values).
- V finite set of valve identifiers.
- $S : V \to \{t = n \mid t \in T, n \in \mathbb{N}_+\} \cup \{t \neq n \mid t \in T, n \in \mathbb{N}_+\}$ An expression from a valve identifier to check whether or not a tank has a certain volume.
- $P \subset T \times T \times \mathcal{P}(V)$ ($\mathcal{P}(V)$ denotes the power set over V) finite set of pipes where water flows from the first element to the second. A pipe (i, j, v) must have $L(i) < L(j)$, meaning water only flows in one direction ('down'). Where all valves must be open for water to flow through (a binary and of the conditions).

The system evolves in discrete time steps with the time denoted t where, at each step, a unit of water flows down each pipe if all valves on the pipe are open. If a tank has

less water than the number of outgoing pipes with all valves open at any step, the result will depend on the particular run time. Examples of runtimes include:

- The result is undefined.
- Non-deterministically decides which pipes get a unit of water.
- A priority-based ordering of pipes.

In this work, we do not specify the runtime as this problem never occurs in our construction. To clarify the model we present controlled subtraction:

- $T = \{x, y, x', y', q', z, z', r, e\}$
- $T' = \{x', y', z'\}$
- $F = \{z'\}$
- e
- r
- $L = \{(x, 1), (y, 1), (x', 1), (y', 1), (q', 2), (z, 3), (z', 3), (r, 0), (e, \infty)\}$
- $C = \{(x, \infty), (y, \infty), (x', 1), (y', 1), (q', 1), (z, \infty), (z', 1), (r, \infty), (e, \infty)\}$
- $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$
- $S = \{(1, q' = 1), (2, y \neq 0), (3, y = 0), (4, x = 0), (5, y' = 1), (6, y' = 0), (7, x' = 0), (8, x' = 1)\}$
- $P = \{(x, e, \{1, 2\}), (x, z, \{1, 3\}), (y, e, \{1\}), (x', e, \{1\}), (y', e, \{1\}),$
  $(r, q', \{5, 8\}), (q', z', \{3, 4, 5, 7\})\}$

Throughout the remainder of the paper, we present, where appropriate, a pictorial view of our functions and a set of equations that both describe the system. The equations define the next step for the time value t. These alternate descriptions seem more intuitive, and can be straightforwardly transformed into the above formal definition.

We note that one can view the valves as a group of conditions combined with a binary *and*. We do not specify them as such, as physical realisations of these more complicated valves may be difficult. However, in principle, they are the same.

4.1.3 Turing completeness

We first prove that our system can construct all unary primitive recursive functions. We assume that our value tanks are unbounded for this proof. In [9, 57, 60, 8] it was shown that these base functions and closure operators are sufficient to construct the unary primitive recursive functions:

- **Successor function**: $S(x) = x + 1$ (cf. Figure 4.8)

- **Subtraction function**: $B(x, y) = x - y$ (cf. Figure 4.2)
- **Composition operator**: $C(h, g)(x) = h(g(x))$ (cf. Figure 4.9)
- **Difference operator**: $D(f, g)(x) = g(x) - f(x)$ (cf. Figure 4.10)
- **Primitive recursion operator**: $P(f) = p, p(0) = 0,\ p(x + 1) = f(p(x))$ (cf. Figure 4.11)

To construct these functions we make two copy functions: a repeatable fetch (inplace copy) and a destructive copy. Essentially, *repeatable fetch* as seen in Figure 4.5 takes input x and outputs it to x_1, then refills x to its original content, recreated in tank a. The *destructive copy* as seen in Figure 4.6 takes an input x and outputs two copy's of x_1 and x_2, emptying x.

Successor function $S(x) = x + 1$: The successor function $S(x)$ adds one to the given input x. Using similar ideas to the addition displayed in Figure 4.3, we arrive at the function displayed in Figure 4.8, where instead of draining from y, we drain from the input control (the control would always contain a unit amount of water).

Composition operator $C(h, g)(x) = h(g(x))$: The composition operator as seen in Figure 4.9 is started by filling the control tank x'. Once the composition is started it runs the function $g(x)$, where $g(x)$'s output y is the input for $f(y)$, and $f(y)$'s output z is the output for the composition.

Difference operator $D(f, g)(x) = g(x) - f(x)$: The difference operator works similar to that of the composition, where the input to the difference is the input to the destructive copy function (Figure 4.6). The output of the destructive copy is then passed to f and g. The output of f and g is then passed to the subtraction function (Figure 4.2). Finally, the output of the subtraction is the output of the difference function D.

Primitive recursion operator $P(f) = p, p(0) = 0$, $p(x + 1) = f(p(x))$: The primitive recursion operator P as seen in Figure 4.11 can be explained by the program in Figure 4.7. If the counter is 0, then the result is 0. Otherwise, function f is repeatedly executed while decrementing x, until x becomes 0. Once x is zero, the function returns the result and fills the control tank . □

Theorem 4.1.1. *Our water system can construct all unary primitive recursive functions.*

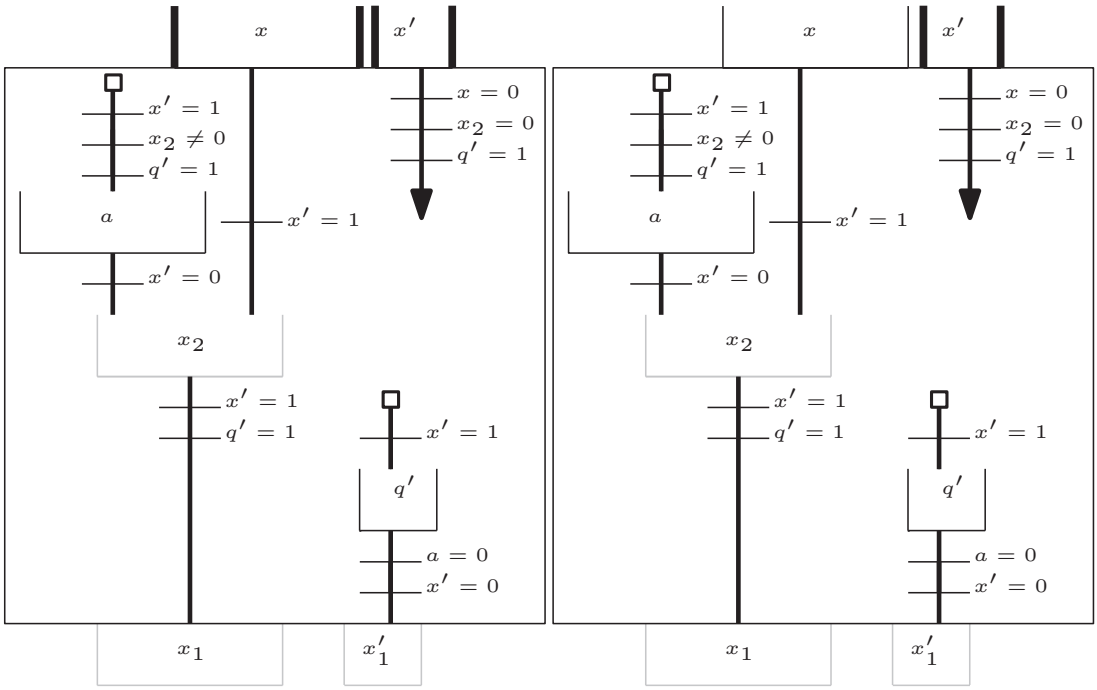


Figure 4.5: A diagram representing the repeatable fetch copy function $i(x) = x$. We have denoted inputs that contain water as a bold line and tanks that, after execution contain water as light grey. The image on the left is first execution of repeatable fetch where x sets the persistent storage and the image on the right the subsequent calls.

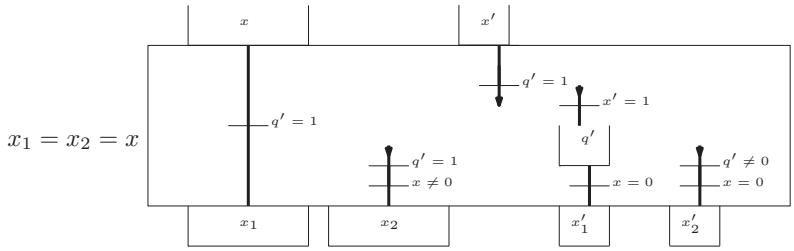


Figure 4.6: A diagram representing the destructive copy function $c(x) = x, x$.

Table 4.1: Equations for inplace copy.

$$
\begin{aligned}
a(t+1) = \quad & \textbf{if } x(t) \neq 0 \ \&\ q'(t) = 1 \textbf{ then } a(t) \oplus 1 \\
& \textbf{else if } x'(t) = 0 \textbf{ then } a(t) \ominus 1 \\
& \textbf{else } a(t) \\
\\
x(t+1) = \quad & \textbf{if } x'(t) = 0 \ \&\ a(t) > 0 \textbf{ then } x(t) \oplus 1 \\
& \textbf{else if } x'(t) = 1 \ \&\ q'(t) = 1 \textbf{ then } x(t) \ominus 1 \\
& \textbf{else } x(t) \\
\\
x'(t+1) = \quad & \textbf{if } x(t) = 0 \ \&\ q'(t) = 1 \textbf{ then } x(t) \ominus 1 \textbf{ else } x(t) \\
\\
q'(t+1) = \quad & \textbf{if } x'(t) = 1 \textbf{ then } q'(t) + 1 \\
& \textbf{else if } a(t) = 0 \ \&\ x'(t) = 0 \textbf{ then } q'(t) \ominus 1 \\
& \textbf{else } q'(t) \\
\\
x_1(t+1) = \quad & \textbf{if } x'(t) = 1 \ \ \& \ \ q'(t) = 1 \ \&\ x(t) > 0 \textbf{ then } x_1(t) \oplus 1 \textbf{ else } x_1(t) \\
\\
x'_1(t+1) = \quad & \textbf{if } a(t) = 0 \ \&\ x'(t) = 0 \ \ \& \ \ q'(t) > 0 \textbf{ then } x'_1(t) \oplus 1 \textbf{ else } x'_1(t)
\end{aligned}
$$

Table 4.2: Equations for destructive copy.

$$
\begin{aligned}
x(t+1) = \quad & \textbf{if } q'(t) = 1 \textbf{ then } x(t) \ominus 1 \textbf{ else } x(t) \\
\\
x'(t+1) = \quad & \textbf{if } q'(t) = 1 \textbf{ then } x'(t) \ominus 1 \textbf{ else } x'(t) \\
\\
q'(t+1) = \quad & \textbf{if } x'(t) = 1 \textbf{ then } q'(t) \oplus 1 \\
& \textbf{else if } x(t) = 0 \textbf{ then } q'(t) \ominus 1 \\
& \textbf{else } q'(t) \\
\\
x_1(t+1) = \quad & \textbf{if } q'(t) = 1 \ \&\ x(t) > 0 \textbf{ then } x_1(t) \oplus 1 \textbf{ else } x_1(t) \\
\\
x_2(t+1) = \quad & \textbf{if } q'(t) = 1 \ \&\ x(t) \neq 0 \textbf{ then } x_2(t) \oplus 1 \textbf{ else } x_2(t) \\
\\
x'_1(t+1) = \quad & \textbf{if } x(t) = 0 \ \ \& \ \ q'(t) > 0 \textbf{ then } x'_1(t) \oplus 1 \textbf{ else } x'_1(t) \\
\\
x'_2(t+1) = \quad & \textbf{if } x(t) = 0 \ \ \& \ \ q'(t) \neq 0 \textbf{ then } x'_2(t) \oplus 1 \textbf{ else } x'_2(t)
\end{aligned}
$$

```
1  if x' = 1 then
2      if x = 0 then
3          y' ← 1; y ← 0;
4          return
5      else
6          u ← 0;
7          loop:
8              v ← f(u);
9              if x = 0 then
10                 y ← v; y' ← 1;
11                 return
12             else
13                 x ← x − 1;
14                 u ← v;
15                 goto loop
```

Figure 4.7: Code to describe the execution of the primitive recursion operator $p = P(f), p(0) = 0$, $p(x+1) = f(p(x))$.

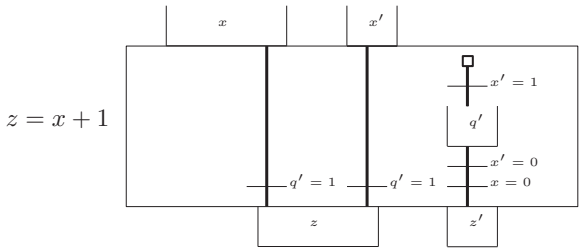


Figure 4.8: A diagram representing the successor function $S(x) = x \oplus 1$.

Table 4.3: Equations for the successor function.

$$
\begin{aligned}
x(t+1) = \quad & \textbf{if } q'(t) = 1 \textbf{ then } x(t) \ominus 1 \textbf{ else } x(t) \\
x'(t+1) = \quad & \textbf{if } q'(t) = 1 \textbf{ then } x'(t) \ominus 1 \textbf{ else } x'(t) \\
q'(t+1) = \quad & \textbf{if } x'(t) = 1 \textbf{ then } q'(t) \oplus 1 \\
& \textbf{else if } x(t) = 0 \textbf{ then } q'(t) \ominus 1 \\
& \textbf{else } q'(t) \\
z(t+1) = \quad & \textbf{if } q'(t) = 1 \;\&\; x(t) > 0 \;\&\; x'(t) > 0 \textbf{ then } z(t) \oplus 2 \\
& \textbf{else if } q' = 1 \;\&\; (x(t) > 0 \textbf{ or } x'(t) > 0) \textbf{ then } z(t) \oplus 1 \\
& \textbf{else } z(t) \\
z'(t+1) = \quad & \textbf{if } x'(t) = 0 \;\&\; x(t) = 0 \;\&\; q'(t) > 0 \textbf{ then } z'(t) \oplus 1 \textbf{ else } z'(t)
\end{aligned}
$$

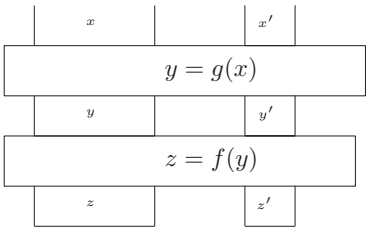


Figure 4.9: A diagram representing the composition of functions $C(x) = f(g(x))$.

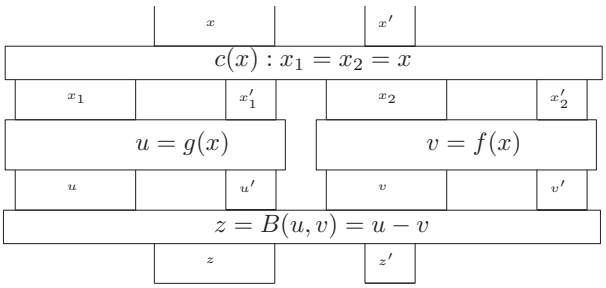


Figure 4.10: A diagram representing the difference of functions $D(x) = g(x) - f(x)$.

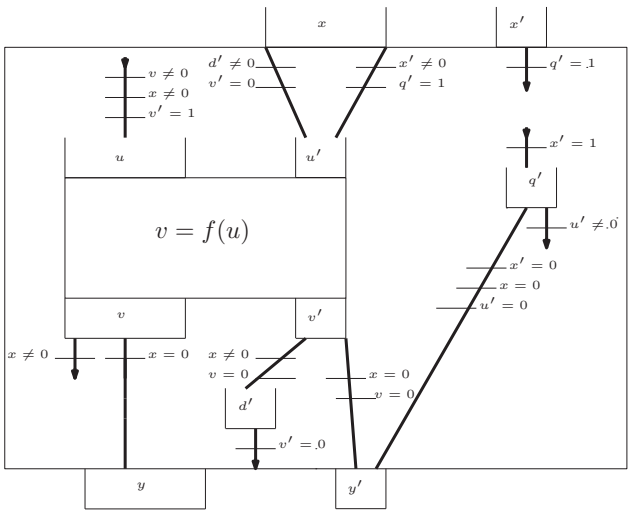


Figure 4.11: A diagram representing the primitive recursion operator $P(f) = p, p(0) = 0$, $p(x + 1) = f(p(x))$.

To prove Turing completeness, we require that our system can construct the unary primitive recursive functions as well as [57]:

- **Addition function**: $A(x, y) = x + y$ (see Figure 4.3)
- $\boldsymbol{\mu}$ **operator**: $\mu_y(f)(x) = \min_y\{f(x, y) = 0\}$

We note that the μ operator currently takes two arguments and all other functions take one (except addition and subtraction). To overcome this issue, we apply the Cantor pairing function [57]:

$$\Pi(x, y) = \frac{(x + y)(x + y + 1)}{2} + y = \frac{((x + y)^2 + 3y + x)}{2}$$

As squaring, multiplying by a constant, and division of one variable are primitive recursive, the only two variable function required to construct the Cantor pairing function is addition (see Figure 4.3). μ operator $\mu_y(f)(x) = \min_y\{f(x, y) = 0\}$: The μ operator as seen in Figure 4.12 first transfers the data from x to x'. It then utilises the inplace copy function (Figure 4.5) i on both x' and y (y is initially 0). It uses these copies to run $f(x, y)$. If $f(x, y)$ results in 0, then y is copied into the result z and the control tank is filled. If the result was not 0, then y is incremented and starts the copying process again until the result becomes 0 (which may or may not occur).

Theorem 4.1.2. *Our water system is Turing complete.*

Theorem 4.1.3. *Our water system can be viewed as a directed acyclic graph.*

Our construction of μ recursive functions (all base functions and the operators) required the pipes to only go in one direction ('down'). Hence, if the pipes are viewed as arcs and tanks as nodes, the digraph contains no cycles. This is in contrast to the system presented in [30], where loops were used to achieve operations such as multiplication and division.

4.1.4 High Level Rules

P systems

The system described in [30] contains a set of tanks that contain a fixed volume of water at any time step. Each tank is a data storage device working similarly to that of a cP system sub-cell. cP systems, however, can create or delete sub-cells during execution, whereas water tanks must be created before and cannot be removed. A rule in a cP system may cause a change to the number of sub-cells or the sub-cells content. Similarly, the pipes and valves of a water tank change the volume of water contained in the tanks. These similarities allow us to define the tank system as a restricted version of cP systems.

Our tank systems evolution can be defined by the grammar in Table 2.4. As the grammar makes no reference to the maximum value in a tank, it is described by a rule that subtracts any water that goes above the maximum allowed value from the

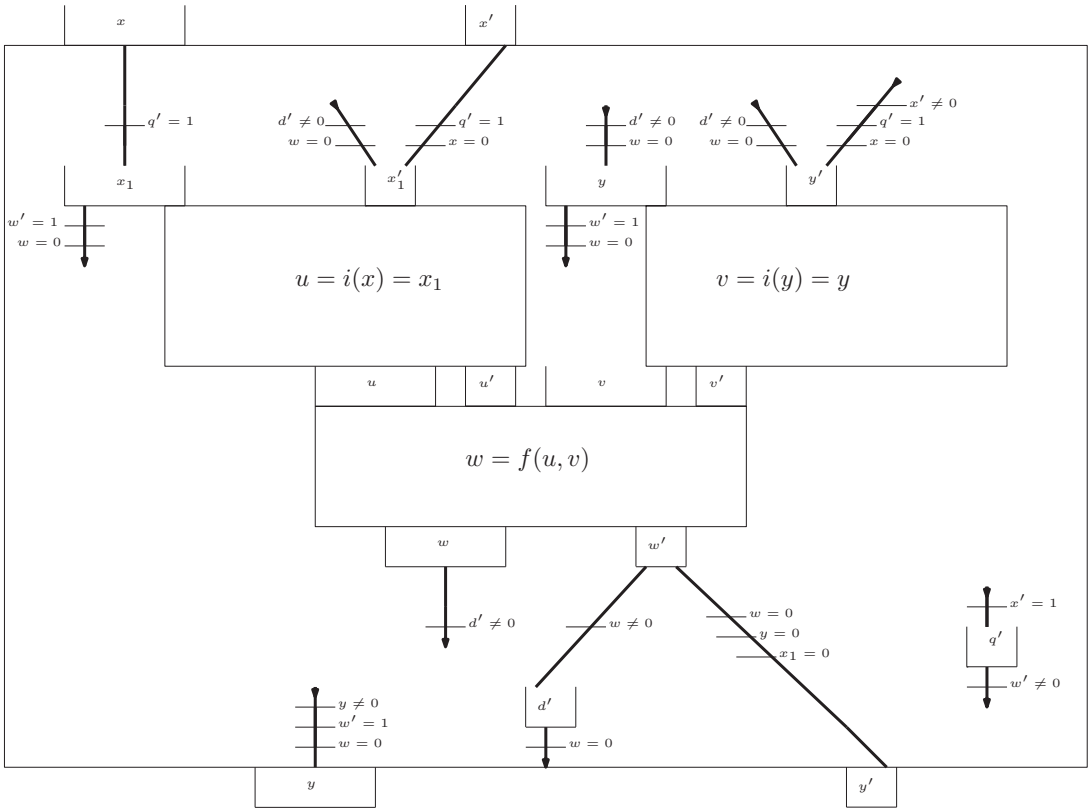


Figure 4.12: A diagram representing the μ operator $\mu_y(f)(x) = min_y\{f(x, y) = 0\}$.

tank. For example, if we have a tank v_x which has a maximum value of 2, then we add the rule in Table 4.4. This rule removes any water above the maximum and ensures the tank returns to its maximum value. We ignore this technical detail when describing our rule sets. We assume the maximum value rules are implicitly defined.

Table 4.4: Rule to ensure tank v_x does not exceed 2 units of water.

