Formal verification of biologically inspired computing models

Yezhou Liu

A thesis submitted in fulfilment of the requirements for the degree of
Doctor of Philosophy in Computer Science,
The University of Auckland, 2022.

In memory of my grandfather (Xuesheng Liu, 12 Jan 1931 - 23 Jul 2020).

Abstract

As a recently proposed biologically inspired computing model, cP systems can solve computationally hard problems in polynomial – often linear or sublinear – time. Similar to other membrane computing models, cP systems work in an ideal way, having unlimited space and computing power. This thesis discusses the formal verification of cP systems.

cP systems support labelled multiset-based terms and generic rewriting rules. To apply a rule, variable terms in its lhs (left-hand-side) and promoters need to be unified against system terms. Although several first-order unification algorithms were proposed in previous work, none of them can be readily applied to cP systems due to the use of labelled and nested multisets. In order to solve the issue, we formally defined the unification problem for labelled multisets and proposed a corresponding algorithm, namely LNMU. This can solve well-formed labelled multiset unification problems in linear time.

To verify cP systems, both model checking and interactive theorem proving are considered in this thesis. By using formal tools including PAT3, ProB, and Coq, several cP systems were verified including two NPC solutions proposed in this study. These are: Π_{SSP} – a cP system that solves the subset sum problem in linear time, and Π_{Sudoku} – a cP system that solves Sudoku ($m \times m$) in sublinear time. In order to automate the verification process, we proposed several mapping guildlines, which can be used to transform cP system notation into modelling languages including CSP#, B, and Gallina automatically.

Using existing general purpose formal tools to verify cP systems often requires human intervention. Furthermore, it is extremely hard to completely model cP systems with complex generic rules in third-party verifiers. In order to overcome these limitations, we designed and implemented a domain-specific formal framework for cP system simulation and verification, namely cPV. By implementing LNMU in an optimised way, cPV can efficiently simulate and verify cP systems. System

properties including deadlockfreeness, confluence, termination, determinism, and goal reachability can be automatically verified in cPV.

We also presented a new research direction: using cP systems as a theorem proving tool. A cP system that can efficiently perform equational deduction was proposed. Using certain cP system friendly encodings and the power of maximal parallelism, the cP system can be exponentially faster than traditional rewrite systems when proving equational theories.

Keywords: Bio-inspired computing, cP systems, formal verification, automated deduction, unification.

Acknowledgements

First and foremost, I would like to thank my PhD supervisors, Associate Professor Jing Sun and Doctor Radu Nicolescu for their excellent supervision and support. Their guidance and advice carried me through all the stages of my study.

I would like to thank other lecturers and students at the University of Auckland, including, but not limited to, Doctor Michael J Dinneen, Doctor James Cooper, Doctor Alec Henderson, Doctor Nacha Chondamrongkul, Doctor Chenghao Cai, Professor Jim Warren, TN Chan, and James Shi for their help and support.

My greatest gratitude goes to my wife, Yang Yang (Yvonne), for her love, understanding, and unconditional support. I am thankful to my parents and grandparents, for their encouragement and their care of my tortoise during these years. I would also thank my dear hippos, Amy and Xibo, who bring happiness to my family everyday.

I would also like to thank The Unviersity of Auckland, for the scholarships, awards, and grant during my period of study. Some of my finest memories are from my time at The University of Auckland.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Natural Computing is the field of research that mainly investigates three classes of methods. These include: (1) human-designed computing inspired by nature, (2) computing that happens in nature, and (3) using natural materials to perform computations. Both biology-based and physics-based approaches and algorithms are aspects of natural computing.

Bio-inspired computing, short for *biologically inspired computing*, is a subfield of natural computing, which focuses on solving computer science problems by using biological models relating to connectionism, social behavior, and emergence. Inspired by the structure of living cells, *Membrane computing* was proposed as a branch of bio-inspired computing [1].

Membrane systems (*P systems*) are a parallel and distributed computational model, which have one or multiple membranes arranged in a hierarchical way, and multisets of objects in different regions delimited by these membranes. The objects are represented by symbols (atoms) from a given alphabet, which can evolve according to evolution rules associated with the regions. The rules are applied non-deterministically in a maximally parallel manner. Membranes in P systems can be divided repeatedly, where 2^n "processors" and exponential memory space can be obtained in n steps [2].

P systems with complex objects (*cP systems*) are a variant of P systems, which supports *complex symbols* (labelled multiset-based terms) and *generic rules* (rules that may contain variables) [3]. By representing membrane structures using complex symbols, cP systems only have evolution rules for top-level membranes. Using a fixed constant number of generic rules, cP systems can solve several NP-complete (nondeterministic polynomial-time complete, NPC) problems in linear

or sublinear time. This can include the subset sum problem [4], Hamiltonian cycle problem [5], travelling salesman problem [5], and Sudoku [6].

After designing a cP system that solves a certain problem, a validation is required to check if the cP system is reliable and behaves as expected. *Formal verification* is a mathematical way to achieve this validation. By describing a cP system as an abstract mathematical model, its properties can be proven or disproven using formal methods. Using complex symbols to represent membrane structures with objects, and generic rules to describe the logic of term rewriting, cP systems have a strong representational power. However, the design of cP systems also makes them hard to model, simulate, or verify using third-party formal tools.

As the first cP system formal verification study, this thesis verifies several cP systems using different approaches, this includes model checking and deductive verification. We[1] proposed multiple cP systems that solve NPC problems, and verified their system properties. A major challenge of simulating and verifying cP systems is to handle the unification of rules. In order to solve this issue, we formally defined the labelled multiset unification problem, and proposed a corresponding algorithm. We discussed the advantages and disadvantages of using third-party general purpose formal tools to model and verify cP systems. In order to overcome the limitations, we designed and implemented a cP system-specific simulation and formal verification framework named cPV, and evaluated its performance. Compared to using general purpose formal tools to verify cP systems, which requires human intervention, cP systems can be verified in cPV fully automatically. The following research questions are answered in this thesis:

- How can formal verification of cP systems be conducted?
- What are the advantages and disadvantages of using different formal tools to verify cP systems?
- Compared to existing formal tools, can we design and implement a better domain-specific software tool for cP system formal verification?

1.1 Outline

This thesis proceeds as follows. Chapter 2 introduces the syntaxes of P systems and cP systems, formal verification and tool support, and related work. The way

[1] The use of "we" throughout this thesis is by purpose, it is used to involve the reader of the thesis, as recommended by Knuth. Nonetheless, the thesis is the sole work of the author.

cP systems work will be explained in detail, and the formal methods and tools used in this study will be introduced. cP systems not only extend traditional P systems by supporting complex symbols and generic rewriting rules, but also apply multiple rules following a top-down weak priority order. Thus, cP systems are a variant rather than a superset of traditional P systems.

Chapter 3 discusses labelled multisets and the corresponding unification problem, which is fundamental to generic rewriting rules in cP systems. A formal definition of labelled multiset unification is given, and an efficient unification algorithm named LNMU is proposed [7]. LNMU always terminates, and can find all the unifiers to a labelled multiset unification problem. By carefully selecting transformation rules in LNMU, well-formed labelled multiset unification problems can be solved in linear time.

Chapter 4 discusses how to verify cP systems via model checking [4]. Two cP systems are proposed, which can solve the subset sum problem in linear time [8], and solve Sudoku in sublinear time [6]. Several system properties of the two cP systems are formally verified using the model checkers PAT3 and ProB. To automate the verification process, two mapping guidelines are proposed, which can be used to transform cP systems into CSP# models and B machines. The performances of verifying cP systems using PAT3 and ProB are compared. A major limitation of model checking is state explosion – cP systems may have a large statespace with exponential states.

As a complementary approach, Chapter 5 introduces how to use the Coq proof assistant to verify cP systems [9]. Using mathematical induction, several properties of cP systems can be proven or disproven without expanding the entire statespace. A library named cP-Coq is designed and implemented in Gallina, to help people verify cP systems in Coq. Using cP-Coq, several cP systems are successfully verified. Similar to the model checking approach, multiple mapping guidelines are introduced to help people model cP systems in Coq. A major limitation of this approach is that human intervention is needed in modelling cP systems, specifying the system properties, and proving the theorems. There is no guarantee that all the theorems can be proven.

Chapter 6 introduces our own software implementations on cP system formal verification. Multiple translators are implemented following the mapping guidelines proposed in previous chapters, which can automatically transform certain cP systems into different models. By integrating the translators with third-party formal tools, several cP systems can be verified fully automatically. However, not

all the cP systems can be automatically modelled and verified using existing formal tools due to the lack of language features. To solve this issue, a cP system-specific formal verifier, namely cPV, is implemented, whose functionalities include language parsing, system simulation, verification algorithms, reduction techniques, counterexample generation, and a graphical user interface (GUI). System properties, including deadlockfreeness, confluence, termination, determinism, and goal reachability of any valid cP system can be automatically verified in cPV. As a modularised and extensible framework, multiple interfaces are provided in cPV, where custom features can be easily added into corresponding modules as needed.

Chapter 7 includes the functional and performance evaluation of cPV. To illustrate the cP system simulation, a case study is introduced. Two benchmark cP system datasets are created and verified in cPV, which can also be used in future studies. By implementing LNMU in an optimised way, cPV is the first software framework that can effectively handle generic rules and compound terms in cP systems.

Chapter 8 explores a future direction of cP system research which uses cP systems to perform automated deduction [10]. A cP system is proposed, which can compute all critical pairs among multiple axioms in logarithmic time. To reduce a term of size m to a normal form, the cP system can be $\mathcal{O}(2^m)$ times faster than traditional rewrite systems. In the future, if a "meta cP system" can be properly designed by encoding other cP systems as input data, the approach can be used to verify cP systems.

Chapter 9 concludes the thesis, and introduces several future directions for cP system formal verification research.

1.2 Key results and publications

The key results of this thesis are presented from Chapter 3 to Chapter 8, which are listed as follows:

- Chapter 3: An efficient labelled multiset unification algorithm
  - Results:
    - The unification problem of labelled multisets is formally defined.
    - An efficient labelled multiset unification algorithm, namely LNMU, is proposed.
    - The first cP system simulator, namely cPSim, is implemented.

  - Publications:
    * Y. Liu, R. Nicolescu, and J. Sun, **An efficient labelled nested multiset unification algorithm**. *Journal of Membrane Computing*, vol. 3, no. 3, pp. 194-204, 2021.
    * Y. Liu, R. Nicolescu, and J. Sun, **Multiset unification and cP system simulation**, in *The International Conference on Membrane Computing 2020*, Vienna, Austria, 2020.

- Chapter 4: Model checking of cP systems
  - Results:
    * A linear cP system solution to the subset sum problem is proposed, which includes 5 rules.
    * A sublinear cP system solution to Sudoku ($m \times m$) is proposed, which includes 16 rules.
    * Two mapping guidelines to transform cP systems into CSP# models and B machines are proposed.
    * The formal verification of cP systems via model checking is conducted and discussed.
  - Publications:
    * Y. Liu, R. Nicolescu, J. Sun, and A. Henderson, **A sublinear Sudoku solution in cP systems and its formal verification**. *Computer Science Journal of Moldova*, vol. 85, no. 1, pp. 3-28, 2021.
    * Y. Liu, R. Nicolescu, and J. Sun, **Formal verification of cP systems using PAT3 and ProB**. *Journal of Membrane Computing*, vol. 2, pp. 84-90, 2020.
    * Y. Liu, R. Nicolescu, and J. Sun, **Formal approach to cP system verification**, in *The 8th Asian Conference on Membrane Computing (ACMC2019)*, p. 232, 2019.

- Chapter 5: Deductive verification of cP systems
  - Results:
    * A Gallina library, namely cP-Coq, is proposed and implemented.

    - Two mapping guidelines to transform cP system components and rules into Gallina are proposed.
    - The formal verification of cP systems via interactive theorem proving is conducted and discussed.
  - Publications:
    - Y. Liu, R. Nicolescu, and J. Sun, **Formal verification of cP systems using Coq**. *Journal of Membrane Computing*, vol. 3, no. 3, pp. 205-220, 2021.

- Chapter 6: cPV – a Formal Verification Framework for cP Systems
  - Results:
    - A domain-specific language for cP systems, namely cPVJ, is proposed.
    - Several translators to transform cP systems (described in cPVJ) into CSP# models, B machines, and Gallina models are implemented.
    - The CSP-translator and B-translator are integrated with PAT3 and ProB, which can be used to verify certain cP systems automatically.
    - A simulation and formal verification framework for cP systems, namely cPV, is proposed and implemented.

- Chapter 7: Evaluation of cPV
  - Results:
    - The functional and performance evaluation of cPV is conducted.
    - Two benchmark cP system datasets are proposed.

- Chapter 8: Towards automated deduction in cP systems
  - Results:
    - A cP system that can perform equational deduction is proposed.
  - Publications:
    - Y. Liu, R. Nicolescu, and J. Sun, **Towards automated deduction in cP systems**. *Information Sciences*, vol. 587, pp. 435-449, 2022.

- Other publications:
  - R. Nicolescu, M. J. Dinneen, J. Cooper, A. Henderson, and Y. Liu, **Logarithmic SAT solution with membrane computing**. *Axioms*, vol. 11, no. 2: 66, 2022

1.3 Summary

This chapter includes a brief introduction and an outline of the thesis. Motivations and major contributions of this study are introduced, and corresponding publications are listed.

In the next chapter, the background of P systems, cP systems, and formal verification will be introduced. A literature review will also be presented, which covers most of the studies that are related to this work.

Chapter 2

Background and Related Work

Most P systems and P system variants share two attractive properties, these are computational completeness and efficiency. P systems are equivalent in power to Turing machines. Using unlimited computational resources, P systems can solve NPC problems in polynomial time [2].

Extended from P systems, an early prototype of cP systems was introduced in [11], which was used to model several distributed algorithms. Using the prototype, solutions to the Boolean satisfiability problem (SAT) [12] and parallel image thinning problem [13] were proposed. cP systems were formally defined in [14], then further introduced in [3, 15].

Several real world or computationally hard problems can be solved in polynomial time using cP systems, these include: the seeded region growing problem [16], the Byzantine agreement problem [17], the most common words problem [18], the Travelling Salesman Problem (TSP) [5], the Hamiltonian Path Problem (HPP) [5], the Hamiltonian Cycle Problem (HCP) [5], the Subset Sum Problem (SSP) [8, 4], Sudoku [6], satisfiability of quantified propositional formulas (QSAT) [19], and SAT [20].

The formal verification of cP systems is a brand new research area, however, several P system models were verified in previous studies via model checking. The rest of this chapter is organised as follows. Section 2.1 introduces the background of P systems, cP systems, and formal verification. Section 2.2 presents work related to this thesis, and Section 2.3 concludes the chapter.

2.1 Background

In the structure of living cells, membranes play an essential role; separating the cytoplasm and nucleus from the environment. Membrane computing formalises the membrane structure to a nondeterministic and parallel computing model.

Several P system variants are proposed in previous studies. These include: (1) tissue P systems [21], which processe symbol-impulses in a net of cells; (2) spiking neural P systems [22], which are a class of neural-like P systems in which the spiking time of neurons plays an essential role; (3) kernel P systems [23], which support structure changing rules with guards which are responsible for changing a system's topology; and (4) cP systems [3], which share the fundamental features of cell-like (tree-based) and tissue (graph-based) P systems, and support complex symbols and generic rules. As a newly proposed P system variant, cP systems are more expressive than other P system variants which only support *ground rules* (rules that do not contain any variables). cP system solutions to computationally hard problems often only contain a fixed constant number of generic rules, and halt in polynomial steps.

To validate cP systems that perform certain computation tasks, formal verification techniques can be applied. Given a cP system Π, we can either exhaustively check if Π holds certain system properties by applying model checking, or mathematically prove Π will always behave in an expected way by performing theorem proving.

2.1.1 P systems

A membrane structure is shown in Fig. 2.1, which consists of five membranes that are hierarchically arranged and labelled with natural numbers. The outmost membrane is called the *skin* membrane, which separates, and hence protects, the internal space of the cell from the *environment*. Multiple inner membranes can be placed inside the skin membrane. *Regions* are delimited by membranes, which may contain symbol objects. Using strings of labelled matching parentheses, the membrane structure shown in Fig. 2.1 can be represented as: $[_1\,[_2\,[_3\,]_3\,[_4\,]_4\,]_2\,[_5\,]_5\,]_1$.

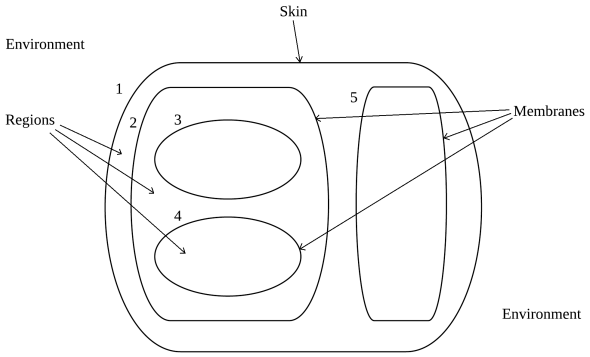

Figure 2.1: A membrane structure

Definition 1. *A P system is a construct of the form [24]:*
$\Pi = (O, C, \mu, \omega_1, \omega_2, \ldots, \omega_m, R_1, R_2, \ldots, R_m, i_o)$,

where:
1) O is the finite and non-empty alphabet of objects;
2) $C \subset O$ is the set of catalysts;
3) μ is a membrane structure, consisiting of m membranes labelled with $1, 2, \ldots, m$;
4) $\omega_1, \omega_2, \ldots, \omega_m$ are strings over O representing the multisets of objects present in the regions $1, 2, \ldots, m$ of the membrane structure;
5) $R_1, R_2, \ldots, R_m$ are finite sets of evolution rules associated with the regions $1, 2, \ldots, m$ of the membrane structure;
6) i_o indicates the output region, which is either one of $1, 2, \ldots, m$ or 0 – which indicates the environment.