$s_1\ v_x(111\_)$	$\rightarrow$	$s_2\ v_x(11)$	(1)

Examples

We now present a few examples of rules for some of the gates presented earlier(ignoring the overflow for brevity):

1. **Addition**: Figure 2.3 shows a tank system for the saturated addition $y_1 = x_1 \oplus x_2$. This tank system can be faithfully emulated by the following cP ruleset:

Table 4.5: Rules to describe $y_1 = x_1 \oplus x_2$.

$s_1\ x_1(1X)\ x_2(1X)\ y_1(Y)$	$\rightarrow$	$s_1\ x_1(X)\ x_2(X)\ y_1(Y11)$	(1)
$s_1\ x_1(1X)\ y_1(Y)$	$\rightarrow$	$s_1\ x_1(X)\ y_1(Y1)$	(2)
$s_1\ x_2(1X)\ y_1(Y)$	$\rightarrow$	$s_1\ x_2(X)\ y_1(Y1)$	(3)

2. **Subtraction**: Figure 2.2 shows a tank system for the saturated subtraction $y_1 = x_1 \ominus x_2$. This tank system can be faithfully emulated by the following cP ruleset:

Table 4.6: Rules to describe $y_1 = x_1 \ominus x_2$.

$s_1\ x_1(X1)$	$\rightarrow$	$s_1\ x_1(X)$	$\mid$	$x_2(\_1)$	(1)
$s_1\ x_2(Y1)$	$\rightarrow$	$s_1\ x_2(Y)$			(2)
$s_1\ x_1(X1)\ y_1(Y)$	$\rightarrow$	$s_1\ x_1(X)\ y_1(Y1)$	$\mid$	$x_2()$	(3)

3. **Controlled subtraction:** The controlled subtraction presented in Figure 4.2 has the rules:

with a trace of the ruleset and tank system shown in the appendix.

Table 4.7: A ruleset for the controlled saturating subtraction operator $z = x \ominus y$.

$s_1\ q()$	$\rightarrow$	$s_2\ q(1)$	$\mid\ c_x(1)\ c_y(1)$	(1)
$s_2\ c_x(1)$	$\rightarrow$	$s_2\ c_x()$	$\mid\ q(1)$	(2)
$s_2\ c_y(1)$	$\rightarrow$	$s_2\ c_y()$	$\mid\ q(1)$	(3)
$s_2\ v_x(X1)$	$\rightarrow$	$s_2\ v_x(X)$	$\mid\ q(1)\ v_y(\_1)$	(4)
$s_2\ v_y(Y1)$	$\rightarrow$	$s_2\ v_y(Y)$	$\mid\ q(1)$	(5)
$s_2\ v_x(X1)\ v_z(Z)$	$\rightarrow$	$s_2\ v_x(X)\ v_z(Z1)$	$\mid\ q(1)\ v_y()$	(6)
$s_2\ q(1)\ c_z()$	$\rightarrow$	$s_3\ q()\ c_z(1)$	$\mid\ c_x()\ c_y()$	(7)

We note that utilising the full power of cP systems we can simplify the rule sets further. For example, a traditional cP solution to subtraction ($x \ominus y$) is achieved using the rules in Table 4.8. Noting that the full cP system solution takes only 1 step whereas, the water-based system will take a linear number of steps.

Table 4.8: Rules to describe $z = x \ominus y$.

$s_1\ x(YZ)\ y(Y)$	$\rightarrow$	$s_2\ z(Z)$	(1)
$s_1\ x(\_)\ y(\_)$	$\rightarrow$	$s_2\ z()$	(2)

4.1.5 Conclusions

We have proven that our water tank system based on the system proposed in [30] is Turing complete, via the construction of μ recursive functions. We have demonstrated how termination can be detected, and how to combine different functions without an exponential explosion of the number of valves. Furthermore, we have shown that the water tank system only requires water to flow in one direction (no loops are required).

4.2 Parallel computing with water

In this section we discuss using our new water system to construct more practical theoretical computing models. We first construct a programmable RAM and then extend this construction to a EREW PRAM.

4.2.1 Constructing a programmable RAM

Constructing a programmable RAM using water can be broken into phases. The first phase is to read the line number that we are up to. Using the opcode for that line, run a function that executes that opcode. After that has been done, check whether

the line number exceeds the program line count; if it does, then the RAM halts; otherwise, it will do the process again. To make it easier to follow, we shall also break up the construction into modules which can then be pieced together to form the entire RAM.

Execute line L of the program

Here we assume that the program being executed will be stored in water tanks $p_{1,1}, p_{1,2}, p_{1,3}, p_{1,4}, p_{2,1}, p_{2,2}, p_{2,3}, p_{2,4}, ..., p_{m,1}, p_{m,2}, p_{m,3}, p_{m,4}$. Where $p_{i,j}$ denotes the ith line with parameter j, noting that we assume that each line of code will have one op code followed by three parameters; the third tank contents will be ignored for operations of two parameters.

To start the program at line L and continue executing until we reach line $m + 1$, we utilise the tank system presented in Figure 4.14. For brevity we have presented an arbitrary program line $p_{o,j}$ but this can be expanded for all program lines by changing the valve for $p_{i,k}$ to $L = i$ for the ith line. For example, the tanks presented in Figure 4.14 for the Euclidean algorithm would initially contain the volumes presented in Table 4.9.

Noting that to run a function we use the tank system presented in Figure 4.15. The tank system presented in Figure 4.15 is not repeated, only one instance exists for a RAM. The line inputs $p_{i,j}$ all drain into the inputs p_1, p_2, p_3, p_4. Using traditional programming the operation can be explained to be the function **RAM** presented in Figure 4.13.

```
1 function RAM(p, L)//p = [p_{1,1}, p_{1,2}, p_{1,3}, p_{1,4}, p_{2,1}, p_{2,2}, ..., p_{m,4}]
2     i_1 ← p_{L-1,1}; i_2 ← p_{L-1,2}
3     i_3 ← p_{L-1,3}; i_4 ← p_{L-1,4}
4     s ← run_op(i_1, i_2, i_3, i_4, L)
5     if s ≠ m+1 then
6         RAM(p, s)
7     halt
```

Figure 4.13: Code to describe the outer loop of our RAM system.

Table 4.9: Initial volumes of water for tanks presented in Figure 4.14 for Euclidean algorithm.

i	$p_{i,1}$	$p'_{i,1}$	$p_{i,2}$	$p'_{i,2}$	$p_{i,3}$	$p'_{i,3}$	$p_{i,4}$	$p'_{i,4}$
1	3	1	3	1	0	1	2	1
2	6	1	6	1	3	1	0	0
3	3	1	3	1	2	1	1	1
4	6	1	8	1	3	1	0	0
5	6	1	10	1	1	1	0	0
6	3	1	1	1	1	1	2	1
7	6	1	1	1	1	1	0	0
8	3	1	2	1	2	1	1	1
9	6	1	2	1	1	1	0	0

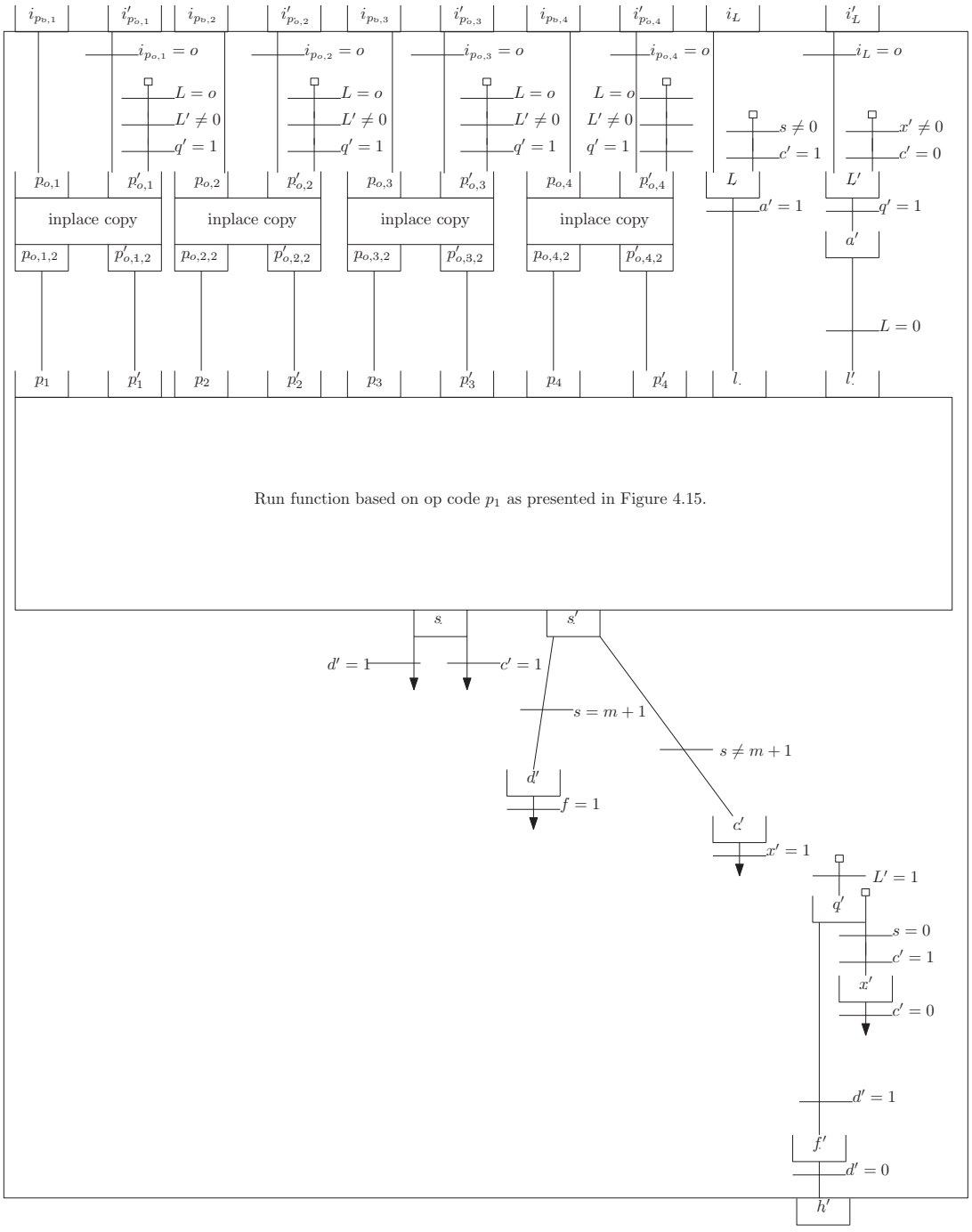


Figure 4.14: A diagram representing the an outer loop of a RAM. Where the input L is the line to start the program's execution (typically line 1). After the operation has completed tank s will contain the next line to execute. Tank s will then be drained and L filled with that contents. Once the program has ended tank h' will be full. Noting that we have shown only for one arbitrary program line o.

Executing an operation code (run_op)

Once the line number is read and the operation number selected, it is passed into the inner function presented in Figure 4.15. This will utilise the opcode stored in p_1 and decide which operation to execute. For simplicity, we have presented an arbitrary operation X; this can be expanded for all operations by taking $X = 1, 2, 3, 4, 5, 6$. This operation can alternatively be summarised by the function **run_op** presented in Figure 4.16.

As described earlier, a RAM can be constructed using six basic operations. However, to simplify these operations, we utilise the fact that these operations can be viewed as a sequence of read and writes from registers.

Utilising read and write operators, we construct the base operations. For brevity, we only show functions 1, 2, 4, and 6; functions 3 and 5 can be straightforwardly derived from functions 2 and 4, respectively.

Although operation one is near identical to the write operation, it is important to note that the operations need to also return the new line number after completing, hence we use the increment function. Operation 1 can be viewed in Figure 4.18. A programmatic view of operation one is presented in Figure 4.19

Operation 2 can be viewed in Figure 4.20 where we have left out the line number increment. The line increment can be done just as we did in the first operation. Operation 4 can be viewed in Figure 4.21. Operation 6 can be viewed in Figure 4.22.

We implement the read and write functions in Figure 4.23 and Figure 4.24 respectively. We note that these two functions are reading and writing to the same set of registers, so the registers $r_1, ..., r_n$ are shared between the figures.

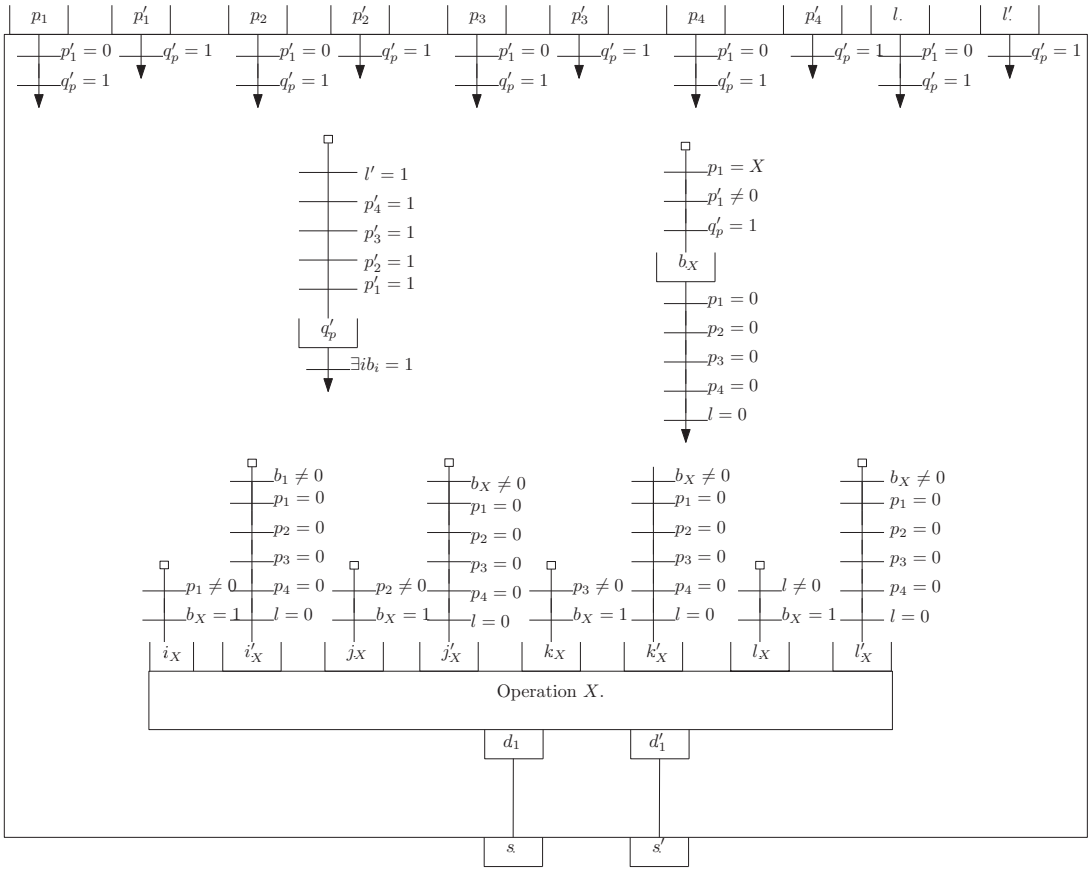


Figure 4.15: A diagram representing which of the 6 functions will be executed. Where for simplicity we have shown for an arbitrary opcode X where $X \in \{1, 2, 3, 4, 5, 6\}$. Selected operations are presented: Operation 1 in Figure 4.18, Operation 2 in Figure 4.20, Operation 4 in Figure4.21 and Operation 6 in Figure 4.22. Noting that $\exists i b_i = 1$ is a shorthand to describe six pipes from q'_p each with one valve as shown in Figure 4.17.

```
1  function run_op(p1, p2, p3, p4, l)
2      switch p1
3          case 1
4              return const(p2, p3, l)
5          case  2
6              return add(p2, p3, p4, l)
7          case 3
8              return sub(p2, p3, p4, l)
9          case 4
10             return indr(p2, p3, l)
11         case 5
12             return indw(p2, p3, l)
13         case 6
14             return tra(p2, p3, l)
```

Figure 4.16: Code to describe the outer loop of our RAM system.

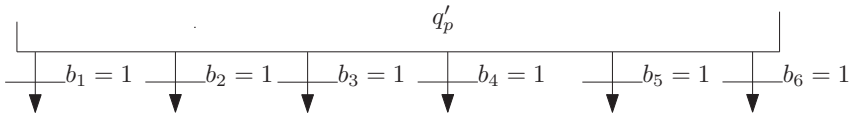


Figure 4.17: Diagram to show the expanded version of the shorthand $\exists i b_i = 1$ for tank q'_p.

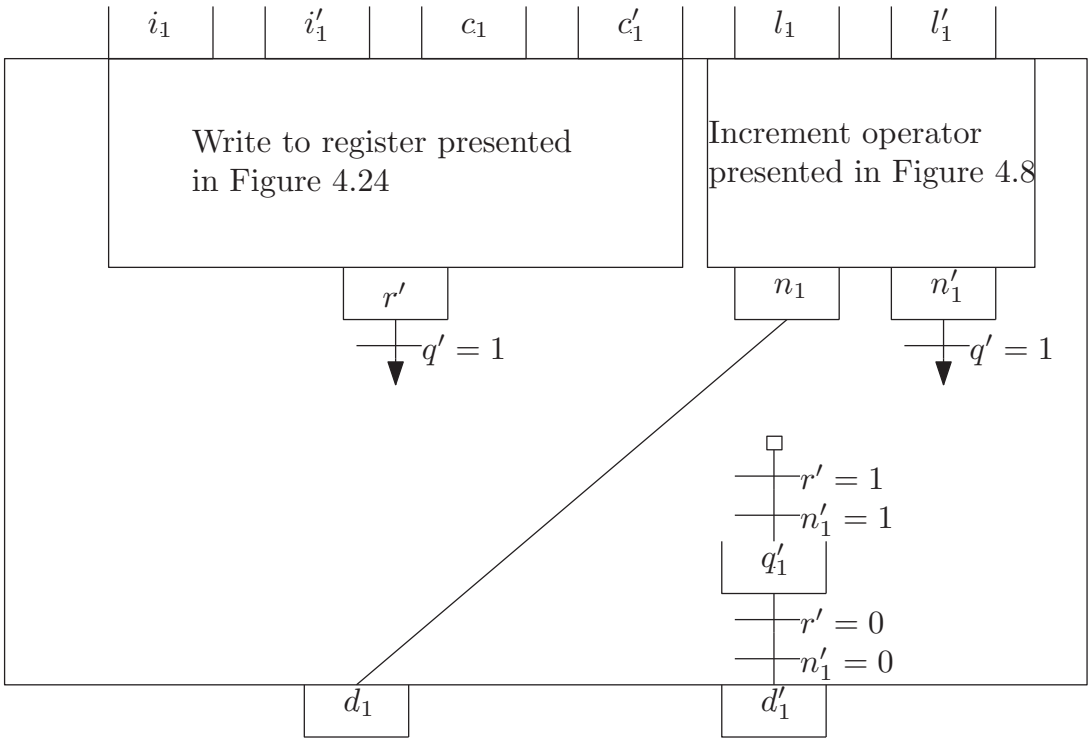


Figure 4.18: A diagram representing operation 1: $r_i \leftarrow C$. Returns $l \oplus 1$ where l is the program line number.

```
function const(i, c, l)
    r_i ← c
    return l + 1
```

Figure 4.19: Code to describe the constant function.

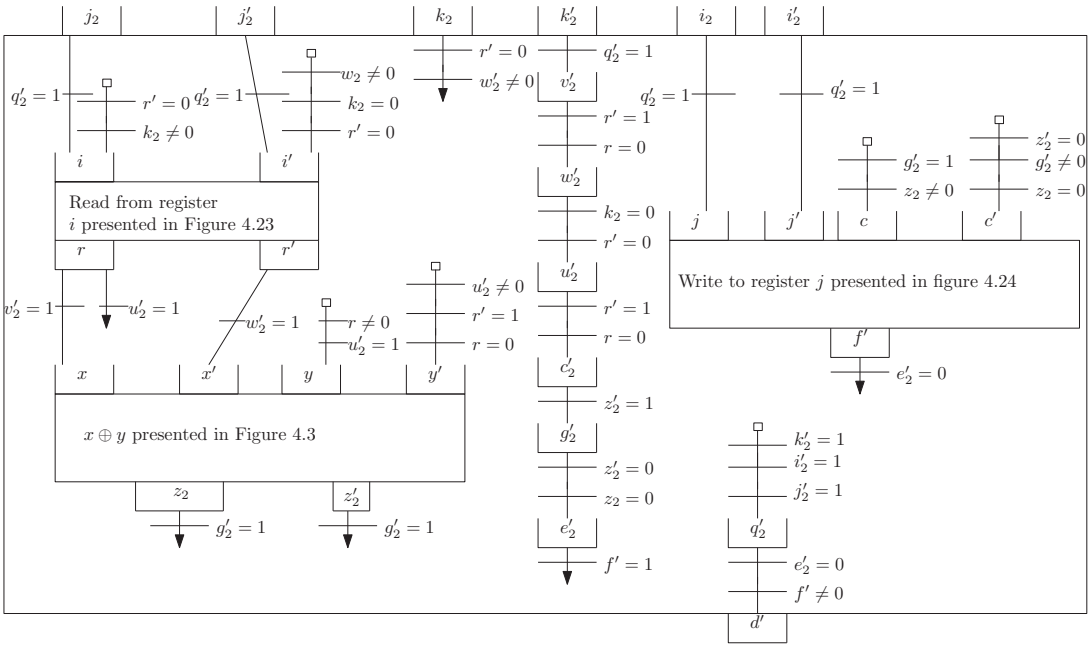


Figure 4.20: A diagram representing operation 2: $r_i \leftarrow r_j \oplus r_k$.