Rules in P systems are of the form $u \to v$, where $u \in O^+$, $v \in (O \times Tar)^*$, and $Tar = \{here, in, out\}$. The multiplicities of symbols are represented as superscripts, for example, $a^2 \equiv aa$ and $b^3 \equiv bbb$. Empty multiset is denoted by λ. For example, the rule $a \to bc$ consumes a and produces b and c; the rule $a^2 \to (a, out)^2$ sends 2 copies of a to the environment (outside the system). Catalysts can be used in the rules, for instance, in the rule $ca \to cb$, c is a catalyst that is required by the rule, which will not evolve or move to other regions.

Consider the following P system [2]:
$\Pi_1 = (O, \lambda, \mu, \omega_1, \omega_2, R_1, R_2, i_o)$,
with the following components:
$O = \{a, b, c\}$,
$\mu = [_1\ [_2\ ]_2\ ]_1$,
$\omega_1 = a^2$,

$\omega_2 = \lambda$,
$R_1 = \{a \to a(b, in_2)(c, in_2)^2, a^2 \to (a, out)^2\}$,
$R_2 = \emptyset$,
$i_o = 2$.

The system starts with two copies of a in region 1. Since both the two rules in R_1 are applicable, in each computational step, one of them will be randomly applied. If $a \to a(b, in_2)(c, in_2)^2$ is applied, which processes a copy of a; another copy of a in region 1 must also be processed by the same rule due to the maximal parallelism. The rule reproduces a copy of a in region 1, and sending a copy of b and two copies of c to region 2. Another rule $a^2 \to (a, out)^2$ sends two copies of a to the environment. In Π_1, after applying $a^2 \to (a, out)^2$, the computation halts.

Membrane 2 is the output membrane with no rule applicable. Suppose $a^2 \to (a, out)^2$ is applied at step n, region 2 will contain the objects $b^{2n}c^{4n}$. Thus, the computation result of Π_1 is: $N(\Pi_1) = \{6n \mid n > 0\}$.

More examples of P systems with different membrane structures and evolution rules can be found in [25, 2, 24].

2.1.2 cP systems

cP systems share the fundamental features of cell-like and tissue P systems. Membranes and objects in a top-level cell are represented as labelled nested multiset-based terms, and top-level cells can be organised in graph networks. Each top-level cell has evolution rules, and sub-cells are only used to represent local data. In this study, we will focus on cell-like cP systems, i.e., each cP system has only one top-level cell. cP systems with multiple top-level cells are introduced in [3]. Theoretically, many cP systems with multiple top-level cells can be simulated by single top-level cell cP systems using generic rewriting rules to model intercellular communications.

The grammar of cP systems is shown in Fig. 2.2. The basic vocabulary of cP systems is *simple term*, which consists of *atom* and *variable*. Lowercase letters are used to represent atoms and uppercase letters are variables. For instance, a, b, c are atoms and X, Y, Z are variables. As a special atom, unity symbol *1* is used to represent Peano natural numbers. For example, the natural number 1 is represented as *1*, 2 is represented as *11* or 1^2, 3 is represented as *111* or 1^3, and so on. Underscores (_) are used to denote anonymous variables. λ refers to the empty multiset.

$< term > ::= < simple\text{-}term > | < compound\text{-}term >$
$< multiset > ::= < term > ...$
$< simple\text{-}term > ::= < atom > | < variable >$
$< compound\text{-}term > ::= < functor > (< multiset >)$
$< functor > ::= < atom >$
$< state > ::= < atom >$
$< l\text{-}state > ::= < state >$
$< r\text{-}state > ::= < state >$
$< rule > ::= < lhs > \rightarrow_{\alpha} < rhs >< promoters >$
$< lhs > ::= < l\text{-}state >< multiset >$
$< rhs > ::= < r\text{-}state >< multiset >$
$< promoters > ::= \ | < multiset >$

Figure 2.2: cP system syntax (lhs = left-hand-side, rhs = right-hand-side, α = rule application model)

Compound terms are recursively built by terms with *functors*, where functors are atoms. For example, $f(1)$, $a(b)$, $a(b(cX)d(e\_))$ are compound terms. Compound terms can be used to represent cells, for instance, a cell with label a that contains the two atoms b and c can be presented as $a(bc)$.

Compound terms are multiset-based, the writing order of their elements does not matter. For example, $a(bbc)$ and $a(cbb)$ are identical, both of them represent a cell with the label a, which contains three atoms b, b and c.

Top-level cells in cP systems have evolution rules. A rule consists of *lhs*, *rhs*, *promoters* and an *application model* (α). Both a rule's lhs and rhs contain a state and a multiset of terms. For example, $s_1\ a\ \rightarrow_1\ s_2\ bc$ is a rewriting rule, whose *l-state* is s_1, *r-state* is s_2, and application model is 1. Atom a is a term in its lhs, atoms b and c are terms in its rhs. This rule consumes a term a and produces two terms b and c. In other words, it rewrites a as bc.

Each cell-like cP system contains a number of terms alternatively named *system terms*, and has a *state* named *system state*. States are atoms, to distinguish them with terms, it is conventional to write them as atom s with subscripts of natural numbers, such as s_1, s_2 and s_3.

A rule is *applicable* if and only if its l-state matches the system state, and its lhs terms and promoters exist in the system (can be unfied againist system terms). After applying the rule, terms matching its lhs will be consumed, and terms appearing in its rhs – by applying the substitution obtained from the matching of the lhs terms and promoters with system terms – will be produced. The system state will then be changed to the rule's r-state.

For generic rules with variable terms such as $a(X)$, $b(\_)$, cP systems support a *one-way unification* (pattern matching). After successfully unifying a rule's variable terms against systems terms, it can be applied.

Suppose that a cP system (at s_1) has two terms $a(1)$, $a(11)$ and a generic rule $s_1\ a(X) \rightarrow_1 s_2\ b(X)$. The variable term $a(X)$ in the rule's lhs can be unified against system terms $a(1)$ and $a(11)$. Two unifiers can be obtained here, which are $\vartheta_1 = \{X \mapsto 1\}$ and $\vartheta_2 = \{X \mapsto 11\}$. Since the application model of this rule is "exactly-once", it will only be applied once. cP systems are *nondeterministic*, thus one of ϑ_1 and ϑ_2 will be randomly selected. Suppose ϑ_1 is selected, the rule will apply ϑ_1, and obtain a ground rule $s_1\ a(1) \rightarrow_1 s_2\ b(1)$. By applying it, the system will consume $a(1)$ and produce $b(1)$.

Two major *application models* are supported in cP systems, these are "*exactly-once* (1)" and "*max-parallel* (+)". As mentioned, in the exactly-once model, a rule will only be applied once. However, in the max-parallel model, all the system terms that can apply a rule will apply it simultaneously.

Suppose a cP system (at s_1) has three system terms $a(1^2)$, $a(1^3)$, $a(1^3)$ and a rule $s_1\ a(1X) \rightarrow_\alpha s_2\ a(X)$. By unifying the variable term $a(1X)$ in the rule with the system terms, three ground rules can be obtained. These are: $s_1\ a(11) \rightarrow_\alpha s_2\ a(1)$, $s_1\ a(11^2) \rightarrow_\alpha s_2\ a(1^2)$ and $s_1\ a(11^2) \rightarrow_\alpha s_2\ a(1^2)$, where the variable X is mapped to 1, 1^2 and 1^2, respectively. When the application model is exactly-once ($\alpha = 1$), the system will non-deterministically choose one ground rule to apply. The computation result can be $a(1)$, $a(1^3)$, $a(1^3)$ or $a(1^2)$, $a(1^2)$, $a(1^3)$. When the application model is max-parallel ($\alpha = +$), the system will apply all the ground rules, and the computation result will be $a(1)$, $a(1^2)$, $a(1^2)$.

Given a rule in the max-parallel model, ground rules (obtained by unifying its variable terms with system terms) that can be applied together are called *compatible*. Suppose a cP system (at state s_1) has four system terms $a(c)$, $a(d)$, $b(e)$, $b(f)$, and a rule $s_1\ a(X)\ b(Y) \rightarrow_+ s_1\ g(XY)$. By unifying $a(X)$ and $b(Y)$ against the system terms, the following ground rules can be obtained: *r1*: $s_1\ a(c)\ b(e) \rightarrow_+ s_1\ g(ce)$, *r2*: $s_1\ a(c)\ b(f) \rightarrow_+ s_1\ g(cf)$, *r3*: $s_1\ a(d)\ b(e) \rightarrow_+ s_1\ g(de)$, and *r4*: $s_1\ a(d)\ b(f) \rightarrow_+ s_1\ g(df)$. The four ground rules *r1*, *r2*, *r3*, and *r4* will be non-deterministically applied. Suppose the system applies *r2*, the system terms $a(c)$ and $b(f)$ will be locked by *r2*, and be rewritten as $g(cf)$. The only free terms in the system are $a(d)$ and $b(e)$, which can be used in *r3*. Thus, *r2* and *r3* can be applied together – they are compatible. Similarly, *r1* and *r4* are compatible. In the max-parallel model, the system will non-deterministically choose *r2*, *r3* or *r1*, *r4* to apply.

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

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

In each step, newly generated terms will be temporarily put into a "virtual product membrane", which will not be available until the next step. Suppose a cP system (at s_1) has two system terms $a(c)$, $b(d)$ and two rules *r1*: $s_1\ a(X) \rightarrow_1 s_2\ b(X)$ and *r2*: $s_1\ b(X) \rightarrow_+ s_2\ c(X)$. *r1* and *r2* will be applied in the same step. The term $b(c)$ generated by *r1* will be sent to the virtual product membrane, which will not be consumed by *r2* in the same step. After applying the two rules, the system state will be changed to s_2, then no rule is applicable. The computational result of the system will be $b(c)$, $c(d)$.

To apply a rule with promoters, they must exist in the system and must not be consumed. Suppose a cP system (at s_1) has two system terms $y(1^6)$, $z(1^4)$, and a rule $s_1 \rightarrow_1 s_2\ x(X) \mid y(XZ)\ z(Z)$. By unifying $y(XZ)$ and $z(Z)$ with $y(1^6)$ and $z(1^4)$, a ground rule can be obtained: $s_1 \rightarrow_1 s_2\ x(1^2) \mid y(1^6)\ z(1^4)$. By applying it, a term $x(1^2)$ will be generated. $y(1^6)$ and $z(1^4)$ are promoters, they will be checked by the rule, but will not be consumed.

2.1.3 A cP system in use

Given the two natural numbers y and z, a cP system (at s_0) with the following ruleset can compute their product ($x = y \times z$).

s_0		$\rightarrow_1$	s_2	$x(\lambda) \mid z(\lambda)$	(*R1*)
s_0		$\rightarrow_1$	s_1	$x(\lambda)\ w(Y) \mid y(Y)$	(*R2*)
s_1	$x(X)\ w(\lambda)$	$\rightarrow_1$	s_2	$x(X)$	(*R3*)
s_1	$x(X)\ w(Y1)$	$\rightarrow_1$	s_1	$x(XZ)\ w(Y)) \mid z(Z)$	(*R4*)

The rule *R1* checks the promoter z, if $z = 0$, by applying *R1*, the system will change its state to s_2, and generate a term $x(\lambda)$ which indicates that $x = 0$. Since no rule's l-state matches s_2, no rule is applicable, therefore, the system halts.

R2 generates a term $x(\lambda)$, assigns the value of $y(Y)$ to $w(Y)$, and changes the system state to s_1.

When $w(\lambda)$ appears in the system, which means $w = 0$, *R3* changes the system state to s_2, reproduces $x(X)$, then the system will halt.

If $w(Y\,1)$ is in the system, which means $w > 0$, *R4* adds the value of $z(Z)$ to $x(X)$, creates a new terms $x(XZ)$, consumes the existing term $x(X)$, and subtracts one from $w(Y\,1)$ by rewriting it as $w(Y)$.

After applying the ruleset a number of times, the system will eventually halt, and the term $x(\_)$ in the final system configuration will indicate the computation result of $y \times z$.

Suppose $y = 3$ and $z = 6$, which are encoded as two system terms $y(1^3)$ and $z(1^6)$. Following the top-down order, *R1* will be considered first, however, since $z(\lambda)$ cannot be found in the system, *R1* is not applicable, and the system will consider *R2* instead.

R2 is applicable, and by unifying $y(Y)$ to $y(1^3)$, a mapping $\{Y \mapsto 1^3\}$ can be obtained. By applying the mapping to $w(Y)$, the system generates two the terms $x(\lambda)$ and $w(1^3)$, and changes its state to s_1. Since no other rule would change the system state from s_0 to s_1 (the same to *R2*), the first computational step is over.

In the second step, the ruleset will be applied again. The system first considers *R1* and *R2*, which are not applicable because their l-states do not match the system's current state (which is s_1). *R3* is also not applicable, since there is no $w(\lambda)$ in the system. *R4* is applicable, and by applying it, the system consumes $x(\lambda)$ and $w(1^3)$, and produces $x(1^6)$ and $w(1^2)$ from the mapping $\{Z \mapsto 1^6\}$; which was obtained by unifying $z(Z)$ in *R4* with the system term $z(1^6)$.

In the third step, *R1*, *R2*, and *R3* are still not applicable. By applying *R4*, the system consumes $x(1^6)$ and $w(1^2)$, and produces $x(1^{12})$ and $w(1)$. In the fourth step, the system will still apply *R4*, consumes $x(1^{12})$ and $w(1)$, and produces $x(1^{18})$ and $w(\lambda)$.

Finally, in the last computational step, since there is a term $w(\lambda)$ in the system, *R3* is applicable, this will be considered before *R4*. By applying *R3*, the system consumes $w(\lambda)$, reproduces $x(1^{18})$, and changes its state to s_2, then halts. In the final system configuration, the term $x(1^{18})$ indicates that $x = 18 = y \times z = 3 \times 6$.

2.1.4 Formal verification and tool support

Formal verification is the process of proving or disproving whether a system satisfies certain formal specifications or properties. This uses formal methods of mathematics. Two major approaches of formal verification are *model checking* and *deductive verification*.

By modelling a system as a finite graph, model checking uses an exhaustive search procedure to determine if certain specifications are satisfied by the system [26]. Two major approaches are commonly used in system modelling, these are *Kripke structures* and *Labelled Transition Systems* (LTS) [27]. Temporal logics are designed for expressing system properties, which include *linear-time* and *branching-time* logics such as Linear Temporal Logic (LTL) [28] and Computation Tree Logic (CTL) [29].

Deductive verification generates a set of proof obligations from a system and its specifications. It also uses proof assistants or automated theorem provers such as Satisfiability Modulo Theories (SMT) solvers to prove the obligations. Compared to model checking, which can be performed automatically, deductive verification requires human knowledge on system modelling, property specification, and theorem proving.

2.1.4.1 Model checkers

Since model checking has emerged as a powerful approach to automatic system verification, several model checkers were proposed and implemented. These include, but are not limited to: (1) Berkeley Lazy Abstraction Software verification Tool (BLAST) [30], which is an automatic verification tool for checking temporal safety properties of C programs; (2) CPA checker [31], which is a verification framework that aims to achieve easy integration of new verification components; (3) NuSMV [32] and NuSMV 2 [33], where NuSMV is a symbolic model checker based on binary decision diagrams (BDDs), and NuSMV 2 is an extended version of NuSMV that combines BDD-based model checking with SAT-based model checking; (4) PRISM [34], which is a symbolic model checker that was developed for the analysis of probabilistic systems; (5) SPIN [35], which is a software tool for verifying the correctness of concurrent systems; and (6) UPPAAL [36], which is an integrated tool suite for modelling and verification of Real-Time Systems (RTS).

In this thesis, two general purpose model checkers are used to verify cP systems. These are Process Analysis Toolkit 3 (PAT3) [37] and ProB [38, 39]. An early version of Process Analysis Toolkit was presented in [40], which was designed to

verify event-based compositional system models specified in Communicating Sequential Processes (CSP). PAT3 is the latest version of PAT, which supports several system modelling options such as Communicating Sequential Programs (CSP#), Probability CSP, RTS, LTS, and timed automata. PAT3 can verify system properties including deadlockfreeness, confluence, termination, determinism, goal reachability, and other properties specified in LTL. Refinement checking can also be performed in PAT3. Making use of processes and events, PAT3 can potentially model both intercellular communications and object manipulations in cP systems.