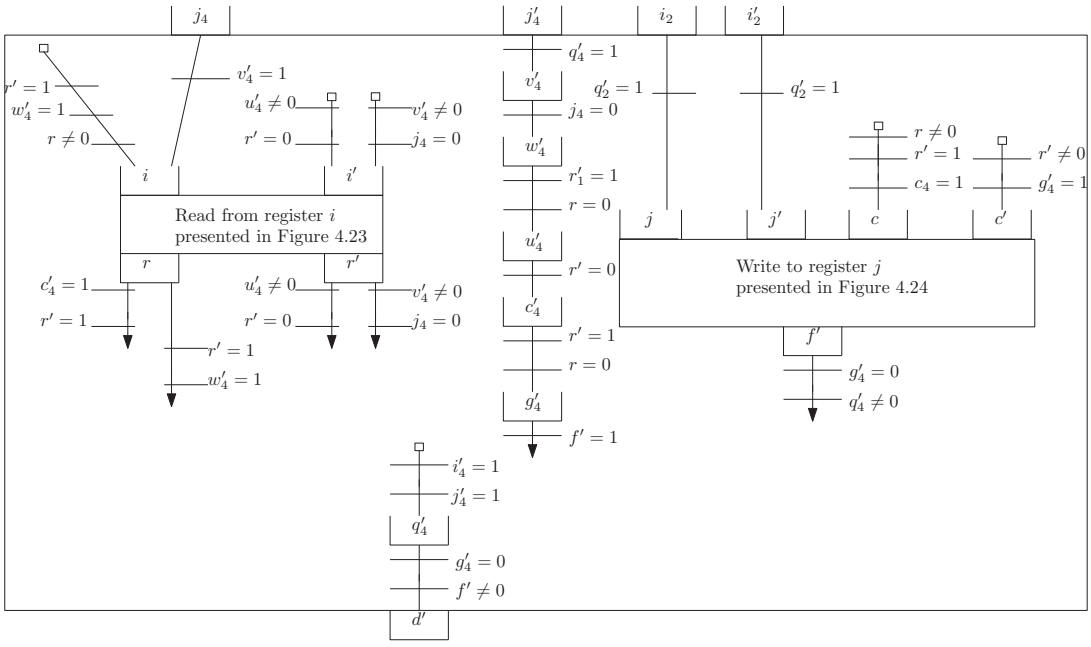


Figure 4.21: A diagram representing operation 4: $r_i \leftarrow r_{r_j}$.

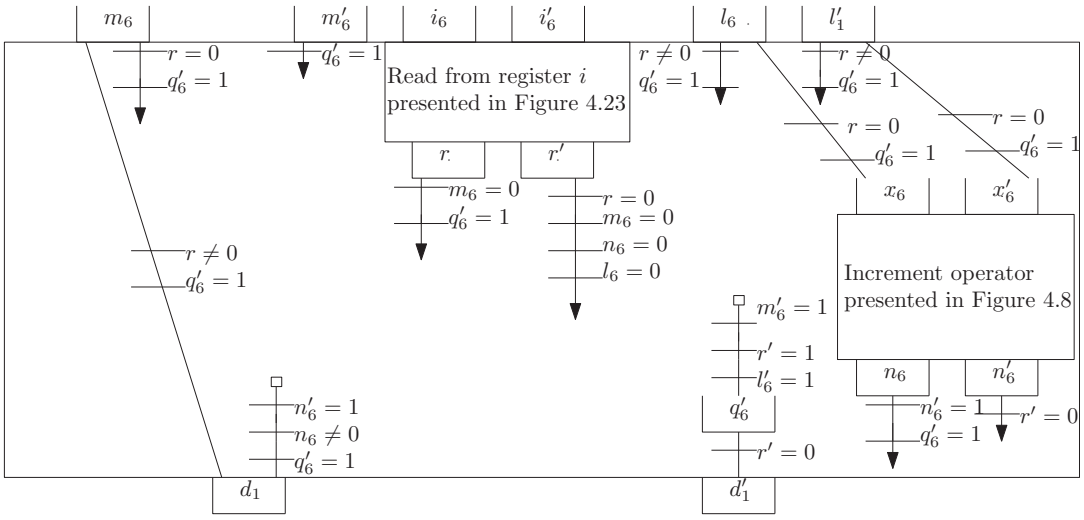


Figure 4.22: A diagram representing operation 6: TRA $m\ r_i > 0$.

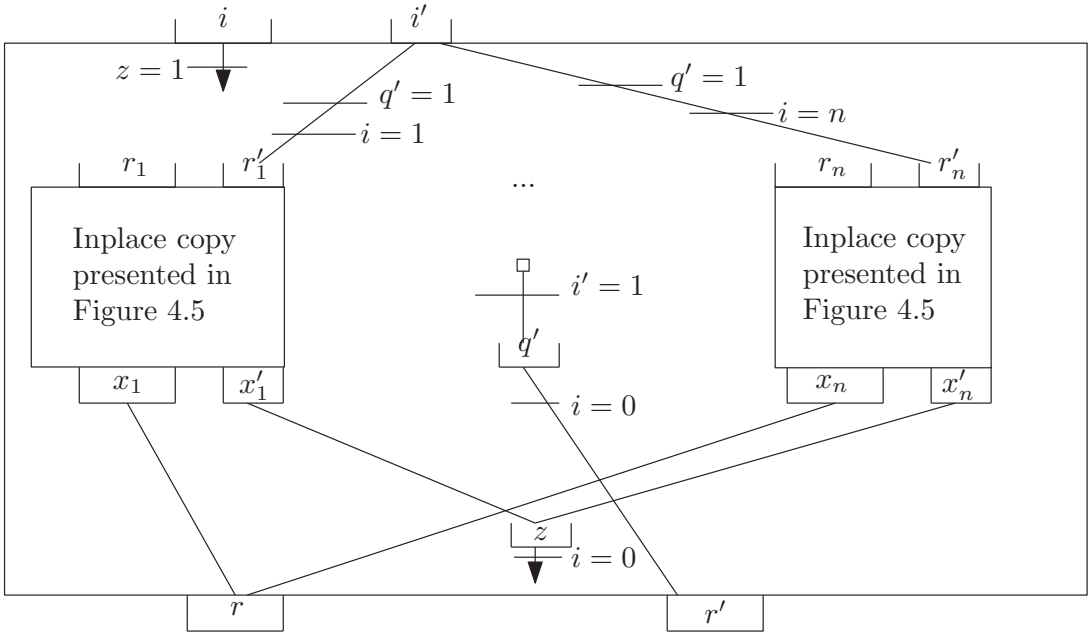


Figure 4.23: A diagram representing read from register i.

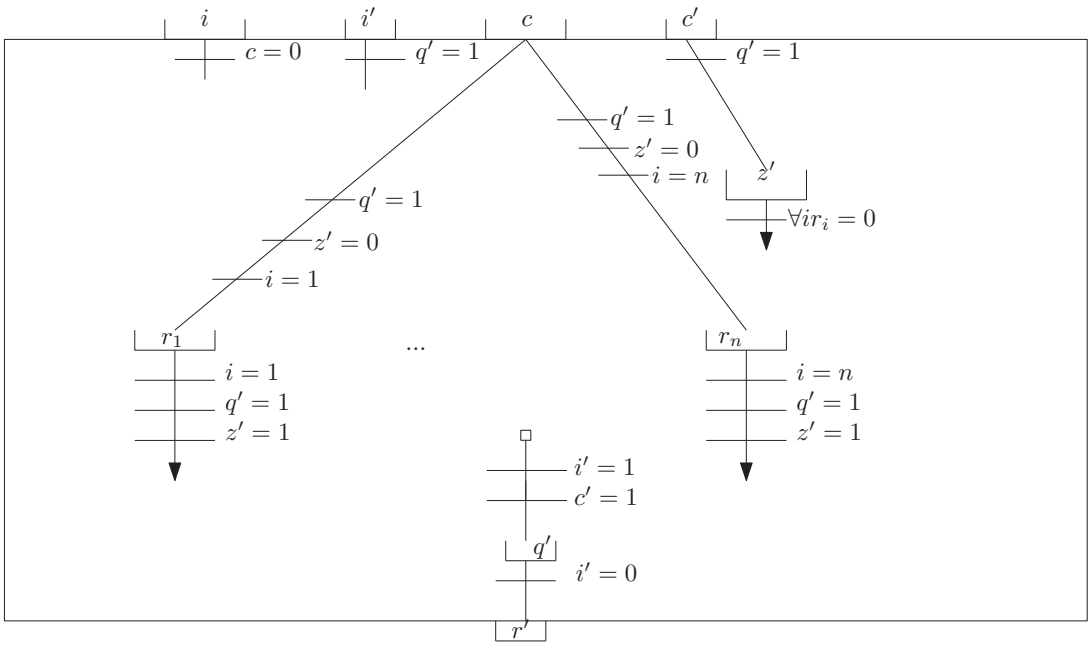


Figure 4.24: A diagram representing write to register i the value $c : r_i \leftarrow c$. Where the $\forall$ is a shorthand to mean n valves with each valve being $r_i = 0$.

4.2.2 Extending to PRAM

Extending to an EREW PRAM, we note the following:

1. The only way for processors to communicate is via read/writes to the shared memory.
2. The output of the PRAM will be the content stored on the shared memory once all processors halt.
3. At each time step, if any two processors try to read and or write from the same shared memory location, the result will be undefined.
4. Each operation will be run synchronously, i.e. at every time step, each processor fully completes one program line.
5. We assume that each processor has its own finite program and a set of local registers.

In this subsection, we extend our previously constructed RAM model to be a single processor. We note that currently, our RAM model defined earlier does not have operations to read/write to shared memory. However, more importantly, it does not satisfy the synchronous behaviour required for operations. In this subsection, we first describe adding shared reads and writes to our previously discussed model. We then describe adding a synchronous lock to ensure each processor evaluates each line of its program simultaneously.

Denoting shared registers ρ we define the following additional operators:

- $r_i \leftarrow \rho_j$: Read from shared memory and store in a local register.
- $\rho_i \leftarrow r_i$ Read from a local register and write to a shared register.

To define these new functions, we note that they are a simpler version of the indirect operation presented in Figure 4.21. With only one read but, instead of the read from a local register r, we define a read from shared memory ρ.

We extend the reading and writing of the local registers presented in Figure 4.23 and Figure 4.24. To extend it, we have each processor $1, 2, ..., p$ have its own input, output, and control tanks for the shared reads and writes. The diagram presented in Figure 4.25 shows how an arbitrary register X reads from shared memory.

The extension to the read can similarly be used for the write, which we omit for brevity. It is important to note that for an EREW PRAM, at any one time step, only one of the processors may read or write to a register. If multiple reads, writes, or a combination are done on the same shared memory, the computation will have an undefined result.

To ensure that each line of code is run synchronously each processor must wait until all other processors have completed their current line. To achieve this, we alter the outer

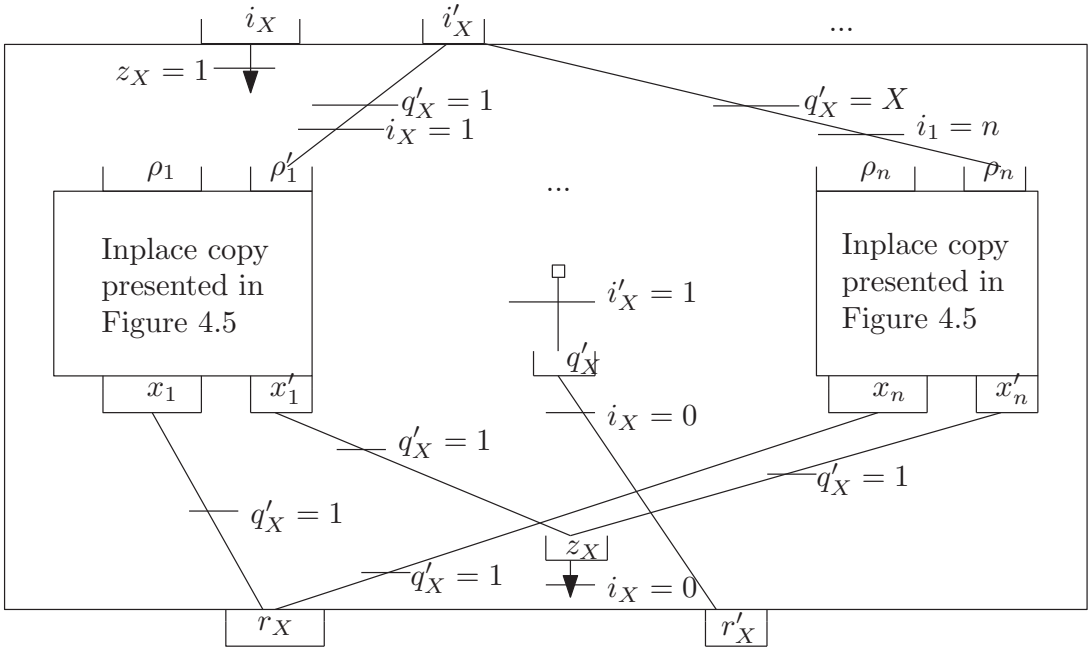


Figure 4.25: Parallel read for arbitrary processor X.

program presented in Figure 4.14 to that presented in Figure 4.26 and Figure 4.27.

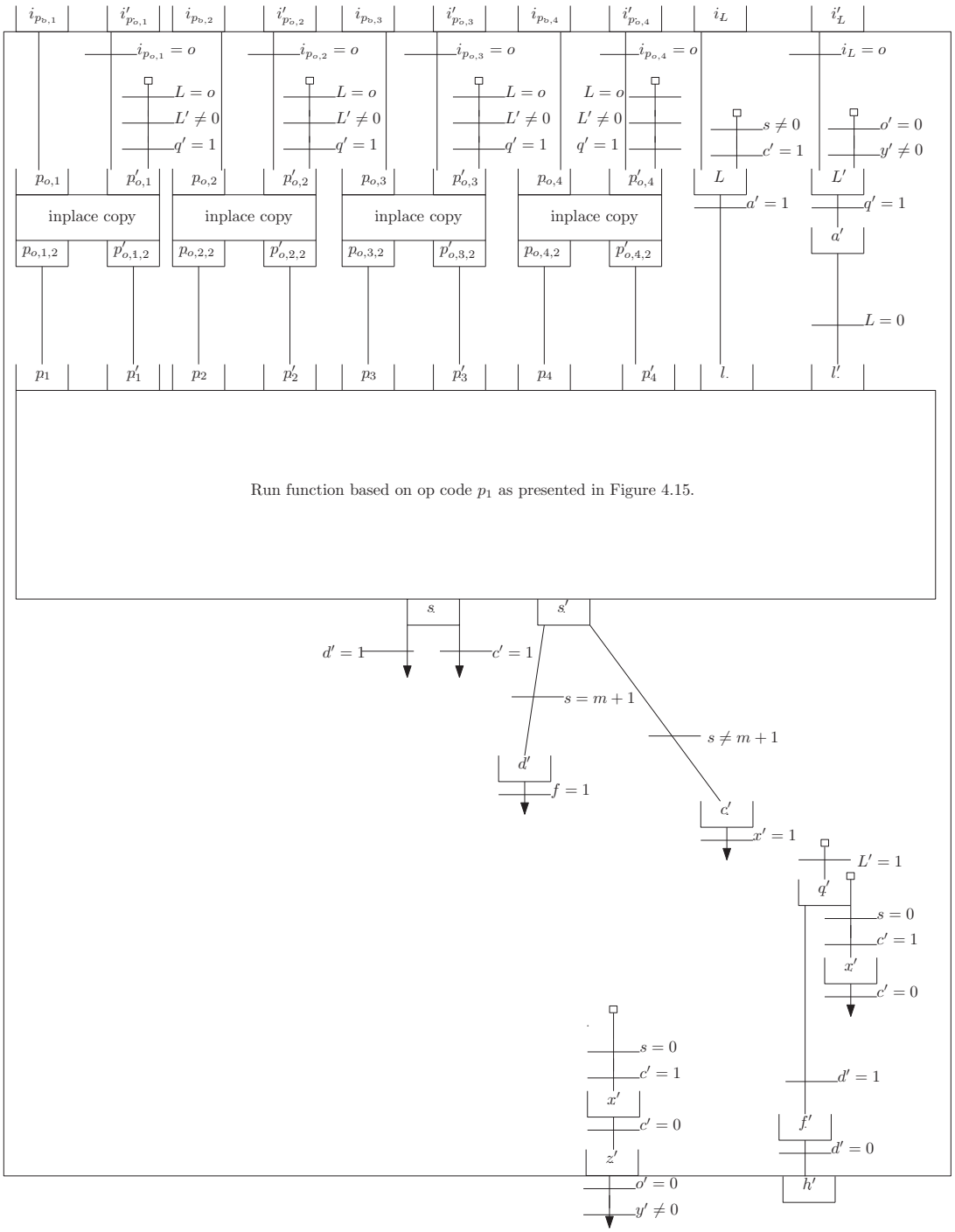


Figure 4.26: Parallel outer program that will only loop based on the synchronisation presented in Figure 4.27

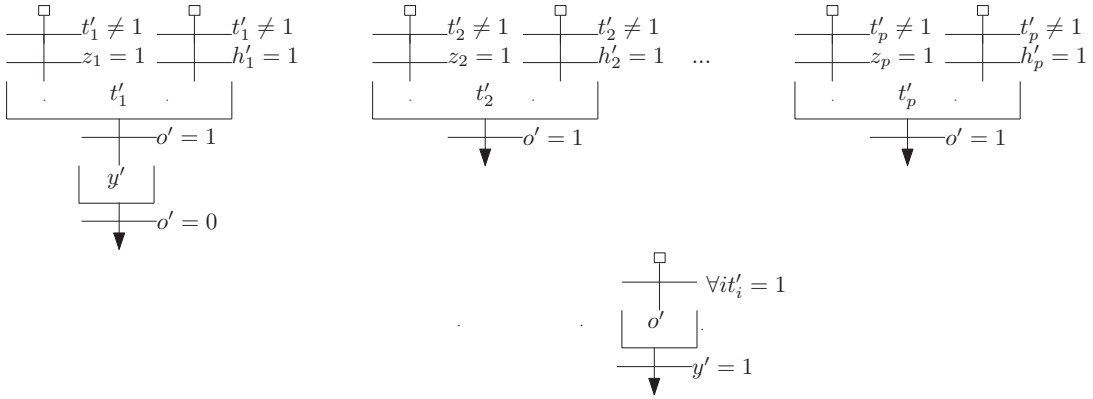


Figure 4.27: The synchronisation scheme which ensures that only after all other processors have done an operation may they go to the next line. Noting that we use the shorthand $\forall i t'_i = 1$ which expands to the valves presented in Figure 4.28

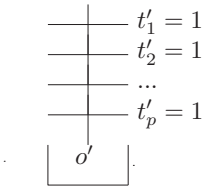


Figure 4.28: Expanded version of $\forall i t'_i = 1$.

4.2.3 Conclusions

Using the previously defined water system [28] we have constructed: 1) a programmable RAM and 2) a programmable EREW PRAM. This demonstrates that the non-centrally controlled model is inherently parallel and can model one of the most well-known parallel computing models.

4.3 Conclusions and Future Work

We have proven that our water tank system based on the system proposed in [30] is Turing complete, via the construction of μ recursive functions. We have solved the open problems of: how termination can be detected, and how to combine different functions without an exponential explosion of the number of valves. Furthermore, we have shown that the water tank system only requires water to flow in one direction (no loops are required). With our new system we then showed that it can 'efficiently' construct a programmable EREW PRAM.

This demonstrates the non-centrally controlled model is inherently parallel and can model one of the most well known parallel computing models. However, we have only modelled the least powerful of the PRAM models, leaving future work to look at allowing concurrent reads and/or writes.

Future work also includes exploring asynchronous circuits which do not require all inputs to be filled before processing starts. For example, a logical 'or' gate could proceed as soon as one of the inputs is full. Another open problem is the cost-based minimisation of the number of valves and pipes. Future work could look at physical realisations of this system and determining which is more 'expensive' based on a well defined measurement of cost. Physical realisations of the system could also have many other benefits such as for education.

Chapter 5

Applications in distributed computing

In this chapter we look at distributed cP systems (multiple top level cells) for solving different practical problems. In the first section we look at solving the Byzantine agreement problem and show that our actor based cP system model is able to model a solution efficiently surpassing previous P system solutions. In the second section we look at the Santa Claus problem and show how cP systems are able to model the problem efficiently.

5.1 Byzantine agreement

In this section, we provide a novel Actor-based cP solution for the Byzantine agreement problem, based on *Exponential Information Gathering* (EIG) trees, for N processes connected in a complete graph. Each Byzantine process is modelled by exactly one cell, which is a considerable improvement over previous solutions: Dinneen et al. [17] uses $3N + 1$ cells per Byzantine process, and Nicolescu [49] still uses N cells per Byzantine process. In fact, our novel cP model is a substantial improvement on all other criteria, such as: vocabulary size, ruleset size, and runtime performance.

Subsections 5.1.1 and 5.1.2 discuss the Byzantine algorithm and its classical implementation based on EIG trees; this material is based on [17, 18]. Subsection 5.1.3 gives a bird's eye view of the complex artefacts required for the previous P system solutions. Subsection 5.1.4 presents our new solution, based on our new actor-like input control. Subsection 5.1.5 evaluates the merits of the new version against the previous versions and presents conclusions.