ProB is a validation toolset for the B-method [41]. A model checker and a refinement checker are contained by ProB, which can be used to detect design errors in B specifications. As a modelling language, B makes heavy use of set theory, this can potentially benefit the modelling of certain cP systems. Two main proof activities in ProB are consistency checking and refinement checking. Here consistency checking monitors if the invariants are preserved during operations, and refinement checking is used to check if a machine is a refinement of another. In addition to B, the latest version of ProB also supports other modelling languages including Event-B, CSP-M, TLA+, and Z.

2.1.4.2 Proof assistants

A proof assistant, or interactive theorem prover, is a software tool that helps people to write formal proofs. Existing proof assistants include, but are not limited to: (1) A Computational Logic for Applicative Common Lisp (ACL2) [42], which was implemented in Common Lisp, and supports automated reasoning in inductive logical theories; (2) Agda [43], which is a proof assistant based on the propositions-as-types paradigm; (3) F* [44, 45], which is a functional programming language in development, aimed at software and hardware verification; (4) Higher Order Logic (HOL) [46], which includes a family of LCF (Logic for Computable Functions) theorem proving systems for higher-order logic; (5) Isabelle [47], which is also a LCF-style higher-order logic theorem prover; and (6) Mizar [48, 49], which consists of a formal language and a theorem prover based on first-order logic and Tarski–Grothendieck set theory.

In this study, the Coq proof assistant [50, 51] is used to verify cP systems. Coq is an interactive theorem prover designed for both first-order logic and higher-order logic. It supports a system specification language named Gallina, and a tactic language called the Calculus of Inductive Constructions (CIC) [50]. Programs written

in Gallina always terminate, which satisfy the weak normalisation property. Coq was successfully applied to several problematic domains including compiler verification [52], data structure verification [53], and mathematical theorem proving such as Feit–Thompson theorem [54] and four color theorem [55].

2.2 Related work

This section includes previous studies related to this thesis. Three major topics will be included in this section, which are first-order unification algorithms, formal verification of different membrane computing models, and the Knuth-Bendix completion.

Different from other P system variants, the hierarchical membrane structures with objects in cP systems are represented by labelled (and nested) multiset-based terms. Compared to other P system variants, which typically use a uniform or semi-uniform family of rulesets to solve a specific instance of a NPC problem, cP systems can provide a simpler solution with a fixed constant number of generic rules that covers all the instances of a problem. To simulate or verify a cP system with generic rules, first-order unification plays an essential role. Terms in a rule's lhs and promoters need to be unified against system terms, and the unifiers obtained from the unification need to be applied to the rule's rhs terms.

Although there is no previous work focusing on the formal verification of cP systems, multiple variants of P systems were successfully verified using formal tools. Since membrane computing was proposed, model checking became the most popular and successful formal approach to verifying P system models. Various general purpose model checkers and domain specific verifiers (built on top of existing model checkers) were used to conduct formal verification of P systems. These include SPIN, NuSMV, ProB, UPPAAL, MeCoSim [56, 57] and kPWorkbench [58].

In addition to model checking, this thesis also introduces how to verify cP systems by performing interactive theorem proving. It also discusses an ambitious future direction: using cP systems to construct an automated theorem proving tool and using this to verify other cP systems. Equational logic is a common denominator among several different logics [59], and equational deduction is fundamental to automated theorem proving. Using the Knuth-Bendix completion algorithm [60], a set of equations (axioms) over terms can be transformed into a confluent rewrite system, which can be use to derive new equations (theorems).

2.2.1 First-order unifcation algorithms

Unification identifies two symbolic expressions by finding variable substitutions. In first-order unification (or syntactic unification of first-order terms), variables cannot map to function symbols. In other words, higher-order variables, i.e., variables for functions, are not allowed.

Classical first-order unification algorithms were surveyed by Baader and Snyder [61], and several studies analysed the complexity of unification problems [62, 63, 64, 65, 66, 67]. Robinson first proposed an algorithm to compute most general unifiers (mgu) of well-formed expressions in a set. This was based on substituting variables in the lexical ordering of the disagreement sets [68, 69]. The notions of unification and mgu were also reinvented by Knuth and Bendix as tools for computing critical pairs of equational theories [60]. Zilli described a simplified version of Robinson's algorithm and discussed its complexity [70], and Paterson and Wegman introduced a linear unification algorithm, which can deal with simple terms (represented by directed trees) on the rhs of multiequations [71].

A well-known nondeterministic first-order unification algorithm is Martelli and Montanari's algorithm [72]. A strategy of efficiently selecting multiequations was also introduced by the authors. The algorithm was proven to be efficient in several extreme cases.

Dovier et al. discussed unification algorithms for several data structures including sets, multisets, and compact lists [73]. Dantsin and Voronkov also described a unification algorithm for sets, multisets and trees [74].

Other first-order unification algorithms include, but are not limited to: Jaffar's nondeterministic algorithm, which can deal with infinite terms without complex data structures [75]; Dwork et al.'s parallel term matching algorithm [76]; Rydeheard and Burstall's categorical unification algorithm [77], and Huet's algorithm [78].

2.2.2 Formal verification of membrane computing models

One of the earliest studies of verifying P systems was conducted by Pérez-Jiménez and Sancho-Caparrini [79]. Here the properties of a P system generating squares of natural numbers were proven manually by the authors without using any software tools. Later, Kefalas et al. described an approach to modelling P systems as communicating X-machines, this aimed to facilitate future formal verification of P systems [80].

Since formal tools were introduced to verify P systems, model checking became the most dominant approach. By modelling P systems in Maude, which is an implementation of rewriting logic, Andrei et al. showcased how to specify and verify properties of P systems using a model checker [81]. Various P systems and their model checking problems were investigated by Dang et al., where SPIN was used to conduct the experiments. Ipate et al. introduced a testing methodology for P systems based on model checking, which was implemented in NuSMV [82]. A comparison between P system verification using SPIN and NuSMV was presented in [83], and a guideline for specifying P systems in Process or Protocol Meta Language (PROMELA) was proposed.

An approach to verifying P systems using ProB and Rodin was proposed in [84], where the authors introduced how to represent P systems as Event-B models. An automated approach to transform P systems from P-Lingua [56, 57] into Promela was proposed by Lefticaru et al., where the generated models can be verified using SPIN [85]. An algorithm for transforming spiking neural P systems into timed automata was propose by Aman and Ciobanu, this can be verified using UPPAAL [86].

Multiple domain-specific tools including MeCoSim and kPWorkbench were also used to simulate and verify certain P system models. For example, Gheorghe et al. simulated and verified a kernel P system that solves the 3-colouring problem using MeCoSim and SPIN [87]. Lefticaru et al. introduced an approach to mapping certain classes of spiking neural P systems to equivalent kernel P system representations, this can be verified in kPWorkbench [88].

2.2.3 Automated deduction and Knuth-Bendix completion

Automated deduction studies how mathematical theorems can be proven by computer programs. Many early implementations of automated deduction were based on a Herbrand's theorem, which describes how to build a sound and complete deduction procedure [89]. A notable milestone was Robinson's resolution method, this provided a refutation technique for sentences in propositional logic and first-order logic [68]. Another breakthrough on automated theorem proving was Knuth and Bendix's work, this described a superposition procedure that can be used to obtain a convergent system from a set of axioms, namely completion [60]. Properties of convergent systems, the superposition algorithm, and the Knuth-Bendix theorem were further formalised by Huet and Oppen [90].

Term rewriting provides a forward chaining method for automated deduction [91, 92]. This can also be seen as a subgoal-reduction strategy with a linear input [93]. A potential issue of this approach is combinatorial explosion, however by only computing necessary critical pairs and applying simplification techniques in equational deduction, the issue can be alleviated. Rewriting rules or equations can be used to simplify each other, i.e., after obtaining a new rule, it will be used to reduce existing rules, then redundant rules can be deleted.

The Knuth-Bendix completion does not guarantee success: it may either be non-terminating (trying to generate an infinite number of new axioms), or fail to handle unorientable equations. By relaxing the problem to confluent rewrite systems, which are not necessarily convergent, Huet proposed a completion algorithm which applies to left-linear rewrite systems [94]. Peterson and Stickel introduced an extended version of Knuth and Bendix's algorithm to handle concepts such as commutativity (non-terminating rewrite relations) [95]. Bachmair et al. proposed an "unfailing" extension of the classic Knuth-Bendix completion procedure, which is refutationally complete for theorem proving in equational theories. [96]. Other studies related to completion include, but are not limited to: [97, 98, 99, 100, 101, 102].

2.3 Summary

This chapter introduces the notation of P systems and cP systems, formal verification, existing formal verification tools, and related work (on unification, formal verification of P systems, and automated deduction).

Previous studies on classic first-order unification were reviewed, and several famous unification algorithms, including Robinson's algorithm, Paterson and Wegman's linear unification algorithm, and Martelli and Montanari's nondeterministic unification algorithm, were introduced.

To verify P systems, model checking was considered as an effective method. Several P systems were verified using different model checkers including SPIN, NuSMV, ProB, and UPPAAL, which were introduced in this chapter.

Automated deduction is a subfield of automated reasoning and mathematical logic, which studies how to use computer programs to write formal proofs. Rewrite systems can be used as efficient tools to perform automated deduction. cP systems can be seen as a type of maximally parallel rewrite systems, which can perform automated deduction more efficiently than traditional rewrite systems. As a core

algorithm and a milestone of automated deduction, the Knuth-Bendix completion and its extensions were introduced.

In the next chapter, we will formally define the unification problem of labelled multisets, introduce a corresponding unification algorithm, and analyse the complexity of the algorithm.

Chapter 3

An efficient labelled multiset unification algorithm

cP systems support complex symbols and generic rules. Terms in cP systems are based on labelled multisets, which are often nested, and can be used to represent membrane structures with objects. Generic rules may contain first-order variables, where "first-order" indicates that variables in a rule can only be mapped to a multiset of objects (cell content) rather than functors (cell or subcell labels).

To apply a generic rule in cP systems, the most important process is to unify the rule's lhs terms and promoters against system terms. Unification solves equations between symbolic expressions by finding variable substitutions [61]. Suppose we have two terms $t_1 = f(a, Y)$ and $t_2 = f(X, b)$, where f is a functor, a, b are atoms, and X, Y are variables. By applying the substitution $\vartheta := \{X \mapsto a, Y \mapsto b\}$, we have $t_1\vartheta = f(a, b) = t_2\vartheta$. Thus, ϑ is said to be a unifier for $t_1 = t_2$.

In this chapter, we will extend the first-order unification problem to labelled (and nested) multisets, and introduce a corresponding algorithm, namely LNMU [103, 7]. The transformations in LNMU will be discussed one by one, and we will introduce how to effectively select transformations. LNMU can solve well-formed (defined in Section 3.4) labelled multiset unification problems in linear time.

The rest of the chapter is organised as follows. Section 3.1 defines the labelled multiset unification problem. Section 3.2 introduces the LNMU algorithm. Section 3.3 shows the analysis of LNMU. Section 3.4 discusses the well-formedness of labelled multiset unification. Section 3.5 shows two worked examples of the algorithm, and Section 3.6 concludes the chapter.

3.1 Definitions

Multisets are unordered set-like collections, where multiplicities of their elements matter. For labelled multisets, each of their elements can be an atom, a variable symbol or another labelled multiset.

Similar to the term definition in cP systems (Fig. 2.2), we consider *functors* as *labels* of multisets. For example: $f(abc)$ is a multiset labelled with f, which contains three atom a, b, and c. Similarly, $f(g(h))$ is a multiset labelled with f, which contains a submultiset with the label g, which contains an atom h. Multisets are unordered collections, for instance, both $f(aab)$ and $f(baa)$ can be used to represent a multiset labelled with f, which contains three atoms a, a, and b.

Function symbols discussed in other studies are different from functors defined in this study. In traditional first-order unification, to unify $f_1(a,b)$ with $f_1(X,Y)$, the only unifier is $\vartheta := \{X \mapsto a, Y \mapsto b\}$. Since f_1 is a function symbol, the order of its parameters matters: $f_1(a,b) \neq f_1(b,a)$. In the labelled multiset unification, since multisets are unordered, to unify $f_2(ab)$ with $f_2(XY)$, we can get four unifiers, these are $\vartheta_1 := \{X \mapsto a, Y \mapsto b\}$, $\vartheta_2 := \{X \mapsto b, Y \mapsto a\}$, $\vartheta_3 := \{X \mapsto \lambda, Y \mapsto ab\}$, and $\vartheta_4 := \{X \mapsto ab, Y \mapsto \lambda\}$. The functor f_2 is a multiset label, and λ represents the empty multiset.

Apparently "arities" or "ranks" do not matter in labelled multiset unification. For instance, it is possible to unify $f(XY)$ with $f(abc)$ to find eight unifiers including $\vartheta_1 := \{X \mapsto \lambda, Y \mapsto abc\}$, $\vartheta_2 := \{X \mapsto a, Y \mapsto bc\}$, $\vartheta_3 := \{X \mapsto b, Y \mapsto ac\}$, $\vartheta_4 := \{X \mapsto c, Y \mapsto ab\}$, $\vartheta_5 := \{X \mapsto ab, Y \mapsto c\}$, $\vartheta_6 := \{X \mapsto ac, Y \mapsto b\}$, $\vartheta_7 := \{X \mapsto bc, Y \mapsto a\}$, and $\vartheta_8 := \{X \mapsto abc, Y \mapsto \lambda\}$.

In the rest of this section, we will formally define labelled multisets and their unification problem.

3.1.1 labelled multisets and multiset-based terms

labelled multisets and multiset-based terms are defined by the grammar given in Fig. 3.1. The prefix "g" refers to "ground" and "v" means "variable". Multisets or terms that do not contain variables are called *ground*. We use lowercase letters to represent atoms, and uppercase letters to represent variables. For example: a, b, c are atoms, and X, Y, Z are variables. The unity symbol *1* is defined as a special atom, and "_" is used to denote an anonymous variable. As mentioned, the symbol λ refers to an empty multiset.

$< multiset > ::= < g\text{-}multiset > | < v\text{-}multiset >$
$< term > ::= < g\text{-}term > | < v\text{-}term >$
$< functor > ::= < atom >$

$< g\text{-}multiset > ::= < g\text{-}term > \ldots$
$< g\text{-}term > ::= < simple\text{-}g\text{-}term > | < compound\text{-}g\text{-}term >$
$< simple\text{-}g\text{-}term > ::= < atom >$
$< compound\text{-}g\text{-}term > ::=$
$\quad < functor > (< g\text{-}multiset >) \ | \ < functor > \{< g\text{-}multiset >\}$

$< v\text{-}multiset > ::= < v\text{-}term > \ldots$
$< v\text{-}term > ::= < simple\text{-}v\text{-}term > | < compound\text{-}v\text{-}term >$
$< simple\text{-}v\text{-}term > ::= < atom > | < variable >$
$< compound\text{-}v\text{-}term > ::=$
$\quad < functor > (< v\text{-}multiset >) \ | \ < functor > \{< v\text{-}multiset >\}$

Figure 3.1: The grammar of labelled multisets and multiset-based terms

A *ground multiset (g-multiset)* contains zero or more *ground terms (g-terms)*. A g-term can either be a *simple-g-term* or a *compound-g-term*. Simple-g-terms are atoms, such as: *1*, *a*, *b* and *c*. Compound-g-terms are recursively built by g-multisets and functors, where functors are atoms. For example: $f(a)$, $g\{a1^3\}$ and $f(g(h\{ab\}))$ are compound-g-terms. Superscripts are used to denote term multiplicities, for example, $1^3 \equiv 111$ and $a^2 \equiv aa$.

Multisets with round parentheses "(" and ")" are *complete* multisets. For instance, $a(b)$ is a multiset labelled with a, which only contains an atom b. Curly braces "{" and "}" denote *partial* multisets, for example, $a\{b\}$ is a multiset labelled with a that contains an atom b, which may also contain something else.

Variable multisets (v-multisets) contain zero or more *variable terms (v-terms)*, where v-terms can be *simple-v-terms* or *compound-v-terms*. Simple-v-terms can be atoms or variables, for instance: a, X, and Y. Compound-v-terms are built by v-multisets and functors, for example, $f(X)$, $g\{XY^2\}$ and $f(g\{1\}h(aX))$.

The grammar only includes first-order variables. Terms such as $X()$, $Y(ab)$, $Z(X\{rs\})$ are invalid.

3.1.2 labelled multiset unification

A traditional definition of unification is: a process that solves equations between symbolic expressions by finding variable substitutions. In this study, we extend

the definition to solve *equations* and *inclusion relations* for labelled multisets.

This design facilitates the use of LNMU in cP systems, which always match v-terms in a rule with a subset of system terms (g-multiset). Theoretically, an inclusion relation can be equivalently transformed into an equation by adding one anonymous variable. For example, $f(X) \subset f(a)g(b)$ can be transformed into $f(X)\_ = f(a)g(b)$.

We define the labelled multiset unification problem as *one-way*, which is also known as a *matching problem*. This problem definition can be extended to two-way as needed, and we can slightly modify the algorithm to solve it.

Let $\Sigma = \{1, a, b, c, \ldots\}$ be the alphabet of atoms, and $V = \{X, Y, Z, \ldots\}$ be the alphabet of variables. Let M_Σ be the collection of all g-multisets, and $M_{\Sigma\cup V}$ be the collection of all v-multisets.