5.1.1 EIG Trees

We assume that the reader is familiar with the basic terminology and notations: functions, relations, graphs, nodes (vertices), arcs, directed graphs, dags, trees, alphabets, strings and multisets. Given two sets, A, B, a subset f of their cartesian product, $f \subseteq A \times B$, is a *functional relation* if $\forall (x, y_1), (x, y_2) \in f \Rightarrow y_1 = y_2$. Obviously, any function $f : A \rightarrow B$ can be viewed a functional relation, $\{(x, f(x)) \mid x \in A\}$, and, vice-versa, any functional relation can be viewed as a function.

The set of *permutations* of n of length m is denoted by $P(n, m)$, i.e. $P(n, m) = \{\pi : [1, m] \rightarrow [1, n] \mid \pi \text{ is injective}\}$. A permutation π is represented by the sequence of its values, i.e. $\pi = (\pi_1, \pi_2, \ldots, \pi_m)$, and we will often abbreviate this further as the sequence $\pi = \pi_1.\pi_2 \ldots \pi_m$. The sole element of $P(n, 0)$ is denoted by $()$, or by λ, if the context removes any possible ambiguity. Given a subrange $[p, q]$ of $[1, m]$, we define a *subpermutation* $\pi(p : q) \in P(n, q - p + 1)$ by $\pi(p : q) = (\pi_p, \pi_{p+1}, \ldots, \pi_q)$. The *image* of a permutation π, denoted by $\mathrm{Im}(\pi)$, is the set of its values, i.e. $\mathrm{Im}(\pi) = \{\pi_1, \pi_2, \ldots, \pi_m\}$. The *concatenation* of two permutations is denoted by $\odot$, i.e. given $\pi \in P(n, m)$ and $\tau \in P(n, k)$, such that $\mathrm{Im}(\pi) \cap \mathrm{Im}(\tau) = \emptyset$, $\pi \odot \tau = (\pi_1, \pi_2, \ldots, \pi_m, \tau_1, \tau_2, \ldots, \tau_k) \in P(n, m + k)$.

An EIG tree $T_{N,L}$, $N \geq L \geq 0$, is a labelled rooted tree of height L that is defined recursively as follows. The tree $T_{N,0}$ is a rooted tree with just one node, its root, labelled λ. For $L \geq 1$, $T_{N,L}$ is a rooted tree with $1 + N|T_{N-1,L-1}|$ nodes (where $|T|$ is the size of tree T), root λ, having N subtrees, where each subtree is isomorphic with $T_{N-1,L-1}$ and each node, except the root, is labelled by an element of $[1, N]$ that is *different* from any ancestor node (and also different from any left sibling node, if we want to display it like an ordered tree). Note that, $T_{N,L-1}$ is isomorphic and identically labelled with the tree obtained from $T_{N,L}$ by deleting all its leaves.

It is straightforward to see that there is a bijective correspondence between the permutations of $P(N, L)$ and the sequences (concatenations) of labels on all paths from the root to the leaves of $T_{N,L}$. Thus, each node σ in an EIG tree $T_{N,L}$ is uniquely identified by a permutation $\pi_\sigma \in P(N, l)$, where $l \in [0, L]$ is also σ's depth, and, vice-versa, each such permutation π has a corresponding node σ_π. We will further use this node-permutation identification, while referring to nodes.

Given EIG tree $T_{N,L}$, an attribute is a function $\aleph : T_{N,L} \rightarrow V$, for some value set V; alternatively, $\aleph$ can be given as a functional subset of $\{\pi \in P(N, t) \mid t \in [0, L]\} \times V$. The classical EIG-based Byzantine algorithm uses two attributes: (i) a top-down attribute *val*, here called α; and (ii) a bottom-up attribute *newval*, here called β.

Figure 5.1 illustrates three isomorphic EIG trees, (a) $T^2_{4,2}$, (b) $T^3_{4,2}$, (c) $T^4_{4,2}$. As we will see next, these are the EIG trees built by non-faulty processes 2, 3, 4 (respectively) in our sample scenario, where process 1 is Byzantine-faulty (so its own internal structure is irrelevant).

Consider EIG tree 5.1.b, for process 3, $T^3_{4,2}$. Level 0 corresponds to permutation set

$\{\lambda\}$. Level 1 corresponds to permutation set $\{(1), (2), (3), (4)\}$. Level 2 corresponds to permutation set $\{(1,2), (1,3), (1,4), (2,1), (2,3), (2,4), (3,1), (3,2), (3,4), (4,1), (4,2), (4,3)\}$. This tree is decorated with two attributes, α and β. Using the alternate notation for permutations (to avoid embedded parentheses), attribute α corresponds to the functional relation $\{(\lambda,1), (1,0), (2,0), (3,1), (4,1), (1.2,0), (1.3,0), (1.4,1), (2.1,0), (2.3,0), (2.4,0), (3.1,0), (3.2,1), (3.4,1), (4.1,1), (4.2,1), (4.3,1)\}$.

5.1.2 The EIG Algorithm

Each process starts with its *own* initial decision choice. At the end, all non-faulty processes must take the same final decision, even if the faulty processes attempt to disrupt the agreement, accidentally or intentionally.

A Byzantine faulty process is the most powerful consensus adversary. It can do anything, except changing or stopping messages sent by non-faulty processes: (*i*) it can read (but not change) other messages; (*ii*) it can stop sending its own expected messages; (*iii*) it can send conflicting messages; (*iv*) it can send malformed messages (including unexpected, out-of-round); (*v*) it can send forged messages, pretending to come from other processes – aka Sybil attack.

The classical EIG-based algorithm solves the Byzantine agreement problem in the *binary decision* case (true $= 1$, false $= 0$), for N *processes*, connected in a *complete graph* (where edges indicate *reliable duplex communication lines*), provided that $N \geq 3F + 1$, where F is the maximum number of faulty processes. This is a *synchronous* algorithm; celebrated results (see for example [40]) show that the Byzantine agreement is *not* possible if $N \leq 3F$, in the *asynchronous* case or when the communication links are *not* reliable.

Without providing a complete description, we provide a sketch of the classical algorithm, *reformulated* based on the theoretical framework introduced in subsection 5.1.1. For a more complete and verbose description of this algorithm, including correctness and complexity proofs, we refer the reader to Lynch [40].

Each non-faulty process, h, has its own copy of an EIG tree, $T^h_{N,L}$, where $L = F + 1$. This tree is decorated with two attributes, $\alpha^h, \beta^h : \{\pi \in P(N,t) \mid t \in [0,L]\} \rightarrow \{0, 1, \texttt{null}\}$, where `null` designates undefined items (not yet evaluated). Attributes α^h and β^h are also known as val_h and $newval_h$ [40], or *top-down* and *bottom-up* [17]. As their alternative names suggest, α^h is first evaluated, in a top-down tree traversal, in increasing level order; next, β^h is evaluated, in a bottom-up traversal, in decreasing level order.

The algorithm works in two phases. Its *first phase* is a *messaging* phase which completes the evaluation of the top-down attribute α^h. Initially, $\alpha^h(\lambda) = v^h$, the initial choice of process h; all the other α^h and β^h values are still undefined. Next, there are L messaging rounds. At round $t \in [1,L]$, h broadcasts to all processes (including self), a reversibly encoded message which identifies its α^h values at level $t-1$

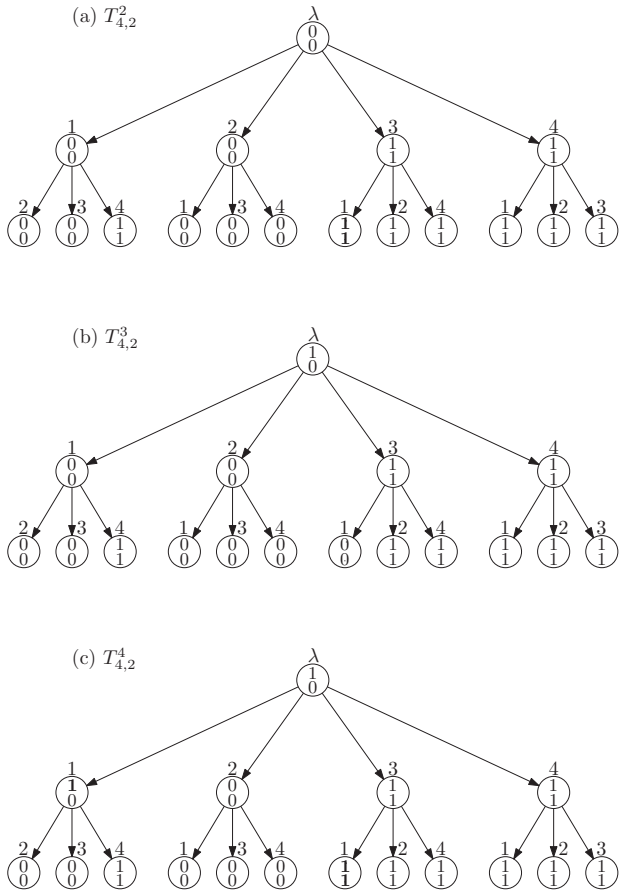


Figure 5.1: Three sample EIG trees, $T^h_{4,2}$, $h \in \{2, 3, 4\}$, completed with two attributes, α and β. The node labels appear besides the node blob. Each node blob contains its two attribute values: the top-down α value at the top, and the bottom-up β value at the bottom.

and their EIG destinations. Here, we encode all these via the set $\{(\pi \odot h, \alpha^h(\pi)) \mid \pi \in P(N, t-1), h \notin \mathrm{Im}(\pi)\}$. All other non-faulty processes broadcast messages, in a similar way. More compact encodings are possible, but we don't follow this issue here.

Process h decodes and processes the messages that it receives. From process f, $f \in [1, N]$, process h receives the set $\{(\pi \odot f, \alpha^f(\pi)) \mid \pi \in P(N, t-1)\}$. Each item $(\pi \odot f, \alpha^f(\pi))$ is used to assign further α^h values, to the next level down the EIG tree, by setting $\alpha^h(\pi \odot f) = \alpha^f(\pi)$.

As this formula suggests, it is indeed *critical* that h "knows" the origin f of each received message and that this origin mark cannot be faked by faulty processes that may use so-called Sybil attacks. Wrong or missing values are replaced by the value of a predefined default parameter, $v_0 \in \{0, 1\}$. Thus, there are L messaging rounds, and, after the last round, all nodes are decorated with values of attribute α. In fact, only the last level α values are actually needed, to start the next phase, a practical implementation can choose to discard the other α values.

Then, the algorithm switches to its *second phase*, the evaluation of the bottom-up attribute β^h. First, for leaves, $\beta^h(\pi) = \alpha^h(\pi), \pi \in P(N, L)$.

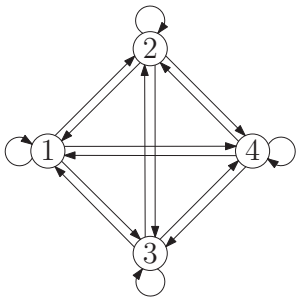
Next, given β^h values for level $t \in [1, L]$, each β^h value for the next level up, $\beta^h(\pi), \pi \in P(N, t-1)$, is evaluated on the basis of the β^h values of node π's children, i.e. on the multiset $\{\beta^h(\pi \odot f) \mid f \in [1, N] \setminus \mathrm{Im}(\pi)\}$, using a local majority voting scheme: $\beta^h(\pi) = 0$, if a strict majority of the above multiset values are 0; or, $\beta^h(\pi) = 1$, if a strict majority of the above multiset values are 1; or, $\beta^h(\pi) = v_0$ (the same default parameter mentioned above), if there is a tie.

At the end, the β^h value for the EIG root, $\beta^h(\lambda)$, is process h's final decision. All non-faulty processes will simultaneously reach the same final decision; any decision taken by faulty nodes is not relevant.

Consider a Byzantine scenario with $N = 4$ and $F = 1$, thus $L = 2$. Assume that processes 1, 2, 3 and 4 start with initial choices 0, 0, 1, and 1, respectively. Further, assume that process 1 is faulty. Figure 5.2 shows sample messages which could be exchanged in this scenario and Figure 5.1 shows the corresponding EIG trees, for non-faulty processes 2,3,4.

Each of the non-faulty processes, 2, 3 and 4, broadcasts identical messages to each of the four processes. The faulty process 1 sends conflicting messages. In our scenario, $x = 0$, in the message sent to 1, 2 and 3, but $x = 1$, in the message sent to 4. Also, $y = 1$, in the message sent to 1, 2 and 4, but $y = 0$, in the message sent to 3. White spaces are placeholders indicating potential messages which are not created, because they would have contained duplicated process numbers (1.1, 2.2, 3.3, 4.4). The second phase is not detailed here, except for the common final decisions (the question mark indicates an irrelevant value).

The second phase is illustrated in Figure 5.1, for all non-faulty processes $2, 3, 4$. All



Process	1	2	3	4
Initial choice	0	0	1	1
Faulty	Yes	No	No	No
Round 1 messages	$(1, x)$	$(2, 0)$	$(3, 1)$	$(4, 1)$
Round 2 messages	$(2.1, 0)$ $(3.1, y)$ $(4.1, 1)$	$(1.2, 0)$ $(3.2, 1)$ $(4.2, 1)$	$(1.3, 0)$ $(2.3, 0)$ $(4.3, 1)$	$(1.4, 1)$ $(2.4, 0)$ $(3.4, 1)$
Final decision	?	0	0	0

Figure 5.2: A sample Byzantine scenario, $N = 4$, $F = 1$, where process 1 is Byzantine faulty. Process 1 sends out syntactically correct but different messages to the non-faulty processes: $x = 0, y = 1$ to process 2; $x = 0, y = 0$ to process 3; $x = 1, y = 1$ to process 4. As shown in Figure 5.1, non-faulty processes 2, 3, 4 build different EIG trees, but they still reach the same final decision.

three EIG tress are shown completed all attribute values. Consider the EIG tree (b) owned by process 3, $T^3_{4,2}$. The α^3 values are filled from messages received in the two messaging rounds, as indicated in Figure 5.2.

The β^3 values are evaluated as required by the algorithm, by a local majority voting scheme. The evaluation of $\beta^3(\lambda)$ reaches a tie, on multiset $\{0, 0, 1, 1\}$, which has two 0's and two 1's; this tie is broken using the default value, here we assume $v_0 = 0$. Thus, $\beta^3(\lambda) = 0$ is the final decision of process 3, which is different from its initial choice, $\alpha^3(\lambda) = 1$.

A similar argument shows that all other non-faulty processes, 2 and 4, end with the same final decision, 0, thereby achieving the required agreement, despite starting with different initial choices and the conflicting messages sent by faulty process 1. Briefly, the Byzantine-faulty process may sometimes affect the outcome (between 0 and 1), but cannot affect the consensus: all non-faulty processes will take the same final decision.

5.1.3 Previous Models

Without going into details, in this subsection, we take a bird's eye view on the previous best solutions, Dinneen et al. (2010) [17], and Nicolescu (2016/2017) [49], briefly highlighting their merits and problems.

Dinneen et al. [17] was the first P system solution to the Byzantine agreement. In fact, it is not one single solution, but (as usual in the classical model of P systems), it is a *family* of related solutions, each one with its own *variable number of symbols and rules*; in this case, $\mathcal{O}(N!)$ symbols and rules (factorial!), where N is the number of processes. Moreover, to ensure a correct solution, protected even against Sybil

attacks, each process was modelled by a main cell surrounded by a constellation of $3N$ "firewall" cells, reduced to $2N$ in a subsequent version. Figures 5.3 and 5.4 highlights this network complexity, by illustrating the interconnections between processes #2 and #3 when $N = 4$.

Using the additional features offered by cP systems, Nicolescu (2016/2017) [49] was the first unitary solution (no family), with a *fixed number of symbols and rules*, not depending on N. However, without adequate control on the message input flow, it still needed Sybil protection by "firewall cells", this time only N per main top-cell. Figure 5.5 highlights this reduced network complexity, by illustrating all interconnections between all processes, for $N = 4$.

The novel solution proposed in this paper removes any need for such firewall artefacts and, in fact, improves the solution on all criteria. The new solution uses exactly N top-level cells, as in the conceptual model, illustrated in Figure 5.2.

5.1.4 The Actor-like Model

In this subsection, we give a complete ruleset for a top-level cell that models a non-faulty process in the EIG-based Byzantine algorithm of subsection 5.1.2. We illustrate this ruleset by traces for Byzantine process #2, used in Figures 5.1, 5.2.

Initial top-cell configuration

Subcells $\bar{\delta}(v)$, $v \in \{0, 1\}$ define the two admissible decision values. Subcell $\bar{v_0}(v)$ contains $v = v_0 \in \{0, 1\}$, the default value known by all processors. Subcell $\bar{\ell}(L)$ represents the maximum number of levels of the EIG tree, precomputed as $L = F + 1 = \lfloor (N + 2)/3 \rfloor$. Subcells $\bar{\pi}(i)$, $i \in \{1, 2, ..., N\}$, define the set of all process ID's.

Additionally, each cell $i \in \{1, 2, ..., N\}$, knows its own initial ID, given by subcell $\bar{\mu}(i)$. Subcell $\bar{\alpha}(v)$ contains $v = v_i \in \{0, 1\}$, the initial choice of process $\#i$. Figure 5.6 shows the initial contents of top-level cell #2.

Sending Messages

The rules for sending messages are given in Figure 5.9. Rule (1) completes the initialisation, creating the root of the EIG tree, where: (i) subcell ℓ represents the current level in the EIG tree; and (ii) subcell θ represents a node in the EIG tree, containing the following sub-subcells:

- ℓ: the node level;
- α: the node value (attribute), α in the top-down phase and then β in the bottom-up phase;

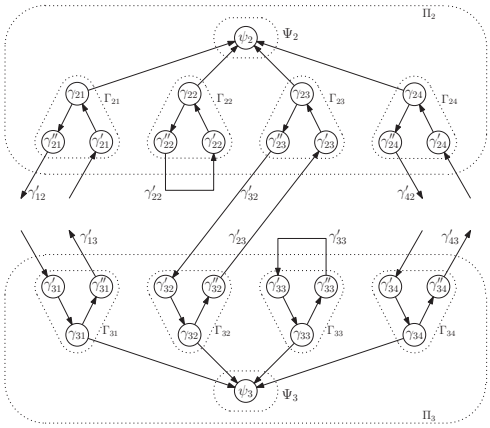


Figure 5.3: Two main cells and their firewalls in Dinneen et al. [17].

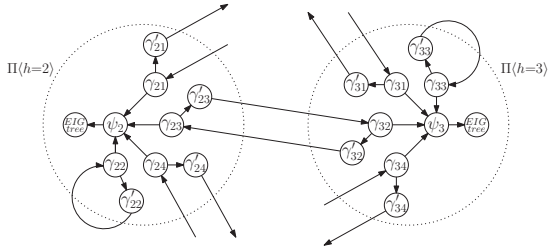


Figure 5.4: Two main cells and their firewalls in an updated version of Dinneen et al. [17].

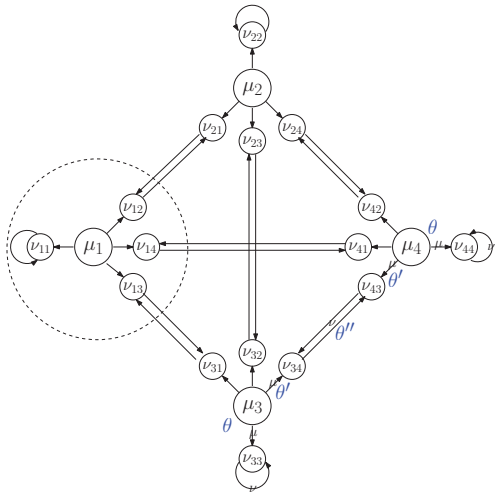


Figure 5.5: Main cells and their firewalls in Nicolescu [49].

$\bar{\delta}(0)$	$\bar{\delta}(1)$	$\bar{v_0}(0)$	$\bar{\ell}(2)$	$\bar{\pi}(1)$	$\bar{\pi}(2)$	$\bar{\pi}(3)$	$\bar{\pi}(4)$	$\bar{\mu}(2)$	$\bar{\alpha}(0)$

Figure 5.6: Initial configuration of top-cell #2, in state S_0.

- π: the list of EIG node labels (Byzantine process IDs) *from this node to the root*;
- ρ: the set image of the above π list;

Note that the above order for π *reverses* the order used in subsection 5.1.2, e.g. EIG node 3.2 is here represented via $\pi[2,3]$. Also, ℓ and ρ are not strictly needed, but allows us to quickly check the node's level and if a given ID is or is not in the list π. Figure 5.7 shows the contents of top-level cell #2, after the EIG root is created.

Rule (2) is the rule which stops the message sending rounds when the current level and the maximum number of levels are equal.

Rule (3) sends out messages to every cell (process), which is then stored in their corresponding input queues. The message contains an already incremented level, meaning it contains the level on which it should be stored. Each message is a θ' term, containing only the essential parts of the corresponding θ term, i.e. without, the redundant ρ subterm. The branch has also been indicated and the α value computed, based on its source tree branch. The final part of the rule is an inhibitor, which requires that the current process ID must not be present in the set of process ID's.

Rule (4) creates the templates for the next level of EIG nodes. The nodes are initialised with the default value, v_0. Rule (5) increments the level, so we are now at the next level. Figure 5.8 shows the contents of top-level cell #2, after sending round #1 messages; note the prepared templates for receiving round #1 messages, initialised with $v_0 = 0$.

Receiving messages

Rule (6) receives well-formed messages. This rule will "consume" the next level template EIG node that was created in Rule (4). Thus, the current templates match the well-formed messages received; non well-formed messages are not accepted, and the template remains with the default value, v_0, created by Rule (4).