A substitution ϑ is a mapping from variables to g-multisets: $\vartheta = \{X_1 \mapsto m_1, X_2 \mapsto m_2, \ldots, X_n \mapsto m_n\}$, where X_1, X_2, ..., $X_n \in V$, and m_1, m_2, ..., $m_n \in M_\Sigma$.

Given a labelled multiset equation or inclusion relation which may contain variables, *labelled multiset unification* solves it by finding variable substitutions. For one-way unification, the equation or inclusion relation is in the form of $m_v \stackrel{=}{\subset} m_g$, where $m_v \in M_{\Sigma\cup V}$, and $m_g \in M_\Sigma$. Both m_v and m_g contain finite numbers of symbols.

The appearance of the metasyntactic symbol $\stackrel{=}{\subset}$ in Fig. 3.2 actually defines two transformations: where $\stackrel{=}{\subset}$ can either be uniformly replaced by "$=$" or "$\subset$". For example, the transformation GROUND can either be interpreted as $G \cup \{mm_v = mm_g\} \Longrightarrow G \cup \{m_v = m_g\}, m \neq \lambda,\ m_v \cap m_g = \varnothing$, or $G \cup \{mm_v \subset mm_g\} \Longrightarrow G \cup \{m_v \subset m_g\}, m \neq \lambda,\ m_v \cap m_g = \varnothing$.

A substitution ϑ that contains bindings for all the variables that appear in m_v is called a *unifier* for $m_v \stackrel{=}{\subset} m_g$, if by applying it we have $m_v\vartheta \stackrel{=}{\subset} m_g$, where $m_v\vartheta$ represents ϑ's application to the multiset m_v.

Suppose $m_v = f(X)Y$ and $m_g = 1f(a)f(b)$, to solve the inclusion relation $m_v \subset m_g$, eight unifiers can be found. These are: $\vartheta_1 = \{X \mapsto a, Y \mapsto \lambda\}$, $\vartheta_2 = \{X \mapsto a, Y \mapsto 1\}$, $\vartheta_3 = \{X \mapsto a, Y \mapsto f(b)\}$, $\vartheta_4 = \{X \mapsto a, Y \mapsto 1f(b)\}$, $\vartheta_5 = \{X \mapsto b, Y \mapsto \lambda\}$, $\vartheta_6 = \{X \mapsto b, Y \mapsto 1\}$, $\vartheta_7 = \{X \mapsto b, Y \mapsto f(a)\}$, and $\vartheta_8 = \{X \mapsto b, Y \mapsto 1f(a)\}$. By applying these unifiers, we can get $m_v\vartheta_i \subset m_g$, where $1 \leq i \leq 8$. For example, $m_v\vartheta_1 = f(a) \subset m_g$, $m_v\vartheta_4 = f(a)1f(b) \subset m_g$, and $m_v\vartheta_7 = f(b)f(a) \subset m_g$.

The notion of the *most general unifier* (*mgu*), mentioned by Robinson [68, 69], is not applicable to matching problems. For a two-way unification problem, for

instance, to unify $f(X)$ with $f(Y)$, we can get infinite unifiers such as $\sigma = \{X \mapsto Y\}$, $\vartheta = \{X \mapsto a, Y \mapsto a\}$, $\omega = \{X \mapsto b, Y \mapsto b\}$. We can get ϑ or ω by further substituting a or b for both X and Y in σ; thus, we say σ is more general than ϑ and ω, thus it is an mgu. For a matching problem, its unifiers do not contain mappings from a variable to other variables (e.g. $\{X \mapsto Y\}$, $\{X \mapsto ZW\}$); thus, the unifiers can not be further substituted – none of them is more general than the others.

3.2 LNMU – an efficient labelled multiset unification algorithm

The *nondeterministic algorithm* we proposed in this study is called Labelled (and Nested) Multiset Unification algorithm (LNMU), Fig. 3.2). This algorithm can solve labelled multiset unification problems.

For a labelled multiset unification problem $\mathcal{P} = \{m_v \overline{\overline{\subset}} m_g\}$, $m_v \in M_{\Sigma \cup V}$, $m_g \in M_\Sigma$, we use LNMU($\mathcal{P}$) to denote "using LNMU to solve $\mathcal{P}$". The collection of all branches corresponding to all possible executions form a tree called the computation tree, which is denoted by $\mathcal{T}$(LNMU($\mathcal{P}$)).

$G_\mathcal{P}$ is the goal set which may contain equations, inclusion relations, and variable bindings. At the beginning of unification, $G_\mathcal{P}$ contains $m_v \overline{\overline{\subset}} m_g$. During the unification, $G_\mathcal{P}$ will successively transform into other forms, which may include variable bindings.

We use $\mathcal{E}$ to denote the collection of all labelled multiset equations and inclusion relations. We use m, m_v, m'_v, m_g, m'_g to represent labelled multisets, where $m_v, m'_v \in M_{\Sigma \cup V}$, and $m, m_g, m'_g \in M_\Sigma$. The symbol $\square$ denotes "success", and $\perp$ denotes "stop with failure". $\varphi(m_g)$ is a set which contains all the functors (labels) of m_g's first-level compound terms, for example, $\varphi(a1f(b)g(h(c))) = \{f, g\}$.

Before immediately diving into the transformations of LNMU, we will first introduce the *size measures* of $\mathcal{P}$ and $G_\mathcal{P}$.

We use $|m_s|$ to denote the size of a labelled multiset m_s. This refers to the total number of atoms, functors and variable symbols in m_s. For example, $|\lambda| = 0$; $|1| = |a| = |f(\lambda)| = 1$; $|aa| = |a(b)| = |f(X)| = 2$; $|a^3| = |f(1X)| = |f(g\{h(\lambda)\})| = 3$.

The size of an equation or inclusion relation $m_v \overline{\overline{\subset}} m_g$, denoted by $|m_v \overline{\overline{\subset}} m_g|$, is defined as $|m_v| + |m_g| + 1$. For example, $|\lambda = \lambda| = 0 + 0 + 1 = 1$, $|\lambda \subset a| = 0 + 1 + 1 = 2$, and $|g(X) \subset g(b)g(c)| = 2 + 4 + 1 = 7$.

1. SUCCESS
$G_{\mathcal{P}} \cap \mathcal{E} = \varnothing \Longrightarrow \square,\ \vartheta = G_{\mathcal{P}}$

2. DELETE-1
$G_{\mathcal{P}} = G \cup \{\lambda \Subset \lambda\} \Longrightarrow G'_{\mathcal{P}} = G$

3. DELETE-2
$G_{\mathcal{P}} = G \cup \{\lambda \subset m\} \Longrightarrow G'_{\mathcal{P}} = G,\ m \neq \lambda$

4. GROUND
$G_{\mathcal{P}} = G \cup \{mm_v \Subset mm_g\} \Longrightarrow G'_{\mathcal{P}} = G \cup \{m_v \Subset m_g\},\ m \neq \lambda,\ m_v \cap m_g = \varnothing$

5. FAIL-1
$G_{\mathcal{P}} = G \cup \{mm_v \Subset m_g\} \Longrightarrow \perp,\ m \not\subset m_g$

6. FUNCTOR (nondeterministic)
$G_{\mathcal{P}} = G \cup \{f(m'_v)m_v \Subset f(m'_g)m_g\} \Longrightarrow G'_{\mathcal{P}} = G \cup \{m_v \Subset m_g\} \cup \{m'_v = m'_g\}$

7. FAIL-2
$G_{\mathcal{P}} = G \cup \{f(m'_v)m_v \Subset m_g\} \Longrightarrow \perp,\ f \notin \varphi(m_g)$

8. VARIABLE (nondeterministic)
$G_{\mathcal{P}} = G \cup \{X^n m_v \Subset m^n m_g\} \Longrightarrow$
$G'_{\mathcal{P}} = G\vartheta_X \cup \{m_v\vartheta_X \Subset m_g\} \cup \vartheta_X,\ \vartheta_X = \{X \mapsto m\},\ X \in V,\ n \in \mathbb{N}^*$

Figure 3.2 LNMU – a nondeterministic unification algorithm

For $\mathcal{P} = \{m_v \Subset m_g\}$, we use $|\mathcal{P}|$ to denote the size of $\mathcal{P}$, where $|\mathcal{P}| = |m_v \Subset m_g|$.

Suppose $G_{\mathcal{P}}$ contains k equations and inclusion relations $m_{v_i} \Subset m_{g_i}$ $(i = 1, 2, \ldots, k)$, the size of $G_{\mathcal{P}}$, denoted by $|G_{\mathcal{P}}|$, is defined as $\sum_{i=1}^{k} |m_{v_i} \Subset m_{g_i}|$. The sizes of variable bindings in $G_{\mathcal{P}}$ are defined as zeros. For example, if $G_{\mathcal{P}} = \{\lambda \subset f(a)\} \cup \{g(X) \subset g(b)g(c)\} \cup \{Y \mapsto d\}$, then $|G_{\mathcal{P}}| = |\lambda \subset f(a)| + |g(X) \subset g(b)g(c)| + |\{Y \mapsto d\}| = 3 + 7 + 0 = 10$.

We use $G_{\mathcal{P}} \overset{\mathrm{T}}{\Longrightarrow} G'_{\mathcal{P}}$ to denote that "by applying the transformation T, $G_{\mathcal{P}}$ will be transformed into $G'_{\mathcal{P}}$".

1. **SUCCESS**: *When $G_{\mathcal{P}} \cap \mathcal{E} = \varnothing$, LNMU succeeds and outputs a substitution*

ϑ.

After solving all the equations and inclusion relations in $G_{\mathcal{P}}$ (reducing them to $\lambda \overline{\overline{\subset}} \lambda$ or $\lambda \subset m$ and deleting them), $G_{\mathcal{P}}$ will only contain variable bindings $\vartheta_{X_1}, \vartheta_{X_2}, \ldots \vartheta_{X_n}$. LNMU will collect these bindings as a substitution ϑ. We will prove that ϑ is a unifier for $\mathcal{P}$ in Theorem 3.3.2 .

Lemma 3.2.1. *If $|G_{\mathcal{P}}| = 0$, then LNMU will succeed.*

Proof. $|G_{\mathcal{P}}| = 0 \iff G_{\mathcal{P}} \cap \mathcal{E} = \varnothing \Longrightarrow \square$, $\vartheta = G_{\mathcal{P}}$. □

2. **DELETE-1**: *If $G_{\mathcal{P}}$ contains $\lambda \overline{\overline{\subset}} \lambda$, then we can delete it from $G_{\mathcal{P}}$.*

   Example: to solve $G_{\mathcal{P}} = \{\lambda = \lambda\} \cup \{X = a\}$, we can delete $\{\lambda = \lambda\}$ from $G_{\mathcal{P}}$, then we only need to solve $X = a$.

   Lemma 3.2.2. *If $G_{\mathcal{P}} \xrightarrow{DELETE\text{-}1} G'_{\mathcal{P}}$ and ϑ unifies $G'_{\mathcal{P}}$, then ϑ unifies $G_{\mathcal{P}}$.*

   Proof. (Any substitution unifies $\lambda \overline{\overline{\subset}} \lambda$) $\Longrightarrow$ (ϑ unifies $\lambda \overline{\overline{\subset}} \lambda$).

   (ϑ unifies $G'_{\mathcal{P}} = G$) $\wedge$ (ϑ unifies $\lambda \overline{\overline{\subset}} \lambda$) $\Longrightarrow$ (ϑ unifies $G \cup \{\lambda \overline{\overline{\subset}} \lambda\} = G_{\mathcal{P}}$). □

   Lemma 3.2.3. *If $G_{\mathcal{P}} \xrightarrow{DELETE\text{-}1} G'_{\mathcal{P}}$, then $|G'_{\mathcal{P}}| = |G_{\mathcal{P}}| - 1$.*

   Proof. $(|G'_{\mathcal{P}}| = |G|) \wedge (|G_{\mathcal{P}}| = |G| + |\lambda \overline{\overline{\subset}} \lambda| = |G| + 1) \Longrightarrow (|G'_{\mathcal{P}}| = |G_{\mathcal{P}}| - 1)$. □

3. **DELETE-2**: *If $G_{\mathcal{P}}$ contains $\lambda \subset m$, $m \neq \lambda$, then we can delete it from $G_{\mathcal{P}}$.*

   Example: to solve $G_{\mathcal{P}} = \{\lambda \subset a\} \cup \{X = a\}$, we can delete $\{\lambda \subset a\}$ from $G_{\mathcal{P}}$, then we only need to solve $X = a$.

   Lemma 3.2.4. *If $G_{\mathcal{P}} \xrightarrow{DELETE\text{-}2} G'_{\mathcal{P}}$ and ϑ unifies $G'_{\mathcal{P}}$, then ϑ unifies $G_{\mathcal{P}}$.*

   Proof. Similar to Lemma 2, any substitution unifies $\lambda \subset m$ (λ is included in any multiset). □

   Lemma 3.2.5. *If $G_{\mathcal{P}} \xrightarrow{DELETE\text{-}2} G'_{\mathcal{P}}$, then $|G'_{\mathcal{P}}| = |G_{\mathcal{P}}| - (|m| + 1)$.*

Proof. $(|G'_{\mathcal{P}}| = |G|) \wedge (|G_{\mathcal{P}}| = |G| + |\lambda \subset m| = |G| + (|m| + 1)) \Longrightarrow (|G'_{\mathcal{P}}| = |G_{\mathcal{P}}| - (|m| + 1))$. □

4. **GROUND**: *$mm_v \overline{\overline{\subset}} mm_g$ in $G_{\mathcal{P}}$ can be transformed into $m_v \overline{\overline{\subset}} m_g$, where $m \neq \lambda$, $m_v \cap m_g = \varnothing$.*

   The transformation GROUND simplifies an equation or inclusion relation by deleting m from both its lhs and rhs.

   Example: to solve $g(a)f(X) \subset g(a)f(b)$, we can eliminate $g(a)$ from its lhs and rhs, and transform it into $f(X) \subset f(b)$.

   Lemma 3.2.6. *If $G_{\mathcal{P}} \xrightarrow{GROUND} G'_{\mathcal{P}}$ and ϑ unifies $G'_{\mathcal{P}}$, then ϑ unifies $G_{\mathcal{P}}$.*

   Proof. Since $mm_v \overline{\overline{\subset}} mm_g \iff m_v \overline{\overline{\subset}} m_g$, $G_{\mathcal{P}}$ and $G'_{\mathcal{P}}$ are semantically equal. □

   Lemma 3.2.7. *If $G_{\mathcal{P}} \xrightarrow{GROUND} G'_{\mathcal{P}}$, then $|G'_{\mathcal{P}}| = |G_{\mathcal{P}}| - |m| \times 2$.*

   Proof. $(|G'_{\mathcal{P}}| = |G| + |m_v \overline{\overline{\subset}} m_g| = |G| + (|m_v| + |m_g| + 1))$
   $\wedge (|G_{\mathcal{P}}| = |G| + |mm_v \overline{\overline{\subset}} mm_g| = |G| + (|m| + |m_v| + |m| + |m_g| + 1))$
   $\Longrightarrow (|G'_{\mathcal{P}}| = |G_{\mathcal{P}}| - |m| \times 2)$. □

5. **FAIL-1**: *If $G_{\mathcal{P}}$ contains $mm_v \overline{\overline{\subset}} m_g$, $m \not\subset m_g$, then LNMU terminates with failure.*

   Example: $af(X) \overline{\overline{\subset}} f(1)$ is unsolvable. No matter what multisets we substitute for X, $f(1)$ does not contain a.

   Lemma 3.2.8. *If $G_{\mathcal{P}} = G \cup \{mm_v \overline{\overline{\subset}} m_g\}$, $m \not\subset m_g$, then no substitution unifies $G_{\mathcal{P}}$.*

   Proof. Since m and m_g are ground, no substitution unifies $mm_v \overline{\overline{\subset}} m_g$, $m \not\subset m_g$. Thus, no substitution unifies $G_{\mathcal{P}} = G \cup \{mm_v \overline{\overline{\subset}} m_g\}$, $m \not\subset m_g$. □

6. **FUNCTOR**: *$f(m'_v)m_v \overline{\overline{\subset}} f(m'_g)m_g$ in $G_{\mathcal{P}}$ can be transformed into $m_v \overline{\overline{\subset}} m_g$ and $m'_v = m'_g$.*

   The transformation FUNCTOR eliminates functors in an equation or inclusion relation. For example, $Xf(Yc) = af(bc)$ can be transformed into two distinct equations: $X = a$ and $Yc = bc$.

FUNCTOR is *nondeterministic*, which can randomly choose different subterms with the same functor. For instance, $Xf(Y) = af(b)f(c)$ can either be transformed into $\{X = af(b),\ Y = c\}$ or $\{X = af(c),\ Y = b\}$.

Lemma 3.2.9. *If* $G_\mathcal{P} \overset{FUNCTOR}{\Longrightarrow} G'_\mathcal{P}$ *and* ϑ *unifies* $G'_\mathcal{P}$*, then* ϑ *unifies* $G_\mathcal{P}$*.*

Proof. (ϑ unifies $G'_\mathcal{P}$) $\Longrightarrow$ (ϑ unifies G) $\wedge$ $(m_v\vartheta \overline{\overline{\subset}}\ m_g) \wedge (m'_v\vartheta = m'_g)$
$\Longrightarrow$ (ϑ unifies G) $\wedge$ $(f(m'_v\vartheta)m_v\vartheta \overline{\overline{\subset}}\ f(m'_g)m_g)$
$\Longrightarrow$ (ϑ unifies G) $\wedge$ $((f(m'_v)m_v)\vartheta \overline{\overline{\subset}}\ f(m'_g)m_g) \Longrightarrow$ (ϑ unifies $G_\mathcal{P}$). □

Lemma 3.2.10. *If* $G_\mathcal{P} \overset{FUNCTOR}{\Longrightarrow} G'_\mathcal{P}$*, then* $|G'_\mathcal{P}| = |G_\mathcal{P}| - 1$*.*

Proof. $(|G'_\mathcal{P}| = |G| + |m_v \overline{\overline{\subset}}\ m_g| + |m'_v = m'_g| = |G| + (|m_v| + |m_g| + 1) +$
$(|m'_v| + |m'_g| + 1)) \wedge (|G_\mathcal{P}| = |G| + |f(m'_v)\ m_v\ \overline{\overline{\subset}}\ f(m'_g)\ m_g| = |G| + (1 +$
$|m'_v| + |m_v| + 1 + |m'_g| + |m_g| + 1)) \Longrightarrow (|G'_\mathcal{P}| = |G_\mathcal{P}| - 1)$. □

7. **FAIL-2**: *If* $G_\mathcal{P}$ *contains* $f(m'_v)m_v \overline{\overline{\subset}}\ m_g$*,* $f \notin \varphi(m_g)$*, then LNMU terminates with failure.*

   The transformation FAIL-2 shows that if an equation or inclusion relation's lhs contains a compound term $f(m'_v)$, whose functor f is not in $\varphi(m_g)$, it is unsolvable.

   Example: $f(X)Y \overline{\overline{\subset}}\ g(a)h(b)$ is unsolvable, since $f \notin \{g, h\}$ – no matter what multisets we substitute for X and Y, $g(a)h(b)$ can not contain $f(X)$.

   Lemma 3.2.11. *If* $G_\mathcal{P} = G \cup \{f(m'_v)m_v \overline{\overline{\subset}}\ m_g\}$*,* $f \notin \varphi(m_g)$*, then no substitution unifies* $G_\mathcal{P}$*.*

   Proof. Similar to Lemma 8, no substitution unifies $f(m'_v)m_v \overline{\overline{\subset}}\ m_g$, $f \notin \varphi(m_g)$. □

8. **VARIABLE**: $X^n m_v \overline{\overline{\subset}}\ m^n m_g$, $X \in V$, $n \in \mathbb{N}^*$ *in* $G_\mathcal{P}$ *can be transformed into* $m_v\vartheta_X \overline{\overline{\subset}}\ m_g$*, where* ϑ_X *is a binding* $\{X \mapsto m\}$*. After choosing* ϑ_X*, it will be applied to all the equations and inclusion relations in* $G_\mathcal{P}$*.*

   The transformation VARIABLE reduces equations and inclusion relations by finding variable bindings nondeterministically.

For example, to unify $XY \stackrel{=}{\subset} af(b)$, VARIABLE can save the binding $\vartheta_X = \{X \mapsto a\}$ into $G_\mathcal{P}$, and reduce $XY \stackrel{=}{\subset} af(b)$ to $Y \stackrel{=}{\subset} f(b)$. VARIABLE is nondeterministic, so other bindings may also be chosen, these include $\vartheta_X = \{X \mapsto \lambda\}$, $\vartheta_X = \{X \mapsto f(b)\}$ and $\vartheta_X = \{X \mapsto af(b)\}$; then $XY \stackrel{=}{\subset} af(b)$ can be reduced to $Y \stackrel{=}{\subset} af(b)$, $Y \stackrel{=}{\subset} a$ and $Y \stackrel{=}{\subset} \lambda$, respectively.

When a binding ϑ_x is chosen, in addition to saving it into $G_\mathcal{P}$, ϑ_x will be applied to all the equations and inclusion relations in $G_\mathcal{P}$. For example, if $\vartheta_X = \{X \mapsto a\}$ is chosen and $G_\mathcal{P}$ contains another equation $f(XZ) = f(ac)$, it will then apply ϑ_X and transform it into $f(aZ) = f(ac)$. VARIABLE will be applied only once for each variable in G.

The multiplicity n of X is considered in VARIABLE. For example, to solve $X^3 f(Y) \stackrel{=}{\subset} 1^4 a^3 f(b)$, VARIABLE can find different bindings for X, which include $\vartheta_X = \{X \mapsto \lambda\}$, $\vartheta_X = \{X \mapsto 1\}$, $\vartheta_X = \{X \mapsto a\}$ and $\vartheta_X = \{X \mapsto 1a\}$. By choosing different bindings, $X^3 f(Y) \stackrel{=}{\subset} 1^4 a^3 f(b)$ can be transformed into $f(Y) \stackrel{=}{\subset} 1^4 a^3 f(b)$, $f(Y) \stackrel{=}{\subset} 1a^3 f(b)$, $f(Y) \stackrel{=}{\subset} 1^4 f(b)$, and $f(Y) \stackrel{=}{\subset} 1f(b)$, respectively.

Lemma 3.2.12. *If* $G_\mathcal{P} \xrightarrow{VARIABLE} G'_\mathcal{P}$, ϑ *unifies* $G'_\mathcal{P}$, *and* $\vartheta' = \vartheta \setminus \vartheta_X$, *then* $\vartheta'\vartheta_X$ *unifies* $G_\mathcal{P}$.

Proof. (ϑ unifies $G'_\mathcal{P}$) $\Longrightarrow$ (ϑ' unifies $G\vartheta_X$) $\wedge$ ($m_v\vartheta'\vartheta_X \stackrel{=}{\subset} m_g$) $\Longrightarrow$ ($\vartheta'\vartheta_X$ unifies G) $\wedge$ ($(X^n m_v)\vartheta'\vartheta_X \stackrel{=}{\subset} m^n m_g$) $\Longrightarrow$ ($\vartheta'\vartheta_X$ unifies $G_\mathcal{P}$). □

Lemma 3.2.13. *If* $G_\mathcal{P} \xrightarrow{VARIABLE} G'_\mathcal{P}$, *then* $|G'_\mathcal{P}| = |G_\mathcal{P}| - |m| \times (n - k_1 - k_2) - (n + k_1 + k_2)$, *where* k_1 *and* k_2 *denote the number of occurrences of* X *in* G *and* m_v, *respectively.*

Proof. $|G\vartheta_X| = |G| + (|m| - 1) \times k_1$.

$|m_v\vartheta_X| = |m_v| + (|m| - 1) \times k_2$.

$|G'_\mathcal{P}| = |G\vartheta_X| + |m_v\vartheta_X| + |m_g| + 1 = |G| + (|m| - 1) \times k_1 + |m_v| + (|m| - 1) \times k_2 + |m_g| + 1 = |G| + |m_v| + |m_g| + (|m| - 1) \times (k_1 + k_2) + 1$.

$|G_\mathcal{P}| = |G| + n + |m_v| + |m| \times n + |m_g| + 1$.

$(|G'_\mathcal{P}| - |G_\mathcal{P}| = (|m| - 1) \times (k_1 + k_2) - (|m| + 1) \times n) \Longrightarrow (|G'_\mathcal{P}| = |G_\mathcal{P}| - |m| \times (n - k_1 - k_2) - (n + k_1 + k_2))$. □

3.3 Analysis of LNMU

For $\mathcal{P} = \{f(X)XY \subset f(a)ab\}$, a part of its computation tree $\mathcal{T}(\text{LNMU}(\mathcal{P}))$ is shown in Fig. 8.3. LNMU fails in the first and the third branch, and successfully finds a unifier $\vartheta = \{X \mapsto a, Y \mapsto b\}$ in the second branch.

If LNMU only explores one branch of $\mathcal{T}(\text{LNMU}(\mathcal{P}))$, it may either succeed or fail. To avoid early failure, a simple strategy to select transformations from LNMU is to follow their indices. In each unification step, if there are $\lambda \overline{\overline{\subset}} \lambda$ or $\lambda \subset m$, we can delete them from $G_\mathcal{P}$ (DELETE-1 or DELETE-2). For any equation or inclusion relation which cannot be deleted, we use GROUND to eliminate the shared g-multiset of its *lhs* and *rhs*. If they contain functors, we use FUNCTOR to transform them into simpler equations or inclusion relations. When transformations including DELETE-1, DELETE-2, GROUND and FUNCTOR cannot be applied anymore, we use VARIABLE to nondeterministically find a binding for one of its variables, and apply it to all the equations and inclusion relations in $G_\mathcal{P}$. This strategy can be slightly optimised in the following way: if by applying GROUND or FUNCTOR, $G_\mathcal{P}$ contains any equation in the form of $X = m$, it should immediately apply VARIABLE, transform into $\lambda = \lambda$, save $\vartheta_X = \{X \mapsto m\}$ into $G_\mathcal{P}$, and apply ϑ_X to $G_\mathcal{P}$. By doing this, LNMU can prune some failing branches such as $G_\mathcal{P} = \{X = a\} \cup \{X = b\}$.

Actually, all variable bindings can be found by first applying FUNCTION and VARIABLE (before any other transformations). Summing up the lemmas in Section 3, we have:

Table 3.1: How transformations in LNMU affect $|G_\mathcal{P}|$

Transformation	$\lvert G'_\mathcal{P}\rvert - \lvert G_\mathcal{P}\rvert$	**LNMU terminates**
SUCCESS		✓
DELETE-1	-1	
DELETE-2	$-(\lvert m\rvert + 1) < 0$	
GROUND	$-(\lvert m\rvert \times 2) < 0$	
FAIL-1		✓
FUNCTOR	-1	
FAIL-2		✓
VARIABLE	$-\lvert m\rvert \times (n - k_1 - k_2) - (n + k_1 + k_2)$	

Theorem 3.3.1. LNMU($\mathcal{P}$) *terminates, i.e., all branches of* $\mathcal{T}(\text{LNMU}(\mathcal{P}))$ *are finite.*

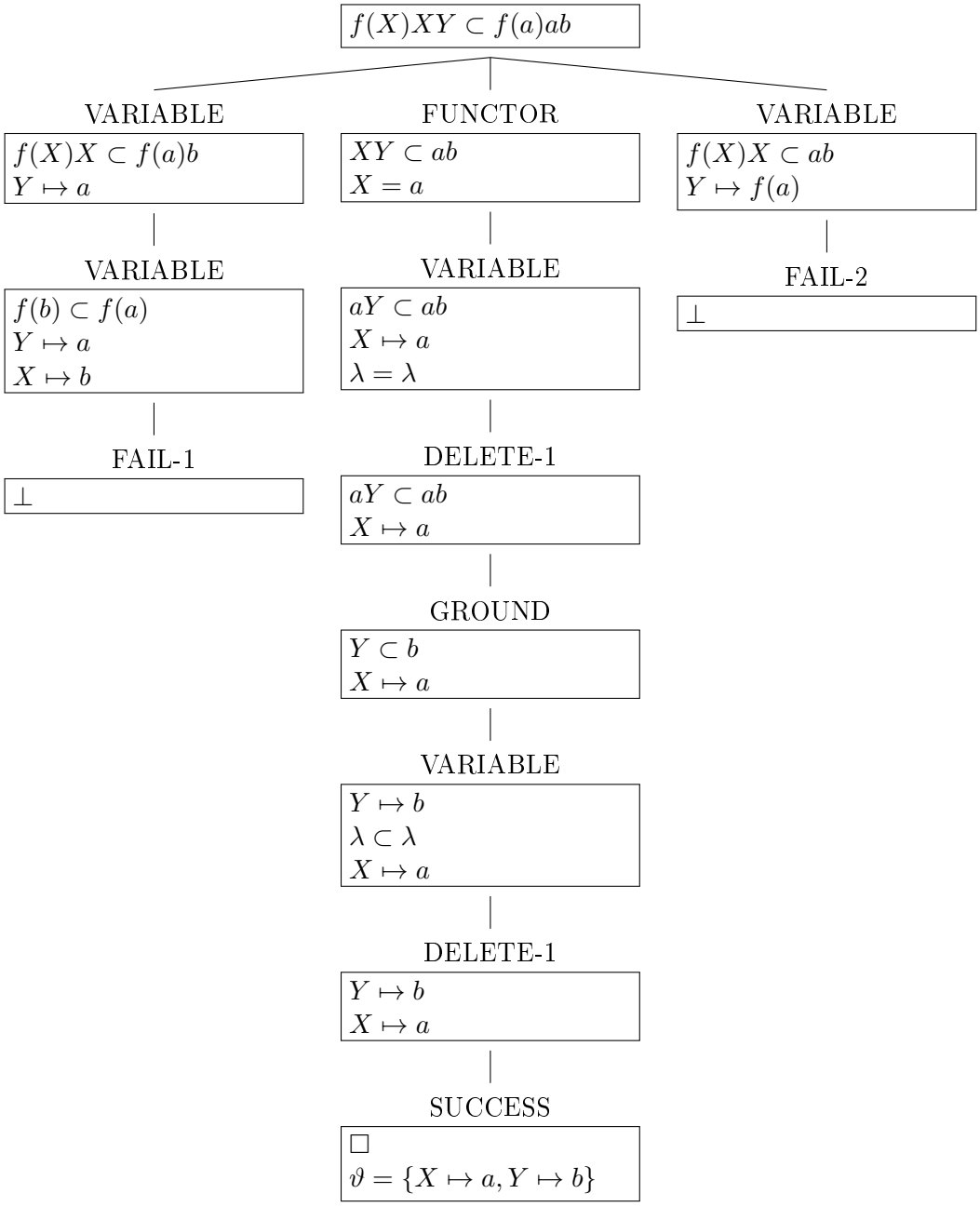

Figure 3.3 Partial view of $\mathcal{T}(\text{LNMU}(\mathcal{P}))$, three branches. $\mathcal{P} = \{f(X)XY \subset f(a)ab\}$

Proof. $\mathcal{P} = \{m_v \stackrel{\equiv}{\subset} m_g\}$, at the beginning, $|G_\mathcal{P}| = |\mathcal{P}|$.

Since the number of variables in $G_\mathcal{P}$ is finite, VARIABLE can only be applied in a finite number of steps. After that, $G_\mathcal{P}$ will not contain any variables; thus, VARIABLE will not be applicable any more – other transformations must be

chosen.

Following Lemmas 3.2.1, 3.2.3, 3.2.5, 3.2.7, 3.2.8, 3.2.10, and 3.2.11, all the LNMU's transformations except VARIABLE will either strictly decrease $|G_{\mathcal{P}}|$ or terminate the algorithm (Table 3.1).

Thus, after a finite number of steps, LNMU($\mathcal{P}$) will either terminate with failure, or decrease $|G_{\mathcal{P}}|$ to 0, and then succeed (Lemma 3.2.1). □

Theorem 3.3.2. *If ϑ is a substitution appearing at the end leaf of a successful branch of $\mathcal{T}$(LNMU($\mathcal{P}$)), then ϑ is a unifier for $\mathcal{P}$.*

Proof. Assume that $G_{\mathcal{P}} = \mathcal{P}$ contains n variables, and consider the transformations along a branch that end with success. Let us highlight the places where VARIABLE is applied: $V_1, V_2, \ldots, V_n$, respectively generating bindings $\theta_1, \theta_2, \ldots, \theta_n$. All transformations that do not involve VARIABLE are represented by stars (*):

$$G_{\mathcal{P}} \stackrel{*}{\Rightarrow} G_{\mathcal{P}'_1} \stackrel{V_1}{\Longrightarrow} G_{\mathcal{P}_1} \stackrel{*}{\Rightarrow} G_{\mathcal{P}'_2} \stackrel{V_2}{\Longrightarrow} G_{\mathcal{P}_2} \cdots G_{\mathcal{P}_{n-1}} \stackrel{*}{\Rightarrow} G_{\mathcal{P}'_n} \stackrel{V_n}{\Longrightarrow} G_{\mathcal{P}_n} \stackrel{*}{\Rightarrow} G_{\mathcal{P}'},$$

where $G_{\mathcal{P}'} \cap \mathcal{E} = \varnothing$. Consider the sequence of substitutions $\vartheta^k = \theta_k\theta_{k+1}\cdots\theta_n$, for $k = n+1, n, \ldots, 1$. Obviously, $\theta^{n+1} = \varnothing$ (the empty substitution), $\vartheta = G_{\mathcal{P}'} = \theta_1\theta_2\cdots\theta_n = \theta^1$.

Using Lemmas 3.2.2, 3.2.4, 3.2.6, 3.2.9, and (critically) 3.2.12, a straightforward induction shows that θ^{n+1} unifies $G_{\mathcal{P}'}$ and $G_{\mathcal{P}_n}$, θ^n unifies $G_{\mathcal{P}'_n}$ and $G_{\mathcal{P}_{n-1}}$, ..., and finally θ^1 unifies $G_{\mathcal{P}'_1}$ and $G_{\mathcal{P}}$.