Rule (7) makes the level above the current level take the default value. This means that all levels above the current level will contain the default value. Figure 5.10 shows the contents of top-level cell #2, after receiving round #1 messages.

The previous send and receive steps are repeated L times, where L is given by the contents of $\bar{\ell}(L)$. Figure 5.12 shows the contents of top-level cell #2, after sending round #2 messages. Figure 5.13 shows the contents of top-level cell #2, after receiving round #2 messages.

$\bar{\delta}(0)$	$\bar{\delta}(1)$	$\bar{v_0}(0)$	$\bar{\ell}(2)$	$\bar{\pi}(1)$	$\bar{\pi}(2)$	$\bar{\pi}(3)$	$\bar{\pi}(4)$	$\bar{\mu}(2)$
$\bar{\alpha}(0)$	$\ell(0)$	$\theta(\ell(0)\ \pi[\,]\ \rho()\ \alpha(\mathbf{0}))$						

Figure 5.7: Top-cell #2 in S_1, after the initialisation of rule (1).

$\bar{\delta}(0)$ $\bar{\delta}(1)$ $\bar{v_0}(0)$ $\bar{\ell}(2)$ $\bar{\pi}(1)$ $\bar{\pi}(2)$ $\bar{\pi}(3)$ $\bar{\pi}(4)$ $\bar{\mu}(2)$ $\bar{\alpha}(0)$
$\ell(1)$ $\theta(\ell(0)\ \pi[\,]\ \rho()\ \alpha(0))$
$\theta(\ell(1)\ \pi[1]\ \rho(1)\ \alpha(0))$ $\theta(\ell(1)\ \pi[2]\ \rho(2)\ \alpha(0))$
$\theta(\ell(1)\ \pi[3]\ \rho(3)\ \alpha(0))$ $\theta(\ell(1)\ \pi[4]\ \rho(4)\ \alpha(0))$

Figure 5.8: Top-cell #2 in S_2, after sending round #1 messages, i.e. broadcasting copies of $\theta'(\ell(1)\ \pi[2]\ \ \alpha(0))$.

$$
\begin{array}{lll}
S_0 & \to_1 & S_1\ \ell(0)\ \theta(\ell(0)\ \pi[\,]\ \rho()\ \alpha(V)) \\
 & & \mid \bar{\alpha}(V)
\end{array}
\qquad (1)
$$

$$
\begin{array}{lll}
S_1 & \to_1 & S_3 \\
 & & \mid \bar{\ell}(L) \\
 & & \mid \ell(L)
\end{array}
\qquad (2)
$$

$$
\begin{array}{lll}
S_1 & \to_+ & S_2\ !_\forall \{\theta'(\ell(L1)\ \pi[X|P]\ \alpha(V))\} \\
 & & \mid \bar{\mu}(X) \\
 & & \mid \ell(L) \\
 & & \mid \theta(\ell(L)\ \pi[P]\ \alpha(V)\ \rho(Z)) \\
 & & \neg\ (Z = XQ')
\end{array}
\qquad (3)
$$

$$
\begin{array}{lll}
S_1 & \to_+ & S_2\ \theta(\ell(L1)\ \pi[X|P]\ \alpha(V)) \\
 & & \mid \ell(L) \\
 & & \mid \bar{\pi}[X] \\
 & & \mid \bar{v_0}(V) \\
 & & \mid \theta(\ell(L)\ \pi[P]\ \alpha(\_)\ \rho(Z)) \\
 & & \neg\ (Z = XQ')
\end{array}
\qquad (4)
$$

$$
S_1\ \ell(L) \quad \to_1 \quad S_2\ \ell(L1) \qquad (5)
$$

Figure 5.9: Ruleset for sending messages.

$\bar{\delta}(0)$ $\bar{\delta}(1)$ $\bar{v_0}(0)$ $\bar{\ell}(2)$ $\bar{\pi}(1)$ $\bar{\pi}(2)$ $\bar{\pi}(3)$ $\bar{\pi}(4)$ $\bar{\mu}(2)$ $\bar{\alpha}(0)$
$\ell(1)$ $\theta(\ell(0)\ \pi[\,]\ \rho()\ \alpha(\mathbf{0}))$
$\theta(\ell(1)\ \pi[1]\ \rho(1)\ \alpha(\mathbf{0}))$ $\theta(\ell(1)\ \pi[2]\ \rho(2)\ \alpha(\mathbf{0}))$
$\theta(\ell(1)\ \pi[3]\ \rho(3)\ \alpha(\mathbf{1}))$ $\theta(\ell(1)\ \pi[4]\ \rho(4)\ \alpha(\mathbf{1}))$

Figure 5.10: Top-cell #2 in S_1, after receiving round #1 messages: here $\theta'(\ell(1)\ \pi[1]\ \ \alpha(0))$ from #1, $\theta'(\ell(1)\ \pi[2]\ \ \alpha(0))$ from #2 (self), $\theta'(\ell(1)\ \pi[3]\ \ \alpha(1))$ from #3, $\theta'(\ell(1)\ \pi[4]\ \ \alpha(1))$ from #4.

$$
\begin{array}{lll}
S_2\ ?_Y\ \{\theta'(\ell(L1)\ \pi[Y|P]\ \alpha(V))\} & \rightarrow_+ & S_1\ \theta(\ell(L1)\ \pi[Y|P]\ \rho(YQ)\ \alpha(V)) \\
\theta(\ell(L1)\ \pi[Y|P]\ \alpha(\_)) & & \mid \theta(\ell(L)\ \pi[P]\ \rho(Q)\ \alpha(\_)) \\
 & & \mid (Q \neq YQ') \\
 & & \mid \bar{\delta}(V)
\end{array} \quad (6)
$$

$$
\begin{array}{lll}
S_2\ \theta(\ell(L)\ \pi[X|P]\ \alpha(\_)) & \rightarrow_+ & S_1\ \theta(\ell(L)\ \pi[X|P]\ \alpha(V)) \\
 & & \mid \ell(L1) \\
 & & \mid \bar{v_0}(V)
\end{array} \quad (7)
$$

Figure 5.11: Ruleset for receiving messages.

$\bar{\delta}(0)$	$\bar{\delta}(1)$	$\bar{v_0}(0)$	$\bar{\ell}(2)$	$\bar{\pi}(1)$	$\bar{\pi}(2)$	$\bar{\pi}(3)$	$\bar{\pi}(4)$	$\bar{\mu}(2)$	$\bar{\alpha}(0)$
$\ell(2)$	$\theta(\ell(0)\ \pi[\,]\ \rho()\ \alpha(0))$								
$\theta(\ell(1)\ \pi[1]\ \rho(1)\ \alpha(0))$			$\theta(\ell(1)\ \pi[2]\ \rho(2)\ \alpha(0))$						
$\theta(\ell(1)\ \pi[3]\ \rho(3)\ \alpha(1))$			$\theta(\ell(1)\ \pi[4]\ \rho(4)\ \alpha(1))$						
$\theta(\ell(2)\ \pi[2,1]\ \rho(2,1)\ \alpha(0))$			$\theta(\ell(2)\ \pi[3,1]\ \rho(3,1)\ \alpha(0))$						
			$\theta(\ell(2)\ \pi[4,1]\ \rho(4,1)\ \alpha(0))$						
$\theta(\ell(2)\ \pi[1,2]\ \rho(1,2)\ \alpha(0))$			$\theta(\ell(2)\ \pi[3,2]\ \rho(3,2)\ \alpha(0))$						
			$\theta(\ell(2)\ \pi[4,2]\ \rho(4,2)\ \alpha(0))$						
$\theta(\ell(2)\ \pi[1,3]\ \rho(1,3)\ \alpha(0))$			$\theta(\ell(2)\ \pi[2,3]\ \rho(2,3)\ \alpha(0))$						
			$\theta(\ell(2)\ \pi[4,3]\ \rho(4,3)\ \alpha(0))$						
$\theta(\ell(2)\ \pi[1,4]\ \rho(1,4)\ \alpha(0))$			$\theta(\ell(2)\ \pi[2,4]\ \rho(2,4)\ \alpha(0))$						
			$\theta(\ell(2)\ \pi[3,4]\ \rho(3,4)\ \alpha(0))$						

Figure 5.12: Top-cell #2 in S_2, after sending round #2 messages, i.e. broadcasting copies of $\theta(\ell(2)\ \pi[2,1]\ \alpha(0))$, $\theta(\ell(2)\ \pi[2,3]\ \alpha(1))$, $\theta(\ell(2)\ \pi[2,4]\ \alpha(1))$.

$\bar{\delta}(0)$	$\bar{\delta}(1)$	$\bar{v_0}(0)$	$\bar{\ell}(2)$	$\bar{\pi}(1)$	$\bar{\pi}(2)$	$\bar{\pi}(3)$	$\bar{\pi}(4)$	$\bar{\mu}(2)$	$\bar{\alpha}(0)$
$\ell(2)$	$\theta(\ell(0)\ \pi[\,]\ \rho()\ \alpha(0))$								
$\theta(\ell(1)\ \pi[1]\ \rho(1)\ \alpha(\mathbf{0}))$			$\theta(\ell(1)\ \pi[2]\ \rho(2)\ \alpha(\mathbf{0}))$						
$\theta(\ell(1)\ \pi[3]\ \rho(3)\ \alpha(\mathbf{0}))$			$\theta(\ell(1)\ \pi[4]\ \rho(4)\ \alpha(\mathbf{0}))$						
$\theta(\ell(2)\ \pi[2,1]\ \rho(2,1)\ \alpha(\mathbf{0}))$			$\theta(\ell(2)\ \pi[3,1]\ \rho(3,1)\ \alpha(\mathbf{0}))$						
			$\theta(\ell(2)\ \pi[4,1]\ \rho(4,1)\ \alpha(\mathbf{1}))$						
$\theta(\ell(2)\ \pi[1,2]\ \rho(1,2)\ \alpha(\mathbf{0}))$			$\theta(\ell(2)\ \pi[3,2]\ \rho(3,2)\ \alpha(\mathbf{0}))$						
			$\theta(\ell(2)\ \pi[4,2]\ \rho(4,2)\ \alpha(\mathbf{0}))$						
$\theta(\ell(2)\ \pi[1,3]\ \rho(1,3)\ \alpha(\mathbf{1}))$			$\theta(\ell(2)\ \pi[2,3]\ \rho(2,3)\ \alpha(\mathbf{1}))$						
			$\theta(\ell(2)\ \pi[4,3]\ \rho(4,3)\ \alpha(\mathbf{1}))$						
$\theta(\ell(2)\ \pi[1,4]\ \rho(1,4)\ \alpha(\mathbf{1}))$			$\theta(\ell(2)\ \pi[2,4]\ \rho(2,4)\ \alpha(\mathbf{1}))$						
			$\theta(\ell(2)\ \pi[3,4]\ \rho(3,4)\ \alpha(\mathbf{1}))$						

Figure 5.13: Top-cell #2 in S_1, after receiving round #2 messages.

Second Phase

The second phase corresponds to the bottom-up evaluation of the EIG tree. Rule (8) corresponds to when the bottom-up evaluation has finished, in which case the final value (which is shared by all non-faulty processors) will be stored in ω.

Rule (9) decrements the level, so moves up the evaluation tree. Rule (9) removes all pairs of opposite values at the leaves. Rule (10) takes the value of the remaining leaves "after" (actually done in parallel, but can be simply viewed in this way, due to the weak priority order.) Rule (9). Rule (11) removes the leaves of the bottom-up evaluation. Rule (12) decrements the current level corresponding to moving up the evaluation tree. Figures 5.15 and 5.16 show the contents of top-level cell #2, after the first, respectively the second, bottom-up evaluation. Figure 5.17 shows the final contents of top-level cell #2, with its final decision in ω.

5.1.5 Conclusions

To the best of our knowledge, the current model is the first P system model that maps each Byzantine process to exactly one cell, thus solving the open problem mentioned in [17, 49]. As Figures 5.3, 5.4 and 5.5 show, all our previous models required additional helper/firewall cells around the main cells, as protection against Sybil attacks. The new Actor-based model is much more robust and does not require such additional external protection.

Table 5.1 compares the previous models [17] (2010), [49] (2016), and the current model (2018), on several complexity measures: (i) several static complexity measures (first rows); and (ii) the runtime steps (the last two rows).

As Table 5.1 highlights, the new Actor-based model substantially improves all considered measures. The raw size is the size of an ASCII text file containing the ruleset, described in LaTeX, but without any layout indications. The compressed size is the raw file size after compression with 7-Zip, with the LZMA2 compression method and Ultra compression level. The firewall rulesets have been included only once, without considering the code replication, $3N$ or N firewall cells per main Byzantine cell. As the files are relatively small, the compressed size is biased by the included compression dictionary, but nevertheless, the results could be interesting.

The size of the compressed ruleset is intuitively related to the information content of the ruleset. Interestingly, all considered models have not very different information contents, which is understandable, as they all model the same problem. The raw sizes are, however, more different. The raw/compressed ratio is intuitively a measure of the ruleset expressivity: the lower the ratio, the better: less noise/redundancy and

[a] $2N + 1$ in later versions.
[b] Cf. Figures 5.9, 5.11, and 5.14.
[c] Can be reduced to 1.

$$S_3\ \ell()\ \theta(\ell()\ \pi[\,]\ \alpha(V)) \quad \rightarrow_1 \quad S_4\ \omega(V) \tag{8}$$

$$S_3\ \theta(\ell(L1)\ \pi[\_|P]\ \alpha(1)) \quad \rightarrow_+ \quad S_3 \tag{9}$$
$$\theta(\ell(L1)\ \pi[\_|P]\ \alpha(0)) \qquad\qquad | \ \ell(L1)$$

$$S_3\ \theta(\ell(L1)\ \pi[\_|P]\ \alpha(X)) \quad \rightarrow_+ \quad S_3\ \theta(\ell(L)\ \pi[P]\ \alpha(X)) \tag{10}$$
$$\theta(\ell(L)\ \pi[P]\ \alpha(\_)) \qquad\qquad | \ \ell(L1)$$

$$S_3\ \theta(\ell(L1)_-) \quad \rightarrow_+ \quad S_3 \tag{11}$$
$$\qquad\qquad | \ \ell(L1)$$

$$S_3\ \ell(L1) \quad \rightarrow_1 \quad S_3\ \ell(L) \tag{12}$$

Figure 5.14: Ruleset for evaluating the EIG tree.

$\bar{\delta}(0)$	$\bar{\delta}(1)$	$\bar{v_0}(0)$	$\bar{\ell}(2)$	$\bar{\pi}(1)$	$\bar{\pi}(2)$	$\bar{\pi}(3)$	$\bar{\pi}(4)$	$\bar{\mu}(2)$	$\bar{\alpha}(0)$
$\ell(1)$	$\theta(\ell(0)\ \pi[\,]\ \rho()\ \alpha(0))$								
$\theta(\ell(1)\ \pi[1]\ \rho(1)\ \alpha(\mathbf{0}))$		$\theta(\ell(1)\ \pi[2]\ \rho(2)\ \alpha(\mathbf{0}))$							
$\theta(\ell(1)\ \pi[3]\ \rho(3)\ \alpha(\mathbf{1}))$		$\theta(\ell(1)\ \pi[4]\ \rho(4)\ \alpha(\mathbf{1}))$							

Figure 5.15: Top-cell #2 in S_3, after first bottom up evaluation.

$\bar{\delta}(0)$	$\bar{\delta}(1)$	$\bar{v_0}(0)$	$\bar{\ell}(2)$	$\bar{\pi}(1)$	$\bar{\pi}(2)$	$\bar{\pi}(3)$	$\bar{\pi}(4)$	$\bar{\mu}(2)$	$\bar{\alpha}(0)$
$\ell(0)$	$\theta(\ell(0)\ \pi[\,]\ \rho()\ \alpha(\mathbf{0}))$								

Figure 5.16: Top-cell #2 in S_3, after second bottom up evaluation.

$\bar{\delta}(0)$	$\bar{\delta}(1)$	$\bar{v_0}(0)$	$\bar{\ell}(2)$	$\bar{\pi}(\iota_1)$	$\bar{\pi}(\iota_2)$	$\bar{\pi}(\iota_3)$	$\bar{\pi}(\iota_4)$	$\bar{\mu}(\iota_2)$	
$\bar{\alpha}(0)$	$\omega(\mathbf{0})$								

Figure 5.17: Top-cell #2 in S_4, with final value in ω.

Table 5.1: Summary of complexity measures (where $L = \lfloor (N+2)/3 \rfloor$).

Measure	**[17] (2010)**	**[49] (2016)**	**This model (2018)**
Cells per process	$3N + 1$[a]	$N + 1$	**1**
Atomic symbols	$\mathcal{O}(N!)$	18	14
States	$\mathcal{O}(L)$	14	5
Rules	$\mathcal{O}(N!)$	23	12[b]
Ruleset size – Raw	2338	2218	1481
Ruleset size – Compressed	624	591	526
Raw/Compressed ratio	3.75	3.75	2.81
Steps per top-down level	5	4	2
Steps per bottom-up level	1	3[c]	1

higher information density. The new Actor-based model seems the most expressive: it has the highest information density with the least noise.

To the best of our knowledge, the new model also compares favourably with any extant description – whether informal, pseudocode, or code – of the classical EIG tree and algorithm: (i) it is fully formal; (ii) it is directly executable; and (iii) it is crisper (without relying on any previously developed library). The reader is invited to compare our formal description to other, even informal descriptions, such as given in classical textbooks, e.g. Lynch [40], pages 108, 109, 120.

5.2 Santa Claus Problem

In this section we discuss the Santa Claus Problem, a classic and challenging concurrency problem, and propose a slightly tighter specification, that precludes a few odd scenarios. We provide an elegant solution to this refined problem using cP systems. We now evaluate, for the first time, our cP systems as a concurrency specification language. We compare our cP specification and similar solutions implemented in several modern languages (e.g. Go, C#, F#), with respect to program-size complexity and (where available) runtime performance.

As it is now often repeated, the free lunch is over—we can no longer make our code run faster by simply waiting for faster hardware. Modern high-performance systems need to exploit concurrency, yet writing concurrent code can easily introduce bugs and complexities, making people spend more time worrying about and satisfying the computer rather than solving the domain problem.

The Santa Claus Problem is a classic concurrency challenge, introduced by Trono [66] in 1994:

> Santa Claus sleeps at the North Pole until awakened by either all of the nine reindeer, or by a group of three out of n elves. He then performs one of two indivisible actions:
>
> 1. If awakened by the group of nine reindeer, Santa harnesses them to the sleigh, delivers toys, and finally unharnesses the reindeer, who then go on vacation.
> 2. If awakened by a group of three elves, Santa invites them into his office, consults with them on toy R&D, and finally shows them out, so they can return to work on constructing toys.
>
> In our refined specification, we strengthen the rules by assuming that each elf will ask one question and get one response from Santa.
>
> As in the classical version, a waiting group of reindeer must be served by Santa before a waiting group of elves. Since Santa's time is precious, marshalling the reindeer or elves into a group must not be done by Santa.

The problem seems simple at first sight; however, it has become a tough testbed for the expressiveness of concurrency constructs. As noted by Benton [6], attempts to solve this problem can easily introduce errors such as:

- Santa takes off to deliver toys, while one or more (possibly all nine) reindeer are still be waiting to be harnessed.
- Santa takes off to deliver toys, after one or more (possibly all nine) reindeer have already gone on vacation once again.
- Santa ends his consulting time and goes to sleep, while one or more (possibly all three) elves are still waiting to ask their questions.
- Santa starts his consulting time, after one or more (possibly all three) elves have already left the office.
- One or more additional elves sneak into the consultation room after Santa invites the group of three who have initially registered for consultation.
- The priority rule is quite challenging to implement fully. There is frequently a (possibly very narrow, but not null) opportunity for a group of three elves to get Santa's attention even when all reindeer have returned and are ready to be harnessed.

Trono's own first solution [66], based on semaphores, highlighted all the above problems. Subsequent solutions tried to remedy this, but even recent solutions may exhibit some of the above problems, or else are excessively complex and fragile (qualifying them as 'messy' would not be a great exaggeration). While primitive constructs such as semaphores can successfully model simple scenarios, they seem less adequate for high-level models of complex scenarios.

Therefore, the Santa Claus Problem has become a great testbed for testing the expressiveness of numerous concurrency constructs and models, such as: semaphores,

monitors, locks, barriers, select-type constructs, guards, message passing (cf. MPI [21]), software transactional memory (STM) [35], actors and actor extensions (multiple heads, multiple mailboxes [63]), communicating sequential processes (CSP, cf. occam [44]), rendezvous (cf. Ada [5]), join calculus (cf. Polyphonic C# [6]), etc. The next subsection offers a brief overview of some of these proposals.

Essentially, there are two distinct problems that must be properly solved, cf. Ben-Ari [5]: (1) how to synchronise a set of processes (here elves or reindeer) and release them as a group; (2) how one process (e.g. Santa) provides more than one distinct service, with different priorities.

5.2.1 Overview

The shared memory model implemented in the form of threads is usually adopted as the first attempt to solve any concurrency challenges, since the thread libraries are conveniently available in many popular programming languages such as C/C++, Java, and C#. In the shared memory model, the program is split into two or more tasks running on different threads which operate on shared data, and the underlying operating system is in charge of scheduling and dispatching these threads for execution. The first solution to the Santa Claus Problem given by Trono utilises such construct with semaphores [66]. However, Trono's solution is only partially correct, mainly because it 'assumes that a process released from waiting on a semaphore will necessarily be scheduled for execution' [5]. Hurt and Pedersen [31] implemented and compared a few different solutions using barriers, locks, semaphores, mutexes, and monitors. However, all these constructs may suffer from various problems such as race conditions, deadlocks, and livelocks. Besides, it is a non-trivial task to prove the correctness, as all possible interleavings of program execution have to be considered.