Thus, ϑ unifies $G_{\mathcal{P}} = \mathcal{P}$. □

Theorem 3.3.3. *If ϑ is a unifier for $\mathcal{P}$, then ϑ is a substitution appearing at the end leaf of a successful branch of $\mathcal{T}$(LNMU($\mathcal{P}$)).*

Proof. Assume $\mathcal{P} = \{m_v \stackrel{=}{\subset} m_g\}$. Let $X_1, X_2, \ldots, X_n$ be the variables in m_v, and let $\vartheta = \vartheta_{X_1}\vartheta_{X_2}\cdots\vartheta_{X_n}$, where $m_i = X_i\vartheta$, $\vartheta X_i = \{X_i \mapsto m_i\}$, for $i = 1, 2, \ldots, n$. The proof runs by induction on n.

Consider $\vartheta_1 = \vartheta \setminus \vartheta_{X_1} = \vartheta_{X_2}\vartheta_{X_3}\cdots\vartheta_{X_n}$. Without loss of generality, since multisets are unordered, we can write $m_v = f_1(f_2(\cdots f_j(X_1^k\, m'_v)\cdots)\cdots)$, with $j \geq 0, k \geq 1$, where dots $(\cdots)$ represent multisets not directly relevant here.

As ϑ is a unifier for $\mathcal{P}$, by applying it to m_v we obtain:

$$\begin{aligned} m_v\vartheta &= m_v\vartheta_{X_1}\vartheta_1 \\ &= f_1(f_2(\cdots f_j(m_1^k\, m'_v\vartheta_{X_1})\cdots)\cdots)\,\vartheta_1 \\ &\stackrel{=}{\subset} f_1(f_2(\cdots f_j(m_1^k\, m'_g)\cdots)\cdots) = m_g. \end{aligned}$$

Thus, by systematically applying FUNCTOR j times, as indicated above, we can obtain the equation $X_1^k\, m'_v = m_1^k\, m'_g$, which generates the binding ϑ_{X_1}, and adds it to the current goal set. One of the equations will contain X_2, and we can repeat the same procedure. Thus, an induction on the remaining variables, $X_2, X_3, \ldots, X_n$, will show that bindings $\vartheta_{X_2}, \vartheta_{X_3}, \ldots, \vartheta_{X_n}$ will also be successively added. Finally, ϑ will appear at the end of a successful branch. □

3.4 Well-formed labelled multiset unification

For $G_{\mathcal{P}} = \mathcal{P} = \{m_v \mathrel{\overline{\overline{\subset}}} m_g\}$, the number of variables in $G_{\mathcal{P}}$ is no more than $|m_v|$, and $|m_v| + |m_g| + 1 = |\mathcal{P}|$. From Table 3.1 we know that VARIABLE is the only transformation that may increase $G_{\mathcal{P}}$.

Suppose that after applying an arbitrary number of transformations, $G_{\mathcal{P}}$ contains $m_{v_i} \mathrel{\overline{\overline{\subset}}} m_{g_i}$ $(i = 1, 2, \ldots, k)$, for any of them we would then have $|m_{g_i}| \leq m_g$.

For variables in m_{v_i}, by applying VARIABLE, three kinds of unifiers result: (1) $\vartheta_X = \{X \mapsto \lambda\}$, (2) $\vartheta_X = \{X \mapsto x\}$, $x \in \Sigma$, (3) $\vartheta_X = \{X \mapsto m\}$, $m \in M_\Sigma$ and $|m| > 1$.

For (1), $|m_{v_i}\vartheta_X| < |m_{v_i}|$; for (2), $|m_{v_i}\vartheta_X| = |m_{v_i}|$; for (3), $|m_{v_i}\vartheta_X| > |m_{v_i}|$. Only (3) may increase m_{v_i}, which is bounded by $|m_g| - 1$ (m_g is used to rewrite one variable of m_{v_i}, which is actually impossible).

During the unification, $G_{\mathcal{P}} \leq |m_v| + |m_g| \times 2 + 1 < |\mathcal{P}| \times 2$. As discussed, the number of variables in $G_{\mathcal{P}}$ is no more than $|m_v|$, so VARIABLE at most can be applied for $|m_v|$ steps, which is less than $|\mathcal{P}|$. After that, any transformation will either terminate LNMU or strictly decrease $G_{\mathcal{P}}$. Since $G_{\mathcal{P}} < |\mathcal{P}| \times 2$, we have: $\forall \mathcal{P}$, after $|m_v| + |\mathcal{P}| \times 2 + 1$ steps, LNMU terminates. Considering $|m_v| < |\mathcal{P}|$, LNMU is linear (exploring one branch).

To guarantee that LNMU can find one or all unifiers of $\mathcal{P}$, we may need to traverse multiple branches of $\mathcal{T}(\text{LNMU}(\mathcal{P}))$, which may take exponential time. We define a *well-formed* labelled multiset unification problem $\mathcal{P}_w$ as: multisets that, during the unification in one equation or inclusion relation of $G_{\mathcal{P}_w}$, each contain, at most, one variable that is not deterministically unifiable by another equation or inclusion relation of the same goal. $\mathcal{P}_w$ can be solved in linear time by carefully selecting transformations where only one of the successful branches of $\mathcal{T}(\text{LNMU}(\mathcal{P}_w))$ is traversed.

$\mathcal{P}_w = \{a(XY)b(X) = a(11)b(1)\}$ is well-formed. By applying FUNCTOR twice, we can get two equations, which are $X = 1$ and $XY = 11$. Both of them

only contain one variable that is not unifiable by other equations. By applying VARIABLE, GROUND and DELETE-1, we can easily get a unifier $\vartheta = \{X \mapsto 1, Y \mapsto 1\}$.

$\mathcal{P} = \{a(XY)b(AB) = a(11)b(cd)\}$ is NOT well-formed. By applying FUNCTOR twice we can get two equations: $XY = 11$ and $AB = cd$. Both of them contain two variables that are not unifiable by other equations. All of their variables can bind with several different g-multisets – in this example, 12 unifiers can finally be found.

We implemented LNMU in our cP system simulator cPSim[1], and tested several labelled multiset unification problems in a Windows PC with an Intel Core i5-8500 processor and 16 GB memory (Table 3.2). The implementation traverses all branches of $\mathcal{T}(\text{LNMU}(\mathcal{P}))$ and can find all the unifiers. In the experiment, each of the well-formed labelled multiset unification problems only have one unifier. For NOT well-formed multiset unification problems, the unification time and number of unifiers drastically increases with their problem sizes or number of variables.

Table 3.2: A comparison between well-formed and NOT well-formed labelled multiset unification problems

Well-formedness	Equation to solve	Number of unifiers	Running time
✓	$f(X)g(Y) \subset a1f(b)g(c)$	1	1ms
✓	$a(XYc(Z))\ b(Xd(ZW)) \subset$ $a(ghc(h))a(gjc(j))\ b(fd(jk))b(gd(jk))$	1	6ms
✗	$a(XY)b(AB) = a(11)b(cd)$	12	7ms
✗	$a(XYZ)b(ABC) = a(111)b(cdef)$	810	605ms
✗	$a(WXYZ)b(ABCD) = a(1^{11})b(1^{7})$	43680	12447ms

3.5 Worked examples

A simple example of labelled multiset unification is to unify $aXg(Y) = a1g(c)$. By applying GROUND, we can delete the shared atom a from both the equations' lhs and rhs, and transform the equation into $Xg(Y) = 1g(c)$. Then we can apply FUNCTOR, to get two equations: $X = 1$ and $Y = c$. For $X = 1$, by applying VARIABLE, a binding $\{X \mapsto 1\}$ will be added to $G_{\mathcal{P}}$, and $X = 1$ will be transformed into $\lambda = \lambda$, which can be deleted by DELETE-1. For $Y = c$, by applying VARIABLE, $\{Y \mapsto c\}$ will be added to $G_{\mathcal{P}}$, and $Y = c$ will be transformed into $\lambda = \lambda$, which can be deleted by DELETE-1, too. After solving all the

[1] `https://github.com/YezhouLiu/cPSim-version1`

equations in $G_{\mathcal{P}}$, $G_{\mathcal{P}} \cap \mathcal{E} = \varnothing$. Then $G_{\mathcal{P}}$ can apply SUCCESS and output a unifier $\vartheta = G_{\mathcal{P}} = \{X \mapsto 1, Y \mapsto c\}$. The example is shown in Table 3.3.

Table 3.3: Solving the labelled multiset unification problem: $aXg(Y) = a1g(c)$

$G_{\mathcal{P}}$	**Transformation to apply**	**Step**
$aXg(Y) = a1g(c)$	GROUND	0
$Xg(Y) = 1g(c)$	FUNCTOR	1
$X = 1, Y = c$	VARIABLE	2
$\lambda = \lambda, Y = c, X \mapsto 1$	DELETE-1	3
$Y = c, X \mapsto 1$	VARIABLE	4
$\lambda = \lambda, X \mapsto 1, Y \mapsto c$	DELETE-1	5
$X \mapsto 1, Y \mapsto c$	SUCCESS	6
$\square$ unifier: $\vartheta = \{X \mapsto 1, Y \mapsto c\}$		

As mentioned, choosing transformations in a different order may bring a different result. For $aXg(Y) = a1g(c)$, if LNMU first applies VARIABLE and nondeterministically finds a binding $\{X \mapsto a\}$, the equation will be transformed into $ag(Y) = 1g(c)$, which is unsolvable.

Table 3.4: Solving the labelled multiset unification problem: $a(XYc(Z))\ b(Xd(ZW)) \subset a(ghc(h))\ a(gjc(j))\ b(fd(jk))\ b(gd(jk))$

$G_{\mathcal{P}}$	**Transformation to apply**	**Step**
$a(XYc(Z))\ b(Xd(ZW)) \subset$ $a(ghc(h))\ a(gjc(j))\ b(fd(jk))\ b(gd(jk))$	FUNCTOR	0
$b(Xd(ZW)) \subset a(ghc(h))\ b(fd(jk))\ b(gd(jk))$, $XYc(Z) = gjc(j)$	FUNCTOR	1
$\lambda \subset\ a(ghc(h))\ b(fd(jk))$, $Xd(ZW) = gd(jk)$, $XYc(Z) = gjc(j)$	DELETE-2	2
$XYc(Z) = gjc(j)$, $Xd(ZW) = gd(jk)$	FUNCTOR	3
$Xd(ZW) = gd(jk)$, $XY = gj$, $Z = j$	VARIABLE	4
$Xd(jW) = gd(jk)$, $XY = gj$, $Z \mapsto j$, $\lambda = \lambda$	DELETE-1	5
$Xd(jW) = gd(jk)$, $XY = gj$, $Z \mapsto j$	FUNCTOR	6
$X = g$, $jW = jk$, $XY = gj$, $Z \mapsto j$	VARIABLE	7
$X \mapsto g$, $\lambda = \lambda$, $jW = jk$, $gY = gj$, $Z \mapsto j$	DELETE-1	8
$X \mapsto g$, $jW = jk$, $gY = gj$, $Z \mapsto j$	GROUND	9
$X \mapsto g$, $jW = jk$, $Y = j$, $Z \mapsto j$	GROUND	10
$X \mapsto g$, $W = k$, $Y = j$, $Z \mapsto j$	VARIABLE	11
$X \mapsto g$, $W = k$, $Y \mapsto j$, $\lambda = \lambda$, $Z \mapsto j$	DELETE-1	12
$X \mapsto g$, $W = k$, $Y \mapsto j$, $Z \mapsto j$	VARIABLE	13
$X \mapsto g$, $W \mapsto k$, $\lambda = \lambda$, $Y \mapsto j$, $Z \mapsto j$	DELETE-1	14
$X \mapsto g$, $W \mapsto k$, $Y \mapsto j$, $Z \mapsto j$	SUCCESS	15
$\square$ unifier: $\vartheta = \{W \mapsto k, X \mapsto g, Y \mapsto j, Z \mapsto j\}$		

Another example is to unify $a(XYc(Z))\ b(Xd(ZW)) \subset a(ghc(h))\ a(gjc(j))\ b(fd(jk))\ b(gd(jk))$, which is shown in Table 3.4. According to the strategy of selecting transformations, described in Section 3.3, we first apply FUNCTOR, match $a(XYc(Z))$ with $a(gjc(j))$, and transform G into $b(Xd(ZW)) \subset a(ghc(h))\ b(fd(jk))\ b(gd(jk))$ and $XYc(Z) = gjc(j)$. By applying FUNCTOR, we can eliminate another fuctor b, match $b(Xd(ZW))$ with $b(gd(jk))$, and transform G into $\lambda \subset\ a(ghc(h))\ b(fd(jk))$, $Xd(ZW) = gd(jk)$, and $XYc(Z) = gjc(j)$, where $\lambda \subset\ a(ghc(h))\ b(fd(jk))$ can be deleted by DELETE-2. Having a simpler G which only contains $XYc(Z) = gjc(j)$ and $Xd(ZW) = gd(jk)$, we can continue to apply FUNCTOR to eliminate the functions c and d, and then apply other transformations. After 15 steps, LNMU can find an unifier $\vartheta = \{W \mapsto k, X \mapsto g, Y \mapsto j, Z \mapsto j\}$ to the problem.

3.6 Summary

In P system variants such as cP systems, labelled multisets are widely used to represent membrane structures with objects. By supporting compound terms and generic rules, cP systems have great representational power (compared to other P system variants). However, the unification of labelled multisets is a challenge. Previous studies discussed several first-order unification algorithms, but none of them can be readily applied to labelled multisets.

In this chapter, we formally defined the first-order one-way unification problem for labelled multisets, and proposed a corresponding algorithm named LNMU. Multiple examples were used to illustrate the application of LNMU, and a software implementation was provided.

LNMU is proven to be terminating, and can find all the unifiers to a labelled multiset unification problem. We further showed that LNMU can solve well-formed labelled multiset unification problems in linear time.

LNMU can help researchers to implement, simulate, and verify cP systems in real life. In Chapter 6, we will introduce a cP-specific simulation and formal verification framework with LNMU fully implemented. Before that, we will first discuss how to use existing general purpose model checkers to model and verify cP systems in the next chapter.

Chapter 4

Model Checking of cP systems

Model checking is a formal method for verifying whether a system meets certain specifications, it explores all possible system states in a brute-force manner. Different modelling and (property) specification languages are supported by different model checking tools (model checkers). Usually, a modelling language is often used to describe how a system behaves, and a specification language is used to specify what the system should do.

To verify a cP system Π using a model checker which supports a modelling language $\mathcal{L}_m$ and a specification langauge $\mathcal{L}_s$, the following general process can be considered:

- Modelling: model Π using $\mathcal{L}_m$, and formalise the property to be checked using $\mathcal{L}_s$.
- Running the model checker: check the validity of the property in the system model.
- Analyse the results: if a property is satisfied, we can move to check the next property. Otherwise, we can analyse the counterexample generated by the model checker, and refine the cP system design.

In this chapter, we will introduce two cP systems, namely Π_{SSP} and Π_{Sudoku}. These can efficiently solve the subset sum problem (SSP) and Sudoku respectively. We will verify the system properties of Π_{SSP} and Π_{Sudoku} using two general purpose model checkers PAT3 and ProB [8, 4, 6].

We propose two mapping guidelines to transform cP systems into verification problems in CSP# and B. As an example, we illustrate how to apply the mapping guidelines to Π_{SSP}, and transform it into a CSP# model and a B machine.

This chapter is organised in the following way. Section 4.1 introduces the two cP systems which solve SSP and Sudoku respectively. Section 4.2 and 4.3 discuss how to translate cP systems to CSP# and B. Section 4.4 shows the verification results of the two cP systems. Section 4.5 summarises the chapter.

4.1 cP system solutions to SSP and Sudoku

A problem is NPC if and only if it is both in NP and NP-hard, where NP is an abbreviation for "nondeterministic polynomial time". For a NP problem, each of its inputs should be associated with a set of solutions of polynomial length, which can be validated in polynomial time. A problem $\mathcal{P}$ is said to be NP-hard, if everything in NP can be transformed in polynomial time into $\mathcal{P}$, even though $\mathcal{P}$ may not be in NP. The class NPC includes the hardest problems in NP, both SSP and Sudoku are NPC problems.

We propose two cP systems Π_{SSP} and Π_{Sudoku}, where Π_{SSP} solves SSP in linear time, and Π_{Sudoku} solves Sudoku in sublinear time. Both of Π_{SSP} and Π_{Sudoku} work under the hypothesis of unlimited processors and memory, which do not break the Church–Turing barrier.

4.1.1 A cP system that solves SSP in linear time

The subset sum problem is defined as follow.

$INSTANCE$: a set $S = \{i_1, i_2, ..., i_n\}$, where $i_x \in Z^+, x \in [1, n]$ and a target integer T. $QUESTION$: is there a subset $A \subseteq S$ such that $\sum_{x \in A} x = T$?

Our cP systems Π_{SSP} has a ruleset with five rules (Fig. 4.1) that describes a layer-by-layer searching algorithms. If there exists a subset A that satisfies $\sum_{x \in A} x = T$, the system will halt at s_2 and output the elements in A. Otherwise, after all the subsets of S have been generated and checked, the system will halt at s_3 and output a term $o(\lambda)$.

In the ruleset, $m(\_)$ denotes the original set S; $t(\_)$ denotes the target integer T; $o(\_)$ is the final output of the system; and $p(\_)$ refers to a "path", which stores the used or visited elements $u(\_)$, unused or unvisited elements $n(\_)$, and the sum of used elements $s(\_)$.

s_0	$\rightarrow_1$	s_1	$p(u(\lambda)\ n(M)\ s(\lambda)) \mid m(M)$	$(R1)$
s_1	$\rightarrow_1$	s_2	$o(X) \mid p(u(X)\ \_\ s(T))\ t(T)$	$(R2)$
s_1	$\rightarrow_1$	s_3	$o(\lambda) \mid p(\_\ n(\lambda))$	$(R3)$
s_1	$\rightarrow_+$	s_1	$p(u(Xm(Y))\ n(Z)\ s(SY)) \mid p(u(X)\ n(m(Y)Z)\ s(S))$	$(R4)$
$s_1\ p(\_)$	$\rightarrow_+$	s_1		$(R5)$

Figure 4.1: The ruleset of Π_{SSP}

The initial state of the system is s_0. By applying *R1* once, the system creates a path term for the original set S, which contains $u(\lambda)$, $n(M)$ and $s(\lambda)$. Thus, all elements in S are marked as unused, and the sum of the path is 0. After applying *R1*, the state of the system will be changed to s_1, then the rest of rules will be activated.

R2 describes an expected termination of the cP system. If the cP system can find a path (subset), whose sum of elements equals T, the system will change its state to s_2 and output the elements in the subset.

R3 describes another termination. After checking all the subsets of S, if the cP system cannot find any subset A satisfying $\sum_{x\in A} x = T$, the system will apply *R3*, output $o(\lambda)$, and halts at s_3.

Rules *R4* and *R5* commit to the same state s_1, which can be applied in the same step. By applying *R4*, the cP system creates new path terms by moving one element from $u(\_)$ to $n(\_)$ and recomputing the sum $s(\_)$. Since *R4* works in the max-parallel model, all the combinations of elements in S will be considered. The path generated by *R4* will be sent to the virtual product membrane, which will be activated in the next step. In the same step, after *R4* is applied, *R5* will clean all path terms in the system except newly generated ones which are still in the virtual product membrane.

Π_{SSP} is a linear solution to SSP. The worst number of running steps is $n+2$, when $\sum_{x\in S} x = T$ or $\nexists A, A \subseteq S, \sum_{x\in A} x = T$.

4.1.2 A cP system that solves Sudoku in sublinear time

Sudoku is a number-placement puzzle designed for a single player, which has $m\times m$ cells divided into m blocks. A solvable Sudoku puzzle may have one or multiple solutions. In a valid Sudoku solution, each row, column and block contains exactly one of each number from 1 to m. For all the classic 9×9 Sudoku puzzles, there are approximately 6.67×10^{21} valid solutions [104].

Our cP system Π_{Sudoku} solves Sudoku in sublinear time. The main strategy of Π_{Sudoku} is to generate all possible solutions (matrices), eliminate invalid ones, and filter them by comparing them to the input numbers. For an $m \times m$ Sudoku, the cP system will first generate all valid m-size row candidates, where each candidate is an arrangement of $[1..m]$. Then the system will use the row candidates to build templates of $m \times m$ matrices. After getting all the matrix templates, the system will filter them by columns and blocks. After this, it will contain all the valid $m \times m$ Sudoku solutions. Then the system can match the matrix templates to a puzzle instance with certain input numbers, and find its solutions.

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

A simple Sudoku puzzle ($m = 4$) is shown in Fig. 4.2. We use two terms $m(4)$ and $n(2)$ to represent its problem size and block size. Four numbers $a(1)$, $a(2)$, $a(3)$ and $a(4)$ can be used to solve the puzzle. The puzzle instance is encoded as $s(r(1)(c(3)(2), c(4)(4)),\ r(2)(c(1)(2), c(2)(4), c(4)(3)),\ r(3)(c(2)(1)),$ $r(4)(c(3)(3)))$. In the encoding, the term s stores all the existing numbers of the puzzle, here the subterm labels r and c refer to "row" and "column" respectively. The subterm $r(2)(c(1)(2), c(2)(4), c(4)(3))$ can be interpreted as "the value in row 2 column 1 is 2, in row 2 column 2 it is 4, and in row 2 column 4 it is 3".

		2	4
2	4		3
	1		
		3	

Figure 4.2: A Sudoku puzzle, $m = 4$

4.1.2.1 Generating row candidates

Π_{Sudoku} first generates all the row candidates. Each row candidate contains all the numbers from 1 to m, and each number only appears once. A ruleset with

four rules can be used to generate row candidates in a column by column manner (Fig. 4.3).

$s_1\ l(M1)$	$\rightarrow_1$	s_2	$l(1) \mid m(M)$	$(R1)$
s_1	$\rightarrow_+$	s_1	$p(X, c(L)(V)) \mid l(L), a(V), p(X), (c(\_)(V) \not\subseteq X)$	$(R2)$
$s_1\ p(\_)$	$\rightarrow_+$	s_1		$(R3)$
$s_1\ l(L)$	$\rightarrow_1$	s_1	$l(L1)$	$(R4)$

Figure 4.3: Ruleset (1) of Π_{Sudoku}: generating row candidates

The rule *R1* uses a counter $l(\_)$ to track the progress of generating row candidates. If the cP system is at s_1 and the value of $l(M1)$ is greater than the puzzle size $m(M)$, *R1* is applicable, which means all the row candidates have been successfully generated. By applying *R1*, the system resets the counter to $l(1)$, changes its state to s_2 and moves to the next ruleset. *R1* is the first rule in Ruleset (1), thus it will always be tested before *R2*, *R3*, and *R4*.

R2 works in the max-parallel model, thus all the compatible unifiers will be applied. *R2* adds a number V at column L to each row candidate $p(X)$. The relation $c(\_)(V) \not\subseteq X$ guarantees the number V has not been used in the same row candidate. At the beginning the computation, $p(\_)$ was empty. By applying the ruleset once, *R2* will create m different terms, which are $p(c(1)(1))$, $p(c(1)(2))$..., $p(c(1)(m))$. By applying the ruleset again, *R2* will generate $m \times (m-1)$ terms including $p(c(1)(1), c(2)(2))$, $p(c(1)(1), c(2)(3))$..., $p(c(1)(m), c(2)(m-1))$. After applying the ruleset m times, the cP system will eventually generate all the $m!$ row candidates, which contains all the arrangements of $[1..m]$.

R3 is another max-parallel rule, which cleans the out-of-date $p(\_)$ terms in the system. The rules *R2*, *R3*, and *R4* commit to the same target state, which will be applied in the same step. Thus, $p(\_)$ terms newly generated by *R2* will not be immediately consumed by *R3*.

R4 increases the counter $l(\_)$ by one in each step. The old counter $l(L)$ will be consumed by *R4*, and a new counter $l(L1)$ will be produced.

To generate all the row candidates for $m \times m$ Sudoku, the ruleset needs to be applied $m+1$ times, then the system state will be changed from s_1 to s_2.

4.1.2.2 Generating matrix templates

The ruleset to build matrix templates is shown in Fig. 4.4. Using the previous generated row candidates, Π_{Sudoku} builds matrix templates row by row.

$s_2\ l(M1)$	$\rightarrow_1$	s_3	$l(1) \mid m(M)$	$(R5)$
s_2	$\rightarrow_+$	s_2	$t(X, r(L)(P)) \mid l(L), p(P), t(X), (r(\_)(P) \notin X)$	$(R6)$
$s_2\ t(\_)$	$\rightarrow_+$	s_2		$(R7)$
$s_2\ l(L)$	$\rightarrow_1$	s_2	$l(L1)$	$(R8)$

Figure 4.4: Ruleset (2) of Π_{Sudoku}: generating matrix templates

The counter $l(\_)$ is used to track the working row. *R5* will be applicable once all the m rows of matrix templates are filled with row candidates – when $l(M1)$ is greater than $m(M)$. *R5* changes the system state to s_3, and resets the counter to $l(1)$.

R6 generates matrix templates row by row. In each step, the system adds exactly one row candidate $p(P)$ at row L to each matrix template $t(X)$. After m steps, the system will finish generating all $m!!/(m! - m)!$ matrix templates. *R7* cleans out-of-date $t(\_)$ terms, and *R8* increases the counter $l(\_)$ by one in each step.

Ruleset (2) takes $m + 1$ steps in total. The matrix templates generated by ruleset (2) do not contain any number conflicts in each row since every row candidate is an arrangement of $[1..m]$. However, they may contain number conflicts in columns and blocks (Fig. 4.5).

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

Figure 4.5: Number conflicts in a matrix template

4.1.2.3 Filtering matrix templates by columns

To delete the matrix templates with number conflicts in columns, Π_{Sudoku} needs one max-parallel rule (Fig. 4.6). The rule works as a filter, which is applied to all the matrix templates simultaneously. In a matrix template $t(\_)$, if there are two cells in the same column – row A column C and row B column C – share the same value V, the template will be consumed (deleted). *R9* only needs one step to run. After applying it, all the matrix templates left in the system do not have any number conflicts in rows and columns.

$s_3 \quad t(r(A)(c(C)(V)\_), r(B)(c(C)(V)\_), \_) \quad \rightarrow_+ \quad s_3 \quad (R9)$

Figure 4.6: Ruleset (3) of Π_{Sudoku}: filtering matrix templates by columns

4.1.2.4 Filtering matrix templates by blocks

To check if matrix templates have number conflicts in blocks, we need to create some supporting terms to indicate the relationship among rows, columns and blocks. For example, when $m = 9$, we can build terms $b(1)(1)$, $b(2)(1)$, $b(3)(1)$, $b(4)(2)$, $b(5)(2)$, $b(6)(2)$, $b(7)(3)$, $b(8)(3)$ and $b(9)(3)$ in the cP system. To check if two Sudoku cells are in the same block, we only need to compare their rows and columns with the supporting terms. Suppose we want to check if two cells – row 4 column 3 and row 6 column 1 – are in the same block (Fig. 4.7). First, we check terms $b(4)(A)$ and $b(6)(B)$ in the supporting terms, as a result we can find $A = 2$ and $B = 2$. Then we check $b(3)(X)$ and $b(1)(Y)$, and find that $X = 1$ and $Y = 1$. If $A = B$ and $X = Y$, the two cells are in the same block; otherwise they are not. In this example, row 4 column 3 and row 6 column 1 are in the same block.

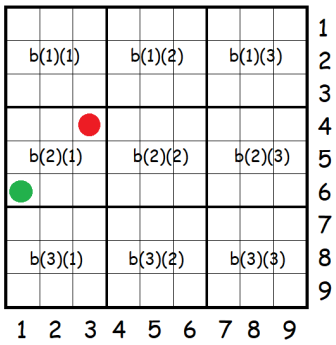

Figure 4.7: Checking if two cells are in the same block

The ruleset we use to build the supporting terms is shown in Fig. 4.8. *R10* creates two terms $v(1)$ and $k(N)$, and changes the state to s_4. Term $v(V)$ holds a value to fill in current supporting term $b(\_)(\_)$, and $k(K)$ tracks the boundary of blocks. *R11* monitors the progress of building supporting terms. When $l(M1)$ is greater than $m(M)$, the system has finished creating all the supporting terms, thus, it changes the state to s_5. *R12* creates a supporting term $b(L)(V)$ based on the counter $l(L)$ and value $v(V)$. *R13* updates terms $v(\_)$ and $k(\_)$ after the counter $l(\_)$ has moved to the next block. *R14* increases the counter $l(L)$ by one in each step.

$$
\begin{array}{llll}
s_3 & \to_1 & s_4 \;\; v(1),\; k(N) \mid n(N) & (R10)\\
s_4\; l(M1) & \to_1 & s_5 \;\; \mid m(M) & (R11)\\
s_4 & \to_1 & s_4 \;\; b(L)(V) \mid l(L),\; v(V) & (R12)\\
s_4\; k(K),\; v(V) & \to_1 & s_4 \;\; k(KN),\; v(V1) \mid n(N),\; l(K) & (R13)\\
s_4\; l(L) & \to_1 & s_4 \;\; l(L1) & (R14)
\end{array}
$$

Figure 4.8: Ruleset (4a) of Π_{Sudoku}: creating block checking supporting terms

In having the supporting terms in the system, *R15* can filter the matrix templates by blocks (Fig. 4.9). If the system detects a matrix template $t(\_)$ has two cells (row X column A and row Y column B) in the same block that share the same value V, it will consume the template. *R15* runs in the max-parallel model, which can filter all the matrix templates in 1 step.

$$
\begin{array}{llll}
s_5 \;\; t(r(X)(c(A)(V)\_),\; r(Y)(c(B)(V)\_),\; \_) & \to_+ & & \\
& & s_5 \;\; \mid b(X)(W), b(Y)(W), b(A)(C), b(B)(C) & (R15)
\end{array}
$$

Figure 4.9: Ruleset (4b) of Π_{Sudoku}: filtering matrix templates by blocks

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

4.1.2.5 Matching matrix templates to a Sudoku instance

One max-parallel rule can be used to match the matrix templates to a Sudoku instance (Fig. 4.10). *R16* compares all matrix templates $t(\_)$ to the instance $s(S)$. If the system finds any conflicts between a matrix template $t(\_)$ and $s(S)$, it deletes the template. *R16* takes one step. After applying it, the $t(T)$ terms in the system are solutions to the instance. A Sudoku instance may have multiple valid solutions, regardless, the cP system is guaranteed to find all of them at the same step.

$$
s_5 \;\; t(r(R)(c(C)(V)\_)\_) \quad \to_+ \quad s_5 \;\; \mid s(r(R)(c(C)(U)\_)\_),\; U \neq V \qquad (R16)
$$

Figure 4.10: Ruleset (5) of Π_{Sudoku}: matching matrix templates to a Sudoku instance

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

4.2 Modelling cP systems in ProB and PAT3

As an extensible model checker, PAT3 supports multiple specification languages and a user friendly editing environment. Compared to several other model checkers including SPIN, UPPAAL, and PRISM, the performance of PAT3 is quite competitive [1]. PAT3 provides a modelling language named CSP# (Communicating Sequential Programs), which combines high-level modelling operators and programmer-favored low-level constructs.

Along with PAT3, we will also model and verify cP systems using ProB, which is an efficient constraint solver and model checker for the B-Method. B features such as non-deterministic operations, arbitrary quantification, sets, sequences, functions, and lambda abstractions are supported by ProB, which is particularly useful for modelling cP systems.

4.2.1 Transforming cP systems into CSP# models

PAT (Process Analysis Toolkit) is a general purpose model checker that was proposed in 2008 [40], which aims to analyse event-based compositional systems. The latest version of PAT is PAT3, which has four layers including modelling, abstraction, intermediate representation, and an analysis layer [37]. A modelling language named CSP# is supported by PAT3, which is an extended version of communicating sequential processes (CSP).

To transform cP systems into CSP# models, rules can be represented as processes, and multisets can be modelled as arrays of integers. States of cP systems can be modelled as global variables. Terms can be modelled as global or local variables, macros, or integer arrays. Promoters can be modelled as conditions. A mapping guideline is shown in Table 4.1.

Following the guideline, the rules of Π_{SSP} (Fig. 4.1) can be translated to CSP#. In the translation, we use arrays to model multisets in cP systems. The CSP# translation of *R1* is shown in Fig. 4.11, notably, a process named S0 is used to describe the rewriting of *R1*. At the beginning of the computation, the system checks if the cP state is s_0, then generates the rhs terms of *R1*. After applying

[1] `https://www.comp.nus.edu.sg/~pat/system/`

Table 4.1: A mapping guideline for transforming cP systems into CSP# models

cP System Component	**cP System Notation**	**CSP# Expression**	**Example**
ground term	$t(10)$	macro	#define t 10;
variable term	$a(X)$	variable	var a;
multiset, set	$a(1,1,2,3)$	array	var a = [1,1,2,3];
state	S_1	global variable	var state = 1;
promoter	$\mid x(X)\ y(X)$	condition	if(x = = y)
rewriting	$\rightarrow_1 x(YZ) \mid y(Y)\ z(Z)$	statement	x = y + z;

R1, all numbers from the original set m(M) are available, none of them are used, the sum of the subset is 0, and the cP state will be changed to s_1. Since arrays in CSP# must have a fixed size, an array [-1,-1,-1,-1] is used to represent λ (the problem size $n = 4$). Two supporting variables "p_u_last" and "p_n_size" are declared, these will be used to track the elements in the term $u(U)$ and $n(N)$.

```
#define N 4 // problem size
var m = [1,2,3,4]; // original set S: m(M)
var t = 10; // target number: t(T)
var out[N]; // system output
var state = 0;
var lambda = [−1,−1,−1,−1];
var p_u[N];
var p_n[N];
var p_s;
var p_u_last;
var p_n_size;
S0() = rule1 {
    if (state = = 0) {
        p_u = lambda; // u(λ)
        p_u_last = 0;
        p_n = m; // n(M)
        p_n_size = N;
        p_s = 0; // s(λ)
        state = 1;
    }
} −> S1_1();
```

Figure 4.11: The CSP# translation of *R1* in Π_{SSP}

In the translation of *R2* (Fig. 4.12), PAT3 checks if the sum of the subset (p_s) is equal to the target number t when the system's cP state is s_1. The CSP# grammar requires events in a process to be atomic, so we split the rule into two

processes, namely S1_1 and S2. Once the system finds that p_s is equal to t, it changes the cP state to s_2, outputs the elements in p_u and terminates.

```
S1_1() = rule2 {
    if(state = = 1 && p_s == t) {
        state = 2;
    }
} -> if(state = = 2) { S2() };
S2() = rule2 {
    out = p_u;
} -> Skip;
```

Figure 4.12: The CSP# translation of *R2* in Π_{SSP}

Similar to *R2*, we split *R3* to two processes, S1_1 and S3 (Fig. 4.13). The system checks the size of the set p_n which stores unused elements. If p_n is empty – which means all the possible subsets of the original set have already been checked – the system changes its cP state to s_3, outputs lambda and halts.

```
S1_1() = rule3 {
    if (state = = 1 && p_n_size = = 0) {
        state = 3;
    }
} -> if (state = = 3) { S3() };
S3() = rule3 {
    out = lambda;
} -> Skip;
```

Figure 4.13: The CSP# translation of *R3* in Π_{SSP}

The CSP# translation of *R4* in Π_{SSP} is shown in Fig. 4.14. The process S1_1 performs a state check, and S1_2 moves exactly one element from p_n to p_u to produce new subsets and recompute p_s. In the second line of the translation, the system creates processes using CSP# notation []i:0..(N-1)@ rule4..., which is a syntax sugar of P(0) [] P(1) [] ... P(N-1), where [] is the choice operator. The translation means either P(0) or P(1)... or P(N-1) may execute, which simulates the non-deterministical generation of path terms in the cP system.