Message Passing Interface (MPI) [21] is a message-passing standard for a distributed memory architecture e.g. a cluster. Most MPI implementations consist of a specific set of API directly callable from C\C++, Fortran and many languages able to interface with such libraries, including C#, Java or Python. In MPI, each program's task is separated onto a different process, mimicking a distributed memory model. The *MPI_Barrier* method can be used to synchronise a group of processes, and synchronous receives replaces wait/notify to simplify the asynchronous logic. These techniques can be used to synchronise the individual Santa, elf, and reindeer processes [31]. Due to the lack of shared memory in MPI, a separate process for each reindeer and elf waiting group is needed. When a process receives a piece of data, it processes it and then passes it onto one or multiple processes. Therefore the same piece of data is never operated on by more than one process at a time, eliminating the data race problem plaguing the shared memory model [31].

Jones [35] proposed another solution using the Software Transactional Memory (STM) model in Haskell. STM enables people to write programs in a more modular way; for example, building large programs by gluing smaller programs together, without

needing to expose their implementations [35]. The STM solution to the Santa Claus Problem introduces the abstraction of a *Gate* and a *Group* for modularity and ease of action coordination. Santa creates two groups, one for the elves and the other for the reindeer. The elves and the reindeer try to join their group if needing to wake Santa up. Santa controls the Gates for marshalling the elves, and the reindeer [35]. The shared memory model makes such modularity difficult: any newly added actions, which operate on the shared data, need to be carefully coordinated by synchronising constructs, such as locks and conditional variables, to avoid data races.

The actor model is gaining momentum in solving parallel and distributed problems, including concurrency challenges. It provides high-level concurrency abstractions via message passing. Each actor comes with an unbounded queue, called a mailbox, for storing any incoming messages. Actors communicate by sending and receiving messages. Sending is an asynchronous operation, meaning any actors can send any number of messages at a time. Receiving is typically a synchronous operation, meaning an actor can only process one message at a time, and it will logically block if the mailbox is empty. Receiving of a message is usually implemented by pattern matching. However, as noted by Sulzmann et al. [63], the classic actor model implemented in languages such as Erlang is restricted to a *single-headed* message pattern – that is, each receive operation only matches one message at a time, in the order of their mailbox position. This restriction makes the priority in the Santa Claus Problem a non-trivial task to implement, since the wake-up message coming from the nine reindeer might be preceded, in the mailbox, by a wake-up message coming from the three elves. To address this limitation, Sulzmann et al. [63] proposed and designed an extension of Erlang style actors with receive clauses containing multi-headed message patterns i.e. matching multiple messages at a time, extending the ability to easily express complex synchronisation patterns.

Ben-Ari [5] highlighted the errors in the original Trono's paper [66], discussed various problems that low-level semaphore-based solutions may have, and proposed a high-level solution in Ada. This solution uses Ada's unique *rendezvous* construct, inspired from join calculus. It also uses the more recent *protected object* construct, essentially an extension of Hanson's monitors. The solution is relatively neat and covers all requirements well, except the last and tricky one, to ensure that reindeer have priority over elves. This priority requires a subtle interaction between more sophisticated concepts, such as *requeuing* tasks, using additional accept (i.e. select) *guards*, and finally still needs a *pragma* requiring a textual order queuing policy.

Benton [6] proposed a neat solution, based on Polyphonic C#, a fairly direct implementation of join calculus ideas [7] in the C# programming language. It solves most problems by extensively using the new *chord* concept, which allows one to associate the header and body of one *synchronous* function with one or more *asynchronous* function headers. Calls to a chord only succeed when all functions have been called, i.e. the synchronous base and all additional asynchronous headers. Calling the synchronous entry proceeds synchronously, as expected, possibly *waiting* for all associ-

ated asynchronous calls. Calling an asynchronous entry does *not block*; if needed, the parameters of the call are implicitly *queued* in the system provided 'message' queue (like in actor systems). Space does not allow us to delve deeper into this topic here, but we illustrate these concepts by the straightforward implementation of a semaphore, cf. fig. 5.18. Still, unless the compiler and runtime ensure a textual order for chord matching, implementing the required priority is not simple and requires a few non-trivial additions.

5.2.2 Santa System

The cP system used to solve the Santa Claus Problem is described using five cell types:

- One Santa cell, denoted κ.
- One Office cell, denoted ω, where elves request to consult Santa.
- One Sleigh cell, denoted σ, where reindeer report their return.
- Nine Reindeer cells, denoted $\rho_1, \rho_2, ...\rho_9$.
- n Elf cells, denoted $\epsilon_1, \epsilon_2, ...\epsilon_n$.

An example graph of the system for two reindeer, and two elves is shown in fig. 5.19. A diagram of the message senders and receivers is shown in fig. 5.20. The system messages can be interpreted as:

- $\epsilon_i \xrightarrow{w} \epsilon_i$: Means that elf ϵ_i is *working*.

  Due to the system being *asynchronous*, the message w can take any length for the different elves. This is why all arbitrary length jobs are simulated using self-addressed messages.
- $\rho_i \xrightarrow{v} \rho_i$: Simulates the time reindeer ρ_i spends on *vacation*.
- $\kappa \xrightarrow{d} \kappa$: Simulates the time that Santa spends *delivering* the presents.
- $\epsilon_i \xrightarrow{p} \omega$: *Puzzled* elf ϵ_i requests the Office, to consult with Santa.
- $\omega \xrightarrow{w} \kappa$: The Office tells Santa to *wake up*, as a group of three elves, are ready to consult.
- $\kappa \xrightarrow{\iota} \omega$: Santa tells the Office to *invite* the group of elves for consultation.
- $\omega \xrightarrow{\iota} \epsilon_i$: The Office *invites* elf ϵ_i to consult with Santa.

```
public async Signal() & public void Wait() {}
```

Figure 5.18: Semaphore in Polyphonic C#.

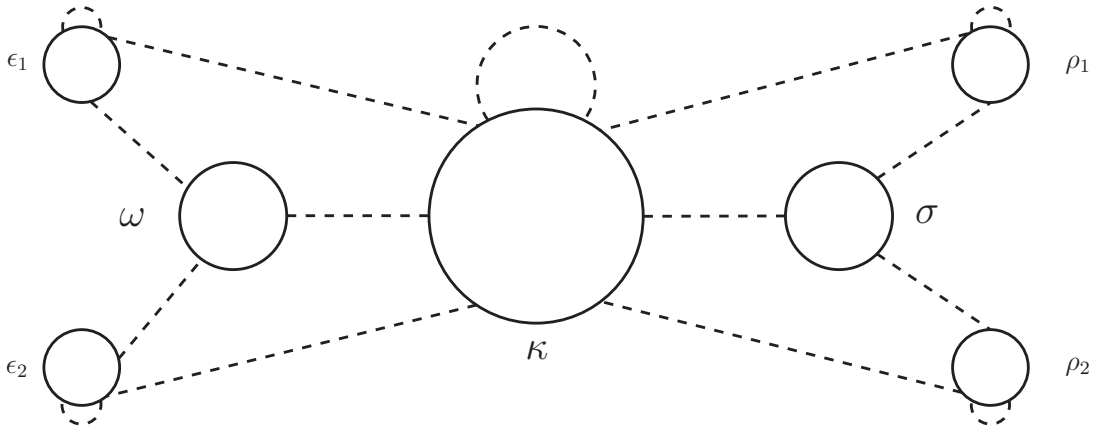


Figure 5.19: cP system graph, for two reindeer and two elves. The dashed lines represent connections between the cells.

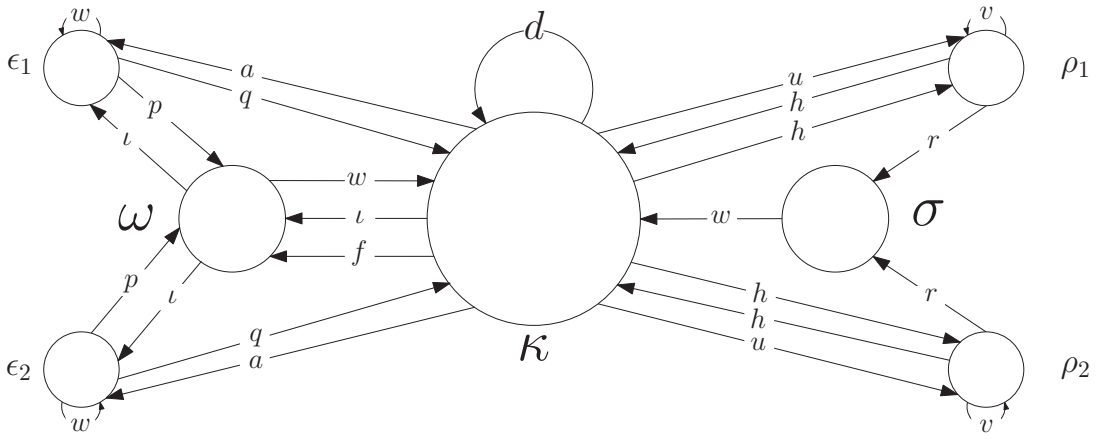


Figure 5.20: An overview of the cP system and messages sent.

- $\kappa \xrightarrow{f} \omega$: Santa tells the Office he is *finished* with the most recent group of elves.
- $\epsilon_i \xrightarrow{q} \kappa$: Elf ϵ_i asks Santa a *question*.
- $\kappa \xrightarrow{a} \epsilon_i$: Santa *answers* elf ϵ_i's question.
- $\rho_i \xrightarrow{r} \sigma$: Reindeer ρ_i tells the Sleigh they have *returned* from vacation.
- $\sigma \xrightarrow{w} \kappa$: The Sleigh tells Santa to *wake up*, as all nine reindeer have returned from vacation.
- $\kappa \xrightarrow{h} \rho_i$: Santa *harnessing* reindeer ρ_i.
- $\rho_i \xrightarrow{h} \kappa$: Reindeer ρ_i confirms it is *harnessed* and ready to pull.
- $\kappa \xrightarrow{u} \rho_i$: Santa tells reindeer ρ_i to *unharness* and go on vacation once again.

The rulesets for each cell type are in figs. 5.21 to 5.25, respectively, where each rule is numbered by the cell symbol followed by a dot ('.') and then the rule index. For example, the reindeer's second rule is listed as $\rho.2$.

s_0 $?_\sigma$ w		$\to_1$	s_1	$(\kappa.1)$
s_1		$\to_+$	s_2 $!_R$ h $\pi(9)$ $\mid$ $\rho(R)$	$(\kappa.2)$
s_2 $?_R$ h $\pi(X1)$		$\to_1$	s_2 $\pi(X)$	$(\kappa.3)$
s_2 $\pi()$		$\to_1$	s_2 $!_\kappa$ d	$(\kappa.4)$
s_2 $?_k$ d		$\to_1$	s_3	$(\kappa.5)$
s_3		$\to_+$	s_0 $!_R$ u $\mid$ $\rho(R)$	$(\kappa.6)$
s_0 $?_\omega$ w		$\to_1$	s_4 $!_\omega$ ι $\pi(3)$	$(\kappa.7)$
s_4 $?_E$ q $\pi(X1)$		$\to_1$	s_4 $!_E$ a $\pi(X)$	$(\kappa.8)$
s_4 $\pi()$		$\to_1$	s_0 $!_\omega$ f	$(\kappa.9)$

Figure 5.21: Ruleset for the Santa cell, κ.

s_0 $r(9)$		$\to_1$	s_0 $!_\kappa$ w $r()$	$(\sigma.1)$
s_0 $?_R$ r $r(X)$		$\to_1$	s_0 $r(X1)$	$(\sigma.2)$

Figure 5.22: Ruleset for the Sleigh, σ.

s_0	$\rightarrow_1$	$s_1\ !_I\ v \mid \theta(I)$	$(\rho.1)$
$s_1\ ?_I\ v$	$\rightarrow_1$	$s_2\ !_\sigma\ r$	$(\rho.2)$
$s_2\ ?_\kappa\ h$	$\rightarrow_1$	$s_3\ !_\kappa\ h$	$(\rho.3)$
$s_3\ ?_\kappa\ u$	$\rightarrow_1$	s_0	$(\rho.4)$

Figure 5.23: Ruleset for a generic reindeer cell, ρ_i (where i is it's ID).

s_0	$\rightarrow_1$	$s_1\ !_\kappa\ w \mid e(3)$	$(\omega.1)$
$s_0\ ?_E\ p\quad e(X)$	$\rightarrow_1$	$s_0\ \omega(E)\ \ e(X1)$	$(\omega.2)$
$s_1\ ?_\kappa\ \iota$	$\rightarrow_1$	s_2	$(\omega.3)$
$s_2\ \omega(E)$	$\rightarrow_+$	$s_3\ !_E\ \iota$	$(\omega.4)$
$s_3\ ?_\kappa\ f\quad e(X)$	$\rightarrow_1$	$s_0\ \ e()$	$(\omega.5)$

Figure 5.24: Ruleset for the Office, ω.

s_0	$\rightarrow_1$	$s_1\ !_I\ w \mid \theta(I)$	$(\epsilon.1)$
$s_1\ ?_I\ w$	$\rightarrow_1$	$s_2\ !_\omega\ p$	$(\epsilon.2)$
$s_2\ ?_\epsilon\ \iota$	$\rightarrow_1$	$s_3\ !_\kappa\ q$	$(\epsilon.3)$
$s_3\ ?_\kappa\ a$	$\rightarrow_1$	s_0	$(\epsilon.4)$

Figure 5.25: Ruleset for generic elf cell, ϵ_i (where i is it's ID).

Initial configuration We assume that each reindeer and elf cell contains one term $\theta(x)$, where x is the unique identifier of the cell (in our case numbers). We also assume that Santa contains terms $\rho(x_1), \rho(x_2), ...\rho(x_9)$, where x_i's are the unique reindeer identifiers.

Example

We illustrate the system evolution with a simple scenario with *two* reindeer and *two* elves. This does not fully describe all possible concurrency issues. However, it presents an intuitive description of the ruleset. See fig. 5.19 for the cP graph and fig. 5.20 for an overview of the message senders and receivers.

Initially, the elves are working, and the reindeer are on vacation; this is denoted in the rule set by rules $\epsilon.1$ and $\rho.1$. The time working and on vacation is represented by the time it takes for a self-addressed message to arrive. When an elf receives this self-addressed message w, he knows that he needs to consult with Santa and requests the Office cell to see Santa (rule $\epsilon.2$). Similarly, when a reindeer receives the self-addressed message v, the reindeer then know they are rested enough to start delivering presents and so report to the Sleigh cell (rule $\rho.2$).

Once the two elves have all told the office cell that they are ready, the office cell asks Santa to wake up and help (rule $\omega.1$). Santa then tells the office that he is ready to see the elves (rule $\kappa.7$). He also sets his question and answer counter to be three. The office cell then tells each of the elves in the group that Santa is ready for questions (rule $\omega.4$).

When an elf receives the message that Santa is ready for questions, they send a question to Santa (rule $\epsilon.3$). Santa answers, in turn, each of the elf's questions, decrements his question and answer counter until it becomes zero (rule $\kappa.8$). When an elf gets an answer, they then return to work (rules $\epsilon.4$ and $\epsilon.2$). When Santa's question and answer counter is zero, he tells the office that he is finished with the current group, and returns to sleeping (rule $\kappa.9$).

When all the reindeer have informed the sleigh cell that they are ready to deliver the presents, the sleigh tells Santa to wake up and deliver the presents (rule $\sigma.1$). Santa wakes up and then harnesses all of the reindeer (rules $\kappa.1$ and $\kappa.2$). When a reindeer is harnessed and ready, they confirm with Santa (rule $\rho.3$). When all of the reindeer are harnessed and ready to deliver, Santa delivers all of the presents (rule $\kappa.4$). Delivery time is represented by a self-addressed message. When Santa finishes delivery, he tells the reindeer to go on vacation (rules $\kappa.5$ and $\kappa.6$) and returns to sleep. When the reindeer receive the message to go on vacation, they do as advised (rules $\rho.4$ and $\rho.1$).

Possible pitfalls

As mentioned previously, there are six common pitfalls in solutions to the Santa Claus Problem.

- Santa takes off to deliver toys, while one or more (possibly all nine) reindeer are still waiting to be harnessed.
- Santa takes off to deliver toys, after one or more (possibly all nine) reindeer have already gone on vacation once again.

Our solution is that Santa sends a message (h) to all of the reindeer to get harnessed. The reindeer, once harnessed, confirm with a message (h). Santa does not start to deliver presents until he has received confirmation from all of the reindeer (i.e. has counted nine confirmation h's).

- Santa ends his consulting time and goes to sleep, while one or more (possibly all three) elves are still be waiting to ask their questions.

When Santa starts consulting with the elves, he starts a question and answer counter. Santa will not finish consulting until he has received and answered questions from the entire group (i.e. has counted three answers to elves' questions, a's).

- Santa starts his consulting time, after one or more (possibly all three) elves have already left the office.

When the elves start consulting, they must receive a message a from Santa, before they can return to work. Hence, each elf in the group will not return to work until Santa has started consulting.

- One or more additional elves sneak into the consultation room after Santa invites the group of three who have initially registered for a consultation.

Elves are invited into the consulting room by the Office cell (rule $\omega.4$). The Office cell is only able to add three elves to the set of elves before it changes state (rules $\omega.1$ and $\omega.2$), and no longer accepts more elves to the group.

- The priority rule is quite tough to fully implement. There is a frequent opportunity (possibly very narrow, but not null) for a group of three elves to get Santa's attention even when all reindeer have returned and are ready to be harnessed.

Although complex in many programming languages, implementing the priority in cP is achieved simply via the *weak priority order* of Santa's rules.

5.2.3 Experiments

Scalability

To test the suitability of our model, we translated our rules into C# code. We used the library Akka.Net to implement the required messaging between cells. We then tested our code against four different solutions. A Go channels solution and a C semaphores solution from [59] where the code was altered to fit our interpretation of the problem. A C# semaphore slim solution in which we used asynchronous semaphores. We tested the code on a computer that had 8GB of RAM and running on an Intel i5 7600. To test the scalability of the code, we had three different parameters:

1. Number of consultations per year: The number of times a group of three elves would consult with Santa before a year was allowed to end.
2. Number of elves.
3. Number of years: A year was defined as when the Sleigh woke up Santa. The Sleigh could not wake Santa until the current year had, at least, the minimum number of consultations for a year.

The results of these experiments are presented in Subsection 5.2.5. The results demonstrate that the traditional C semaphores are not as effective in this scenario as the more modern approach of the async semaphore. We also note, that although our simulation code is not as efficient as the other solutions, it does demonstrate the effectiveness of cP systems to model inherently parallel problems.

Complexity of Code

To get an indication of the complexity of our solution compared to the existing solution, we used three different measures presented in Table 5.2.

1. Source lines of code (SLOC). SLOC is one of the major inputs into cost estimation models as stated in [47], among various program-size complexity measures (e.g. [16]). Hence, we use this as an estimation of the cost of the solutions.
2. Number of tokens can be taken roughly as the complexity of the solution. The fewer tokens used means that the solution is more compressed and has fewer irrelevant symbols.
3. The raw/compressed ratio is intuitively a measure of the expressiveness: the lower the ratio, the better: less noise/redundancy and higher information density.

To measure the SLOC for the cP solution, we used the number of rules as this roughly corresponds to the number of lines of code. For the other solutions, we took the number of lines based on the number of semicolons and open curly braces and subtracted when a for loop added more than necessary semicolons. This is to remove ambiguity when a solution may place multiple operations on one line and try and make fair comparisons. We can see that our cP solution uses around a third as many lines as the next closest.

To measure the number of tokens of the cP solution, we counted them manually, to give a rough estimate of the number of tokens required. We tokenised each of the keywords variables and operations for the other solutions and removed any unnecessary spaces. When the solutions are tokenised, we can see that our solution compares favourably to the others, with ours using around half as many.

We used the Latex code without formatting as the raw file to measure the raw/-compressed ratio for the cP solution. For the other solutions, we used the source code files. We compressed all of the files using 7-Zip, with the LZMA2 compression method and Ultra compression level. We see that the cP solution has the lowest ratio suggesting the higher information density. We also see that the C# simulation of the cP solution has a high ratio. This difference is expected as Akka .Net is simply an actor framework, not a cP system simulator; hence, making it simulate the cP system adds redundancy.

5.2.4 Conclusions

We think that our cP solution compares very favourably with other extant solutions, in terms of elegance and conciseness. We should also stress that our system is completely formal, directly executable (in principle), and—in contrast to all other solutions—is self-contained (does not require specialised libraries). From [31] summarises a simple but useful complexity measure using the number of lines of code. In our case, we have a total of 24 rules, roughly corresponding to 24 SLOC.

Last but not least, this exercise suggests that complex concurrency problems should be verified with formal methods or tools. The Santa Claus Problem highlights how easily one (even experts) can make mistakes while modelling complex concurrency scenarios; convincing intuitions and explanations are not enough. This is a theme for further research.

5.2.5 Tables and Graphs

Table 5.2: A comparison of different complexity measures for a selection of solutions to the Santa Claus problem.

	SLOC	Number of tokens	Raw size of file (bytes)	Size of compressed (bytes)	Raw/Compressed ratio
C# (semaphore slim)	210	1147	14,700	2,610	5.63
C (semaphore)	347	2552	12,725	2,612	4.87
C# (Akka.Net)	209	1470	7,696	1,584	4.85
C# (channels)	120	1015	5,297	1,232	4.3
Kotlin(channels)	121	729	5,020	1,184	4.24
F#(mailbox)	112	861	5,006	1,233	4.06
Go (channels)	114	717	3,941	1,035	3.81
F#(Hopac)	100	681	3,644	972	3.75
JS(channels)	114	717	3,861	1,034	3.73
Latex (cP system)	≈ 24	≈ 317	1,078	467	2.26

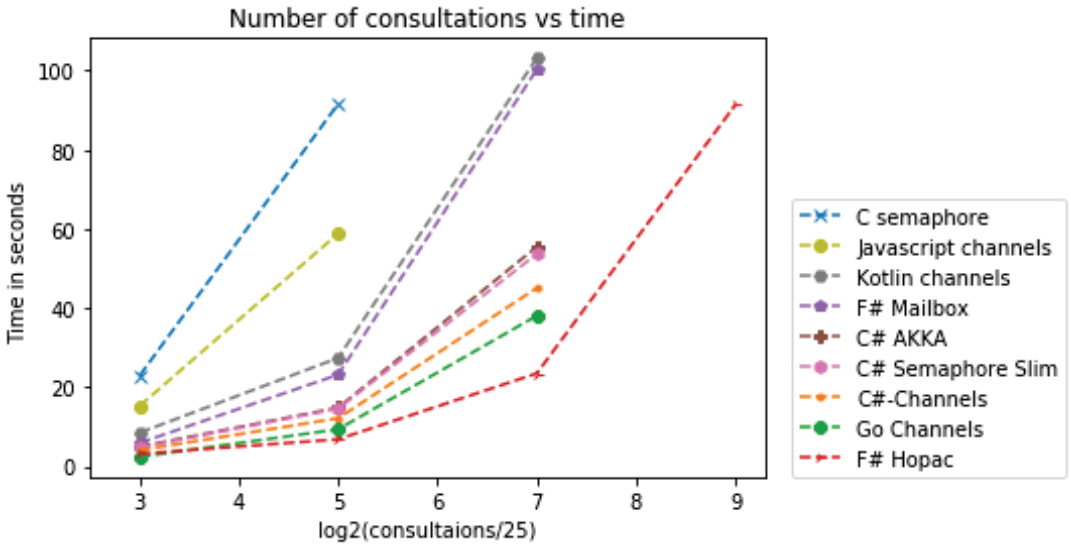


Figure 5.26: Number of consultations vs time taken (seconds).

Table 5.3: Number of consultations vs time taken (seconds).

Name	200	800	3200	12800
C semaphore	22.92	91.61	NA	NA
Javascript channels	15.28	58.83	NA	NA
Kotlin channels	8.53	27.41	103.18	NA
F# Mailbox	5.95	23.19	100.28	NA
C# AKKA	5.02	14.86	55.5	NA
C# Semaphore Slim	4.72	14.61	53.69	NA
C#-Channels	4.1	12.18	45.07	NA
Go Channels	2.41	9.39	38.08	NA
F# Hopac	3.2	6.86	23.5	91.36

Table 5.4: Number of consultations vs time taken (seconds).

Name	50	100	200
C semaphore	5.78	11.56	22.92
Javascript channels	9.09	7.76	15.28
Kotlin channels	3.99	5.07	8.53
F# Mailbox	1.58	3.02	5.95
C# AKKA	2.68	3.36	5.02
C# Semaphore Slim	2.33	3.09	4.72
C#-Channels	2.08	2.73	4.1
F# Hopac	1.89	2.27	3.2
Go Channels	1.81	1.23	2.41

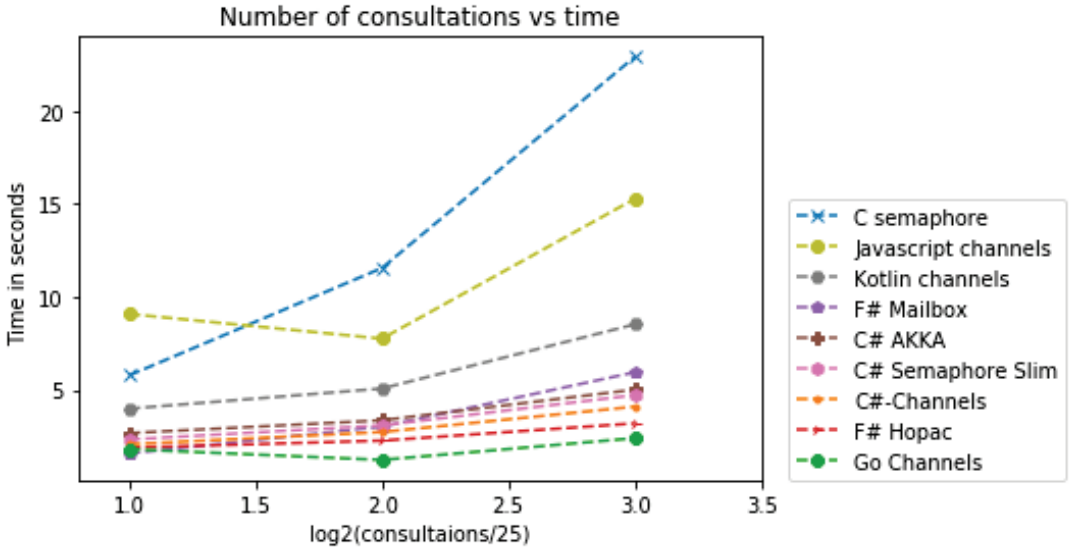


Figure 5.27: Number of consultations vs time taken.

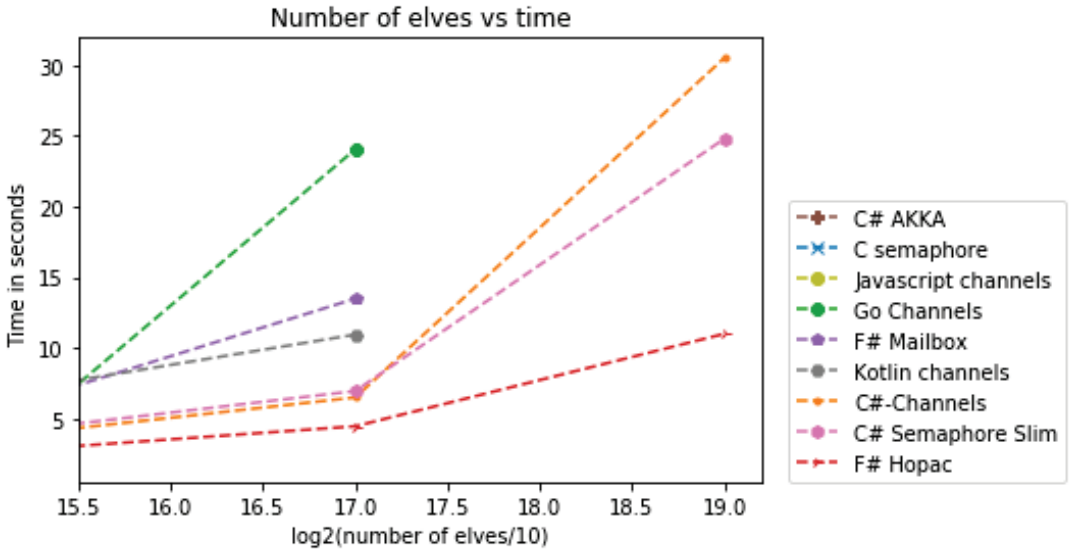


Figure 5.28: Number of elves vs time taken (seconds).

Table 5.5: Number of elves vs time taken.

Name	327680	1310720	5242880
C# AKKA	NA	NA	NA
C semaphore	NA	NA	NA
Javascript channels	NA	NA	NA
Go Channels	1.95	24.06	NA
F# Mailbox	5.35	13.49	NA
Kotlin channels	6.66	10.95	NA
C#-Channels	3.62	6.5	30.53
C# Semaphore Slim	3.89	6.96	24.81
F# Hopac	2.61	4.46	11.0

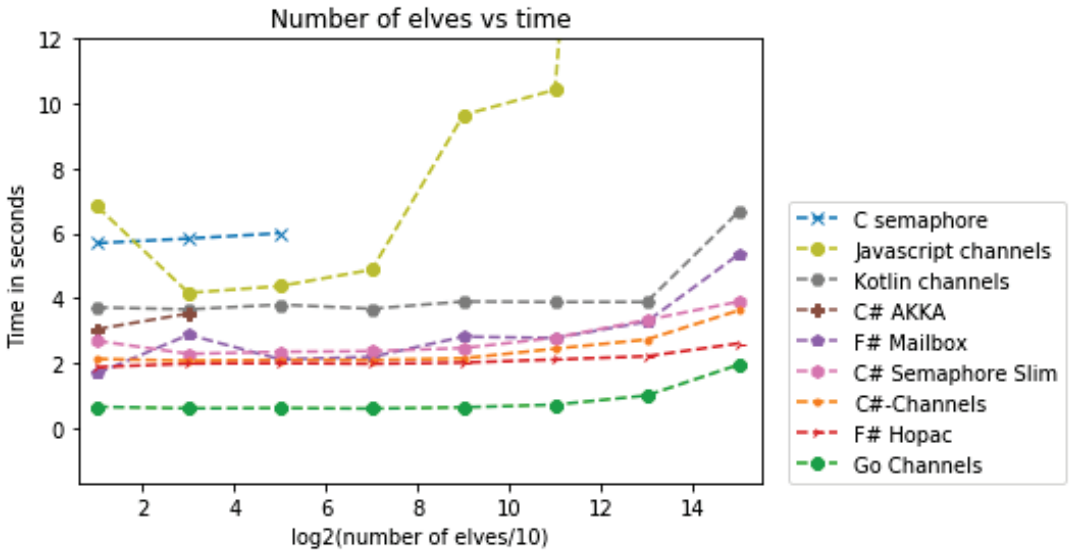


Figure 5.29: Number of elves vs time taken (seconds).

Table 5.6: Number of elves vs time taken (seconds).

Name	20	80	320	1280	5120	20480	81920	327680
C semaphore	5.7	5.84	6.01	NA	NA	NA	NA	NA
Javascript channels	6.84	4.17	4.38	4.88	9.65	10.43	46.43	NA
Kotlin channels	3.72	3.66	3.8	3.68	3.9	3.89	3.89	6.66
C# AKKA	3.05	3.53	NA	NA	NA	NA	NA	NA
F# Mailbox	1.68	2.88	2.12	2.19	2.82	2.77	3.28	5.35
C# Semaphore Slim	2.69	2.29	2.35	2.37	2.47	2.79	3.33	3.89
C#-Channels	2.13	2.07	2.1	2.1	2.15	2.45	2.73	3.62
F# Hopac	1.88	1.99	2.0	1.98	2.02	2.11	2.21	2.61
Go Channels	0.66	0.61	0.62	0.6	0.64	0.72	1.0	1.95

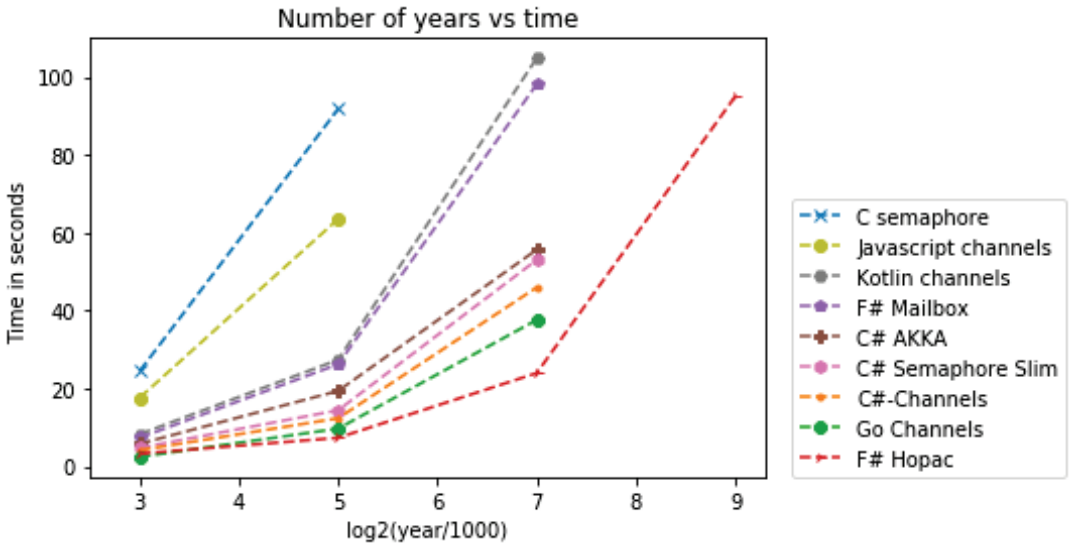


Figure 5.30: Number of years vs time taken (seconds).

Table 5.7: Number of years vs time taken (seconds).

Name	8000	32000	128000	512000
C semaphore	24.43	91.76	NA	NA
Javascript channels	17.59	63.5	NA	NA
Kotlin channels	8.24	27.36	104.82	NA
F# Mailbox	7.31	26.21	98.11	NA
C# AKKA	5.79	19.37	55.64	NA
C# Semaphore Slim	4.74	14.4	53.01	NA
C#-Channels	4.14	12.34	45.95	NA
Go Channels	2.41	9.69	37.6	NA
F# Hopac	3.28	7.33	23.96	95.03

Table 5.8: Number of years vs time taken (seconds).

Name	2000	4000	8000
C semaphore	6.28	12.49	24.43
Javascript channels	27.05	8.08	17.59
Kotlin channels	4.52	5.47	8.24
F# Mailbox	5.26	4.7	7.31
C# AKKA	15.15	4.29	5.79
C# Semaphore Slim	2.59	3.25	4.74
C#-Channels	2.17	2.78	4.14
F# Hopac	3.39	2.36	3.28
Go Channels	8.39	1.11	2.41

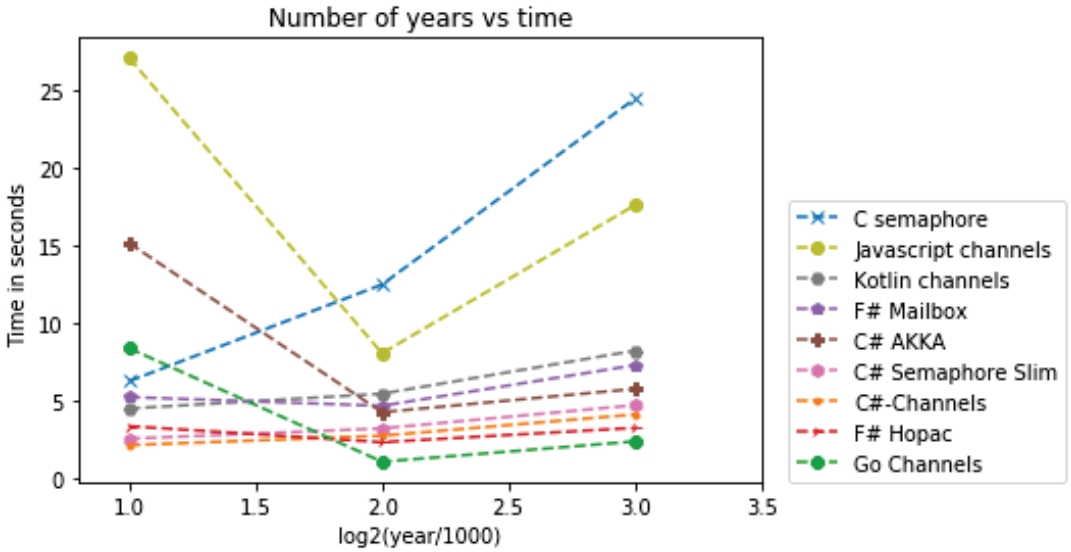


Figure 5.31: Number of years vs time taken (seconds).

5.3 Conclusions and Future work

In this chapter we have looked at how cP systems can efficiently model both distributed and parallel computing problems. We first showed using the recently proposed actor like controls on messages that our cP system outperforms the previous P system solutions to the Byzantine agreement problem on not only number of steps but also number of cells and code complexity.

We then showed how cP systems could model the Santa Claus problem demonstrating that cP systems can be used as a parallel computing specification language. We then show how our new solution can be utilised and compared to extant solutions. We see that our solution has far fewer lines of code than that of traditional programming languages.

Future work in this direction includes: Looking at modelling more sophisticated consensus algorithms (non EIG based), which offer better messaging performance and are actually used in large scale critical applications, such as Blockchains, investigating the uses of cP systems to more practical applications, and formal verification of cP systems which will be very useful on tricky problems such as the Santa Claus problem.

Chapter 6

Conclusions

This thesis has looked at the computational power of both cP systems and our newly proposed water-based system (as discussed, the constructions in the water section can be used with cP systems).

We started by looking at what cP systems can achieve with a single top-level cell. We showed for the first time that cP systems can solve PSPACE complete problems. Furthermore, 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. Whereas, as shown in Table 3.1, the previous confluent P system solutions use a variable number of rules, alphabet symbols and membrane nesting depth.

After this, we then solved NP-complete problems in sublinear time. We showed that the reduction from k-colouring to k-SAT can be made in constant time. This reduced solution surpassed the already proposed problem-specific cP systems solution [13] (a linear solution $\mathrm{O}(n)$) with our solution running in sublinear time($\mathrm{O}(\sqrt{n})$).

We then looked at water computing, in which we first proposed an improved modular design, which duplicates the main water flows by associated control flows. We solved the three open problems of the previous design [30] by demonstrating: how functions can be stacked without a combinatorial explosion of valves; how the termination of the system can be detected; and how to reset the system. We proved that the system is Turing complete by modelling the construction of μ-recursive functions. Finally, we demonstrated that this model can construct 'efficiently': 1) A programmable sequential, random-access machine (RAM), which we then extend to construct: 2) a programmable exclusive read exclusive write (EREW) parallel random-access machine (PRAM).

We finally looked at multi-cell cP systems that can be used to model parallel or distributed algorithms. We started by solving the Byzantine agreement problem using

the recently proposed actor controls on messages. To the best of our knowledge, our new solution compares favourably to the previous P system solutions as seen in Table 5.1. We then solved the Santa Claus problem using cP systems. Previously, cP systems have been successfully used as a specification language for a wide variety of problems. We evaluated, for the first time, our cP systems as a concurrency specification language. We compared our cP specification and similar solutions implemented in several modern languages (e.g. Go, C#, F#). Our cP solution compares very favourably with other extant solutions, in terms of elegance and conciseness (approximately a quarter of the SLOC of the solutions we compared against as seen in Table 5.2).

Both our solutions for the Santa claus and Byzantine agreement problem compare very favourably to the extant solutions whether informal, pseudocode, or code. Our solutions (i) are fully formal; (ii) directly executable; and (iii) crisper (without relying on any previously developed library).

Future work in this area could be focused on a variety of different directions. However, here we suggest three directions.

- Investigating the upper limits of a single cP system cell. In this thesis, we have shown that they can solve hard problems. However, we have not shown what they cannot compute in a space or time limit.
- Physical implementations of these two systems. A physical implementation of the two systems would allow these algorithms to be realised and utilised.
- cP systems for parallel and distributed algorithms. Here one could investigate using it as a concurrency specification language for well-known problems and investigate the usefulness for more implementations on standard hardware.

Bibliography

[1] Andrew Adamatzky. A brief history of liquid computers. *Philosophical Transactions of the Royal Society B*, 374(1774), 2019.

[2] Artiom Alhazov and Mario J Pérez-Jiménez. Uniform solution of QSAT using polarizationless active membranes. In Jérôme Durand-Lose and Maurice Margenstern, editors, *International Conference on Machines, Computations, and Universality*, volume 4664 of *Lecture Notes in Computer Science*, pages 122–133. Springer, 2007.

[3] Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009.

[4] Joshua J Arulanandham, Cristian S Calude, and Michael J Dinneen. Solving SAT with Bilateral Computing. *Romanian Journal of Information Science and Technology*, 6(1-2):9–18, 2003.

[5] Mordechai Ben-Ari. How to Solve the Santa Claus Problem. *Concurrency: Practice and Experience*, 10(6):485–496, 1998.

[6] Nick Benton. Jingle Bells: Solving the Santa Claus Problem in Polyphonic C#. *Unpublished manuscript*, Mar 2003.

[7] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for C#. In Boris Magnusson, editor, *ECOOP 2002 — Object-Oriented Programming*, pages 415–440, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[8] Cristian Calude. *Theories of computational complexity*. Elsevier, 2011.

[9] Cristian Calude and Lila Sântean. On a theorem of Günter Asser. *Mathematical Logic Quarterly*, 36(2):143–147, 1990.

[10] Ashok K. Chandra, Dexter C. Kozen, and Larry J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, January 1981.

[11] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, page 151–158, New York, NY, USA, 1971. ACM.

[12] Stephen A. Cook and Robert A. Reckhow. Time bounded random access machines. *Journal of Computer and System Sciences*, 7(4):354–375, 1973.

[13] James Cooper and Radu Nicolescu. Alternative representations of P systems solutions to the graph colouring problem. *Journal of Membrane Computing*, 1(2):112–126, 2019.

[14] James Cooper and Radu Nicolescu. The Hamiltonian cycle and travelling salesman problems in cP systems. *Fundamenta Informaticae*, 164(2-3):157–180, 2019.

[15] Daniel Díaz-Pernil, Miguel A Gutiérrez-Naranjo, Mario J Pérez-Jiménez, and Agustín Riscos-Núñez. A linear–time tissue P system based solution for the 3–coloring problem. *Electronic Notes in Theoretical Computer Science*, 171(2):81–93, 2007.

[16] Michael J. Dinneen. A program-size complexity measure for mathematical problems and conjectures. In *Proceedings of the International Workshop on Theoretical Computer Science (WTCS 2012)*, volume 7160 of *Lecture Notes in Computer Science*, pages 81–93. Springer, Heidelberg, 2012.

[17] Michael J. Dinneen, Yun-Bum Kim, and Radu Nicolescu. A faster P solution for the Byzantine agreement problem. In Marian Gheorghe, Gheorghe Păun, Thomas Hinze, Grzegorz Rozenberg, and Arto Salomaa, editors, *11th International Conference on Membrane Computing (CMC11), Revised Selected Papers*, volume 6501 of *Lecture Notes in Computer Science*, pages 175–197. Springer, 2010.

[18] Michael J. Dinneen, Yun-Bum Kim, and Radu Nicolescu. A faster P solution for the Byzantine agreement problem. Report CDMTCS-388, Centre for Discrete Mathematics and Theoretical Computer Science, The University of Auckland, Auckland, New Zealand, July 2010.

[19] Arnold Emch. Two Hydraulic Methods to Extract the nth Root of any Number. *The American Mathematical Monthly*, 8(1):10–12, 1901.

[20] Faith E Fich, Prabhakar Ragde, and Avi Wigderson. Relations between concurrent-write models of parallel computation. *SIAM Journal on Computing*, 17(3):606–627, 1988.

[21] Message P Forum. MPI: A Message-Passing Interface Standard. Technical report, University of Tennessee, Knoxville, TN, USA, 1994.

[22] Eli Gafni, Joseph Naor, and Prabhakar Ragde. On separating the EREW and CREW PRAM models. *Theoretical Computer Science*, 68(3):343–346, 1989.

[23] Phillip B Gibbons. A more practical PRAM model. In *Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*, pages 158–168, 1989.

[24] Miguel A. Gutiérrez-Naranjo, Mario J. Pérez-Jiménez, and Francisco J. Romero-Campero. A Linear Solution for QSAT with Membrane Creation. In Rudolf Freund, Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa, editors, *Membrane Computing*, volume 3850 of *Lecture Notes in Computer Science*, pages 241–252. Springer, 2006.

[25] Alec Henderson and Radu Nicolescu. Actor-like cP Systems. In *Membrane Computing*, volume 11399 of *Lecture Notes in Computer Science*, pages 160–187. Springer, 2019.

[26] Alec Henderson, Radu Nicolescu, and Michael J. Dinneen. Solving a PSPACE-complete problem with cP systems. *Journal of Membrane Computing*, 2(4):311–322, Dec 2020.

[27] Alec Henderson, Radu Nicolescu, and Michael J. Dinneen. Sublinear P system solutions to NP-complete problems. Report CDMTCS-559, Centre for Discrete Mathematics and Theoretical Computer Science, University of Auckland, Auckland, New Zealand, January 2022.

[28] Alec Henderson, Radu Nicolescu, Michael J Dinneen, TN Chan, Hendrik Happe, and Thomas Hinze. Turing completeness of water computing. *Journal of Membrane Computing*, 3(3):182–193, 2021.

[29] Alec Henderson, Ocean Wu, Michael J Dinneen, and Radu Nicolescu. cP systems Solution of the Santa Claus Problem. In *Asian Branch of International Conference on Membrane Computing (ACMC2018)*, page 358, 2018.

[30] Thomas Hinze, Hendrik Happe, Alec Henderson, and Radu Nicolescu. Membrane computing with water. *Journal of Membrane Computing*, 2(2):121–136, 2020.

[31] Jason Hurt and Jan B. Pedersen. Solving the Santa Claus Problem: A comparison of various concurrent programming techniques. *Communicating Process Architectures 2008: WoTUG-31*, 66:381, 2008.

[32] Mihai Ionescu, Gheorghe Păun, and Takashi Yokomori. Spiking neural P systems. *Fundamenta informaticae*, 71(2, 3):279–308, 2006.

[33] Tseren-Onolt Ishdorj, Alberto Leporati, Linqiang Pan, Xiangxiang Zeng, and Xingyi Zhang. Deterministic solutions to QSAT and Q3SAT by spiking neural P systems with pre-computed resources. *Theoretical Computer Science*, 411(25):2345–2358, 2010.

[34] Tseren-Onolt Ishdorj, Otgonnaran Ochirbat, and Chuluunbandi Naimannaran. A μ-fluidic Biochip Design for Spiking Neural P Systems. *International Journal of Unconventional Computing*, 15(1), 2020.

[35] Simon Peyton Jones. Beautiful Concurrency. *Beautiful Code: Leading Programmers Explain How They Think*, pages 385–406, 2007.

[36] Yun-Bum Kim. *Distributed algorithms in membrane systems*. PhD thesis, ResearchSpace@ Auckland, 2012.

[37] Alberto Leporati, Luca Manzoni, Giancarlo Mauri, Antonio Porreca, and Claudio Zandron. Characterizing PSPACE with shallow non-confluent P systems. *Journal of Membrane Computing*, 1(2):75–84, 2019.

[38] Alberto Leporati, Luca Manzoni, Giancarlo Mauri, Antonio E Porreca, and Claudio Zandron. Solving QSAT in sublinear depth. In Thomas Hinze, Grzegorz Rozenberg, Arto Salomaa, and Claudio Zandron, editors, *International Conference on Membrane Computing*, volume 11399 of *Lecture Notes in Computer Science*, pages 188–201. Springer, 2019.

[39] Yezhou Liu, Radu Nicolescu, and Jing Sun. Formal verification of cP systems using PAT3 and ProB. *Journal of Membrane Computing*, 2(2):80–94, 2020.

[40] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[41] Kushani Mahatantila, Rohana Chandrajith, HAH Jayasena, and KB Ranawana. Spatial and temporal changes of hydrogeochemistry in ancient tank cascade systems in Sri Lanka: evidence for a constructed wetland. *Water and Environment Journal*, 22(1):17–24, 2008.

[42] Vincenzo Manca. DNA and membrane algorithms for SAT. *Fundamenta Informaticae*, 49(1-3):205–221, 2002.

[43] Carlos Martín-Vide, Gheorghe Păun, Juan Pazos, and Alfonso Rodríguez-Patón. Tissue P systems. *Theoretical Computer Science*, 296(2):295–326, 2003.

[44] SGS-THOMSON Microelectronics. occam 2.1 reference manual. http://www.wotug.org/occam/documentation/oc21refman.pdf, 1995.

[45] Michael J Moylan. *Fluid logic in simple terms*. Transatlantic Arts, 1968.

[46] Benedek Nagy. On efficient algorithms for SAT. In *International Conference on Membrane Computing, LNCS 7762*, pages 295–310. Springer, 2012.

[47] Vu Nguyen, Sophia Deeds-Rubin, Thomas Tan, and Barry Boehm. A SLOC counting standard. In *Cocomo ii forum*, volume 2007, pages 1–16. Citeseer, 2007.

[48] Radu Nicolescu. Parallel thinning with complex objects and actors. In Marian Gheorghe, Grzegorz Rozenberg, Arto Salomaa, Petr Sosík, and Claudio Zandron, editors, *International Conference on Membrane Computing*, volume 8961 of *Lecture Notes in Computer Science*, pages 330–354. Springer, 2014.

[49] Radu Nicolescu. Revising the Membrane Computing Model for Byzantine Agreement. In Alberto Leporati, Grzegorz Rozenberg, Arto Salomaa, and Claudio

Zandron, editors, *Membrane Computing*, pages 317–339, Cham, 2017. Springer International Publishing.

[50] Radu Nicolescu and Alec Henderson. An introduction to cP Systems. In Carmen Graciani, Agustín Riscos-Núñez, Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa, editors, *Enjoying Natural Computing: Essays Dedicated to Mario de Jesús Pérez-Jiménez on the Occasion of His 70th Birthday*, volume 11270 of *Lecture Notes in Computer Science*, pages 204–227. Springer, 2018.

[51] Radu Nicolescu, Florentin Ipate, and Huiling Wu. Towards high-level P systems programming using complex objects. In *International Conference on Membrane Computing*, pages 255–276, 2013.

[52] Radu Nicolescu and Huiling Wu. Complex objects for complex applications. *Romanian Journal of Information Science and Technology*, 17(1):46–62, 2014.

[53] I Parberry. Parallel speedup of sequential machines: A defense of parallel computation thesis. *SIGACT News*, 18(1):54–67, March 1986.

[54] Gheorghe Păun. Computing with membranes. *Journal of Computer and System Sciences*, 61(1):108–143, 2000.

[55] Gheorghe Păun. P systems with active membranes: Attacking NP-Complete problems. *Journal of Automata, Languages and Combinatorics*, 6(1):75–90, 2001.

[56] Alban William Phillips. *Mechanical models in economic dynamics*. London School of Economics an Political Science, 1950.

[57] Julia Robinson. General recursive functions. *Proceedings of the american mathematical society*, 1(6):703–718, 1950.

[58] William H Ryder. A System Dynamics View of the Phillips Machine. In *27th International Conference of the System Dynamics Society*, 2009.

[59] Emil Sekerinski and Shucai Yao. Refining Santa: An Exercise in Efficient Synchronization. In John Derrick, Brijesh Dongol, and Steve Reeves, editors, Proceedings *18th Refinement Workshop,* Oxford, UK, 18th July 2018, volume 282 of *Electronic Proceedings in Theoretical Computer Science*, pages 68–86. Open Publishing Association, 2018.

[60] Daniel E Severin et al. Unary primitive recursive functions. *Journal of Symbolic Logic*, 73(4):1122–1138, 2008.

[61] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 2012.

[62] Hermann Stamm-Wilbrandt. *Programming in Propositional Logic or Reductions: Back to the Roots (Satisfiability)*. Sekretariat für Forschungsberichte, Inst. für Informatik III University of Bonn, 1993.

[63] Martin Sulzmann, Edmund S.L. Lam, and Peter Van Weert. Actors with multi-headed message receive patterns. In *International Conference on Coordination Languages and Models*, pages 315–330. Springer, 2008.

[64] Nicolas Taberlet, Quentin Marsal, Jérémy Ferrand, and Nicolas Plihon. Hydraulic logic gates: building a digital water computer. *European Journal of Physics*, 39(2):025801, 2018.

[65] Georg Trogemann, Alexander Yuryevich Nitussov, and Wolfgang Ernst. *Computing in Russia: the history of computer devices and information technology revealed*. Vieweg Braunschweig, 2001.

[66] John A. Trono. A new exercise in concurrency. *ACM SIGCSE Bulletin*, 26(3):8–10, 1994.

[67] Adrian Turcanu and Florentin Ipate. Computational properties of two P systems solving the 3-colouring problem. In *2012 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 62–69. IEEE, 2012.

[68] Huiling Wu. *P Systems as Formal Models for Distributed Algorithms*. PhD thesis, ResearchSpace@ Auckland, 2014.

[69] F. Zappa and S.E. Esculapio. *Microcontrollers. Hardware and Firmware for 8-bit and 32-bit devices*. LIGHTNING SOURCE Incorporated, 2017.

Appendix A Distributed solution

Here we present an alternative ruleset to achieve the Cartesian product, cf. rules 9 and 10 from Table 3.17. This ruleset utilises *synchronous* communication between cells (see [50] for more details about multi cell communication). One of the key differences is that this ruleset always consumes something on the left hand side. This seems to make single cell cP systems more difficult to design but also should remove the ability to produce unreasonable amounts of data in 1 step.

Our solution utilises $\sqrt{n}+1$ top level cells with cell 0 being the main cell and all others using identical rule sets only communicating with cell 0 (cf Figure 6.1).

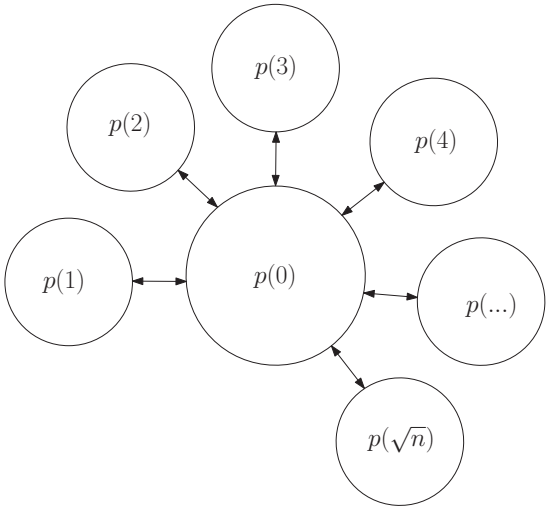


Figure 6.1: Diagram showing the graph representation of the distributed system.

Our alternative ruleset can be broken up into two parts. The first ruleset is identical for $\sqrt{n}$ cells with a processor id $p(i)$ where $i \in \{1, 2, \ldots \sqrt{n}\}$. The rules:

$$s_1\ ?\{X\} \quad \rightarrow_+ \quad s_2\ X \tag{1}$$
$$s_2\ a(X) \quad \rightarrow_+ \quad s_3\ a(X\ b(Y)) \mid p(Y) \tag{2}$$
$$s_3\ X\ p(Y) \quad \rightarrow_+ \quad s_1\ !_0\{X\} \tag{3}$$

describe the system. With each cell getting sent, a group of allocations which it then sends back with after doing a Cartesian product with its processor id.

The main cell will simply send the allocations to all of the other cells using the following rules:

$$s_4\ a(X) \quad \rightarrow_+ \quad s_5\ !_\forall \{j(X)\} \tag{8.1}$$
$$s_5\ ?_\_ \{X\} \quad \rightarrow_+ \quad s_6\ X \tag{8.2}$$

Once it has received the results from the other cells it then processes them using the adjusted rules:

$$\begin{array}{lll} s_6 & \rightarrow_+ & s_7 \\ a(x(i(I)\ Z)\ j(Y)\ b(X)) & & \quad d(x(i(I)\ Z)\ j(\alpha(Y)\ \beta(X))) \\ & & \quad d(x(i(IP)\ Q)\ j(\alpha(Y)\ \beta(X))) \\ & & \mid b(x(i(IP)\ Q)\ j(X)) \\ & & \mid m(P) \end{array} \tag{9'}$$

$$\begin{array}{lll} s_6 & \rightarrow_+ & s_7 \\ a(x(i(I)\ Z)\ j(Y)\ b(X)) & & \quad d(x(i(I)\ Z)\ j(\alpha(Y)\ \beta(X))) \\ & & \mid b(x(i(IP)\ Q)\ j(X)) \\ & & \mid p(P) \end{array} \tag{10'}$$

We note that the states of the original ruleset will also need to be adjusted to incorporate the changes. However, this should be straightforward.

Appendix B Bijection between integers and branch numbers

Given an allocation of variables $\{x_0 = \alpha_0, x_1 = \alpha_1, \ldots, x_{n-1} = \alpha_{n-1}\}$ we use the following code to get branch number j:

```
j = 0
for i ← 0 to n − 1 do
    if α_i = 0 then
        j ← j ∗ 2
    else
        j ← j ∗ 2 + 1
```

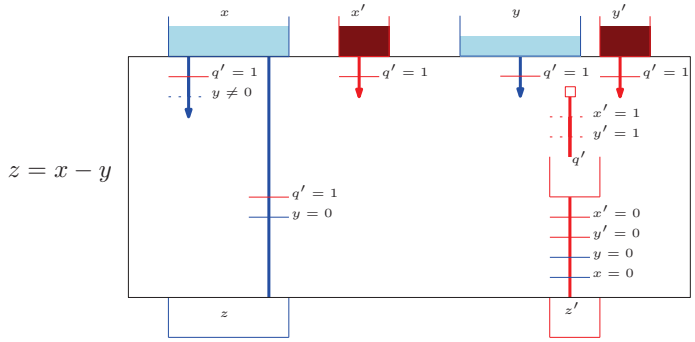
Given a branch number j we can retrieve an allocation a using the following code:

```
a ← {}
for i ← n − 1; to 0 do
    if j % 2 = 0 then
```

$a \leftarrow a \cup \{x_i \leftarrow 0\}$
$j \leftarrow j/2$
else
$a \leftarrow a \cup \{x_i \leftarrow 1\}$
$j \leftarrow (j-1)/2$

Appendix C Example trace of controlled subtraction

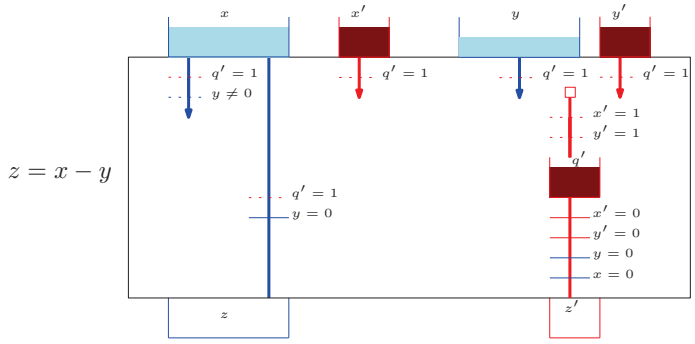
Here we present a trace of saturation subtraction for both the tank system presented in Figure 4.2, and cP rules in Table 4.7.



s_1	$q()$	$c_x(1)$	$c_y(1)$	$c_z()$	$v_x(3)$	$v_y(2)$	$v_z()$

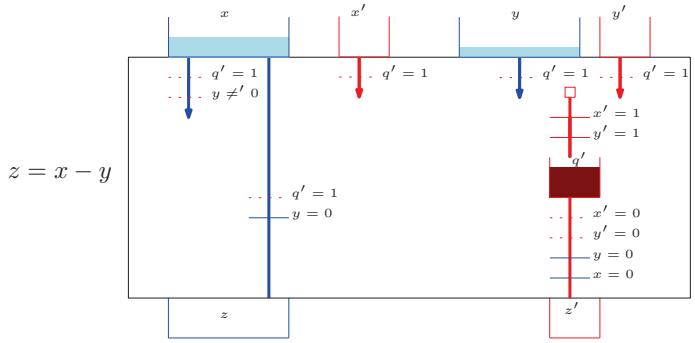
Figure 6.2: The initial state of controlled subtraction of $3 \ominus 2$.

$+$



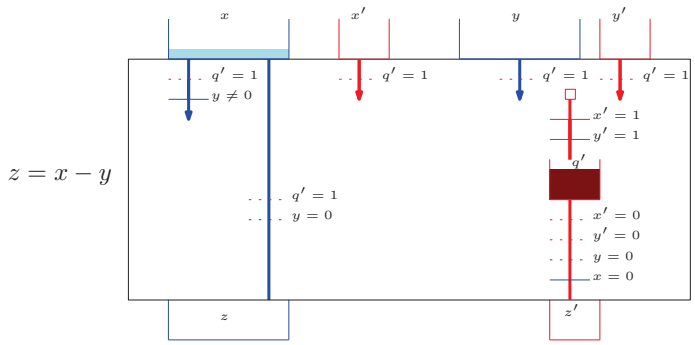
s_2	$q(1)$	$c_x(1)$	$c_y(1)$	$c_z()$	$v_x(3)$	$v_y(2)$	$v_z()$	rule (1)

Figure 6.3: The first step of controlled subtraction of $3 - 2$.



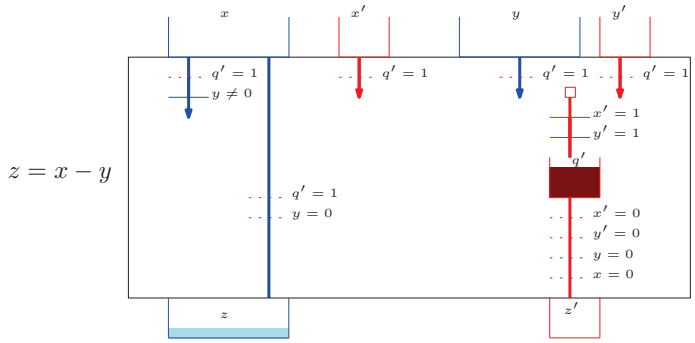
s_2	$q(1)$	$c_x()$	$c_y()$	$c_z()$	$v_x(2)$	$v_y(1)$	$v_z()$	rules (2, 3, 4, 5)

Figure 6.4: The second step of controlled subtraction of $3 \ominus 2$.



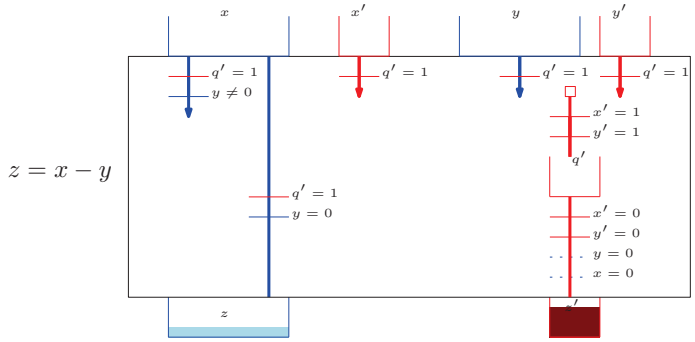
s_2	$q(1)$	$c_x()$	$c_y()$	$c_z()$	$v_x(1)$	$v_y()$	$v_z()$	rules (4, 5)

Figure 6.5: The third step of controlled subtraction of $3 \ominus 2$.



s_2	$q(1)$	$c_x()$	$c_y()$	$c_z()$	$v_x()$	$v_y()$	$v_z(1)$	rule (6)

Figure 6.6: The fourth step of controlled subtraction of $3 \ominus 2$.



s_3	$q()$	$c_x()$	$c_y()$	$c_z(1)$	$v_x()$	$v_y()$	$v_z(1)$	rule (7)

Figure 6.7: The last step of controlled subtraction of $3 \ominus 2$.