R5 in Fig. 4.1 does not need to be explicitly translated into PAT3, which uses processes to manage the statespace. After all the events in a process have been processed, PAT3 will move to execute the next process. Each process contains its own copies of variables, thus, the terms contained by previous processes do not need to be manually deleted.

```
S1_1() = if (state = = 1) { S1_2() };
S1_2() = [] i:{0..(N−1)} @ rule4 {
    state = 1;
    if (p_n[i] != 0) {
        p_u[p_u_last] = p_n[i];
        p_n[i] = 0;
        p_s = p_s + p_u[p_u_last];
        p_u_last++;
        p_n_size−−;
    }
} −> S1_1();
```

Figure 4.14: The CSP# translation of *R4* in Π_{SSP}

4.2.2 Transforming cP systems into B machines

The ProB model checker supports several modelling languages including B, Event-B, and CSP [39]. B language has a rich set of built-in operations, which is particularly helpful for modelling cP systems. The mapping guideline for transforming cP systems into B machines is shown in Table 4.2.

Table 4.2: A mapping guideline for transforming cP systems into B machines

cP System Component	cP System Notation	B Expression	Example
ground term	$t(10)$	constant	CONSTANT t
variable term	$a(X)$	variable	VARIABLES a
multiset	$a(1,1,2,3)$	sequence variable	a := [1,1,2,3];
set	$a(1,2,3,4)$	set variable	a := {1,2,3,4};
state	S_1	integer variable	state := 1;
promoter	$\mid x(X)\ y(X)$	precondition	PRE x = y
rewriting	$\rightarrow_1 x(YZ) \mid y(Y)\ z(Z)$	statement	x := y + z;

In the mapping guideline, ground simple terms can be modelled as constants, and other simple terms can be modelled as variables in B. States can be modelled as variables of integers, and can be checked as preconditions. Promoters can be modelled as preconditions.

Multiset-based compound terms in cP systems such as $a(11bbc)$ can be modelled as sequences in B. Several sequence operations are pre-implemented in B, which include prepend element (E->s), append element (s<-E), reverse (rev(s)), first (first(s)), and last (last(s)).

Set-based compound terms such as $a(bcd)$ can be modelled as sets in B. Several set operations including cardinality (card(S)), Cartesian product (S*T), union (S\/T), intersection (S/\T), difference (S−T), element of (E:S) and subset of (S<:T) are provided by B.

Following the mapping guideline, the rules of Π_{SSP} (Fig. 4.1) can be translated to B. In the translation of *R1* (Fig. 4.15), the system state and the promoter m(M) are checked as preconditions. If the system's state is s_0 and m(M) exists, it copies all elements from m(M) to p_n, sets p_u as empty and p_s to zero, and changes the system state to s_1.

```
CONSTANTS
    m, t
PROPERTIES
    m = {1,2,3,4} & // original set S: m(M)
    t = 10 // target number: t(T)
VARIABLES
    state, p_s, p_u, p_n
    p_u: seq(m) &
    state: 0..3 &
    p_u: seq(m) &
    p_n: POW(m) &
    p_s >= 0
INITIALIZATION
    state, p_s, p_u, p_n := 0,0,[],{}
OPERATIONS
    rule1 =
        PRE state = 0 & card(m) > 0
        THEN p_n := m; // n(M)
        p_u := []; // u(λ)
        p_s := 0; // s(λ)
        state := 1
        END;
```

Figure 4.15: The B translation of *R1* in Π_{SSP}

```
out2 <-- rule2 =
    PRE state = 1 &
    p_s = t
    THEN out2 := p_u;
    state := 2
    END;
```

Figure 4.16: The B translation of *R2* in Π_{SSP}

The translation of *R2* is shown in Fig. 4.16. If the state check is passed, and the sum of a subset (p_s) is equal to the target number t, the system will change its state to s_2 and outputs the elements in p_u. Otherwise, *R2* is not applicable.

```
out3 <-- rule3 =
    PRE state = 1 &
    p_s /= t &
    card(p_n) = 0
    THEN out3 := [-1];
    state := 3
    END;
```

Figure 4.17: The B translation of *R3* in Π_{SSP}

The translation of *R3* describes another termination of the system (Fig. 4.17). When the system state is s_1, p_s is not equal to t, and the size of p_n is zero. This means the system has already checked all the subsets of m(M) and did not find any valid solutions to the problem, thus, the system will change its state to s_3 and output $o(\lambda)$. [-1] is used to represent $o(\lambda)$ in the B translation. Since no rule in Π_{SSP} starts at s_3, the system will halt after applying *R3*.

```
rule4(y) =
    PRE state = 1 &
    y:p_n // m(Y)
    THEN p_u := y --> p_u;
    p_n := p_n - {y};
    p_s := p_s + y;
    state := 1
    END;
```

Figure 4.18: The B translation of *R4* in Π_{SSP}

The B translation of *R4* describes the generation of the subsets (Fig. 4.18). In the B operation rule4(y), the parameter y is an element of p_n, which is added to p_u, and removed from p_n. The sum of the subset p_s will also be recomputed. All the possible values of y ($y \in p\_n$) will be considered that can be used to simulate the max-parallel application of *R4*.

Similar to the CSP# translation, the term consuming rule – *R5* in Fig. 4.1 – does not need to be translated. When we use ProB to verify Π_{SSP}, duplicate terms will not be generated.

4.3 Model checking results and disscussion

By modelling Π_{SSP} in both PAT3 and ProB[2], and Π_{Sudoku} in PAT3[3], we verified several system properties of the two cP systems. A desktop PC with i5-8400 CPU (2.80 GHz, 6 cores) and 8-GB memory was used to conduct the experiments.

4.3.1 Model checking results of Π_{SSP}

The PAT3 and ProB verification results of Π_{SSP} are shown in Table 4.3. Properties of Π_{SSP} including deadlockfreeness, termination, determinism, goal reachability, divergencefreeness, invariant violation, and other LTL properties were verified.

Table 4.3: Model checking results of Π_{SSP}

Problem instance	$n=4$ $S=\{1,2,3,4\}$ $T=10$				$n=7$ $S=\{1,2,4,55,56,57,119\}$ $T=295$				$n=10$ $S=\{1,2,4,55,56,57,119,235,244\}$ $T=777$			
Tool	PAT3		ProB		PAT3		ProB		PAT3		ProB	
Expected goal	✓				✗				✗			
Goal	✓	0.001s	✓	<0.5s	✗	0.419s	✗	4.333s	✗	572.4s	✗	3124s
Deadlockfreeness	✓	0.002s	✓		✓	0.413s	✓		✓	469.0s	✓	
Invariant violation	-		✗		-		✗		-		✗	
New errors			✗				✗				✗	
Termination	✓	0.002s	-		✓	0.002s	-		✓	0.003s	-	
Divergencefreeness	✓	0.003s			✓	0.676s			✓	891.9s		
Nondeterminism	✓	0.001s			✓	0.002s			✓	0.002s		
Reachability: S_2	✓	0.002s			✗	0.413			✗	493.2s		
Reachability: S_3	✗	0.001s			✓	0.001			✓	0.005s		

In cP systems, a deadlock is a system configuration that does not have any outgoing edge, which is also not an expected halting configuration. Badly designed rules may cause deadlocks. For example, if *R1* in Π_{SSP} is miswritten as: $s_0 \ \rightarrow_1 \ s_4 \ p(u(\lambda) \ n(M) \ s(\lambda)) \mid m(M)$, by applying it, the cP system will be stuck at s_4, since no rule in Π_{SSP}'s ruleset can be applied at s_4.

A process is divergent if it does not terminate or terminates in an exceptional state. Self-looping rules in cP systems can make the system diverge, which is often undesired. Most cP systems are non-deterministic: during the unification of the rules, multiple unifiers can be randomly selected. However, for certain cP systems, if each of their rule unifications have only one unifier, they are deterministic. Suppose we have a cP system Π_{NAT}, which starts at s_1 with a term $a(1)$ and has a rule R_{NAT}: $s_1 \ a(X) \ \rightarrow_1 \ s_1 \ a(X1)$. Since Π_{NAT} only has one system term during the computation, when R_{NAT} is applied only one unifier can be found, this

[2] https://github.com/YezhouLiu/cP-subset-sum
[3] https://github.com/YezhouLiu/cP-Sudoku

makes Π_{NAT} deterministic. Π_{NAT} can generate all the natural numbers, thus it is also non-terminating.

Invariant violation is a general property for ProB, where multiple variable constraints can be written as invariants. During the verification, ProB will keep tracking the constraints to make sure they are satisfied. Once a constraint is violated, ProB will raise a counterexample for the violation. "New errors" for ProB refers to errors and warnings that are generated by the Prolog error manager.

Since SSP is NP-complete, the model checking statespace grows significantly with the problem size. For instance, ProB's statespace contains 18743 states when $n = 7$, and 13492904 states when $n = 10$. Checking NPC solutions with large problem sizes is often time consuming. To check all the system properties that are shown in Fig. 4.1, when $n = 10$, the running time of PAT3 is around $2300s$ and with ProB it is more than $3000s$.

Deadlockfreeness and divergencefreeness checking is often slower than the checking of other properties, since more internal states and transitions need to be checked. For example, to guarantee a system is deadlockfree, a model checker often needs to check the entire statespace.

The system properties can be verified in PAT3 and ProB are slightly different, while both of them show that Π_{SSP} is deadlockfree, terminating, and can reach the expected goal. From the PAT3 verification results we can also find that Π_{SSP} is divergencefree, which means Π_{SSP} is both confluent and terminating.

Modelling cP systems in PAT3 and ProB have different advantages. In CSP#, fixed-size array is one of the most important data structures, which can be used to represent (non-nested) multisets in cP systems. For cP systems that solve set-based problems, it is often easier to model them in B language. By using set operations provided by ProB, the behaviour of cP systems can be described briefly.

There is no conflict between the verification results of PAT3 and ProB, however, the performance of PAT3 is generally better than ProB. For example, when $n = 10$, to verify properties such as goal reachability or deadlockfreeness, PAT3 is approximately five times faster than ProB. Thus, to verify a cP system that has a large statespace such as Π_{Sudoku}, PAT3 would be a better option compared to ProB.

4.3.2 Model checking results of Π_{Sudoku}

The statespace of Π_{Sudoku} is larger than factorial, thus it is practically impossible to verify the entire solution in any model checker. We verified the two core rulesets

(1) and (2) of Π_{Sudoku} in PAT3.

Table 4.4 shows the model checking results. As expected, the two rulesets in Π_{Sudoku} are deadlockfree, divergencefree, terminating, non-deterministic, and can reach the expected goal.

Table 4.4: Model checking results of ruleset (1) and (2) in Π_{Sudoku}

Ruleset	Problem size	Deadlockfreeness	Divergencefreeness	Terminating	Deterministic	Goal reachability
(1)	4	True	True	True	False	True
(1)	9	True	True	True	False	True
(2)	4	True	True	True	False	True

A major limitation of applying model checking to cP systems is state explosion, the verification of Π_{Sudoku} is a great example of this. To verify the two rulesets of Π_{Sudoku}, we chose two problem sizes which are $m = 4$ and $m = 9$. When $m = 9$, PAT3 encountered a memory explosion issue. To verify ruleset (1), PAT3 generated $9! = 362880$ row candidates and successfully checked the statespace. However, to verify ruleset (2), PAT3 needed to generate of the $9!!/(9! - 9)!$ matrix templates, which is impossible in practice. Even though PAT3 implements abstraction algorithms and can generate its statespace on the fly, the statespace is too intense to check.

Because of the combinatorial explosion and limited languages features supported by PAT3, max-parallel filtering rulesets, including ruleset (3), (4a), (4b), and (5) in Π_{Sudoku}, are not suitable to be verified via model checking. There is no straightforward way to manually release memory space of terms in model checkers, such as PAT3, to emulate term consumptions in cP systems.

4.4 Summary

In this chapter, we proposed two cP systems, Π_{SSP} and Π_{Sudoku}, which can solve SSP and Sudoku in linear and sublinear time, respectively. Making use of generic rules in cP systems, Π_{SSP} only has 5 rules, and Π_{Sudoku} only has 16 rules.

We used two model checkers – PAT3 and ProB – to verify Π_{SSP} and Π_{Sudoku}. Several system properties including deadlockfreeness, termination, determinism, goal reachability, divergencefreeness, and invariant violation were successfully verified.

To model cP systems in PAT3 and ProB, we proposed two mapping guidelines to transform cP systems into CSP# models and B machines. To automate the transformation process, following the mapping guidelines, we implemented a B-

translator and a CSP-translator for ground cP systems, which will be discussed in Chapter 6.

A major limitation of verifying cP systems using model checkers is combinatorial explosion. Theoretically, an arbitrary number of cP system rules can be applied in the same step, which can create exponential terms in one step. For certain cP systems, to exhaustively traverse their statespaces is impossible in practice.

Existing model checkers often only support low-level languages, which only have limited built-in data structures. To properly model cP systems into the model checkers requires human intervention. It is not guaranteed that all the cP systems can be completely modelled in existing model checkers. For example, modelling cP systems that contain multiple highly nested compound terms in PAT3 and ProB is non-trivial.

In the next chapter, we will discuss deductive verification, and introduce how to verify cP systems using the Coq proof assistant.

Chapter 5

Deductive Verification of cP systems

In addition to model checking, another approach of formal verification is deductive verification. Using proof assistants (interactive theorem provers) or automatic theorem provers (which include satisfiability modulo theories (SMT) solvers), we can model a cP system, specify its properties as a set of proof obligations, and prove the proof obligations manually or automatically.

In this chapter, to formally verify cP systems, we consider the Coq proof assistant[50], which is not an automated theorem prover but supports a set of automatic theorem proving tactics. Coq supports a specification language called Gallina. Code written in Gallina has a weak normalisation property, which has to be terminating.

To verify a cP system Π using Coq, the following process can be considered:

- Model Π using Gallina.
- Specifying the properties of cP systems as proof obligations.
- Prove the proof obligations using certain tactics.

We propose an open source library named cP-Coq, which describes cP system components and includes a number of basic theorems that support verification [9]. Two modelling guidelines are introduced, which can be used to transform cP notation into Gallina. cP systems including Π_{SSP} are used as examples to illustrate the approach.

The chapter is organised as follows. Section 5.1 introduces how to model cP systems in Gallina. Section 5.2 includes two case studies and shows how to prove

certain proof obligations. Section 5.3 discusses the pros and cons of the approach, and Section 5.4 concludes the chapter.

5.1 Modelling cP systems in Coq

cP systems are not only Turing complete, but also more efficient than many existing computing systems theoretically. Rules in cP systems can work in a non-deterministic max-parallel way, which is hard to fully simualte in Coq. Although non-terminating computations are not allowed in Gallina, we can still model several cP systems and verify their properties. In this study, we implemented a Coq library to assist the modelling of cP systems, namely cP-Coq[1].

5.1.1 Modelling cP system components

In cP-Coq, atoms are defined as lowercase letters and variables are uppercase letters. Functors – labels of terms (cells) – are defined as atoms. Ground terms are recursively defined as a labelled and nested multiset (of ground terms). A state is defined as a constructor s with a natural number.

```
Inductive atom := | a | b | c | d | e | f | g | h .
Inductive variable := | X | Y | Z | W | U | V.
Definition functor := atom.
Inductive g_term :=
    | Num (n1: nat)
    | Atom (a1: atom)
    | Term (label : functor) (b1: bag g_term).
Inductive state := s (n : nat).
```

A cP system computation consists of a sequence of transitions between different system configurations. A system is terminated when it reaches a configuration with no rule applicable. In cP-Coq, an inductive type called "cPsystem_conf" is defined to represent cP system configurations. Each cP system configuration has a state, and contains a multiset of ground terms (system terms).

```
Inductive cPsystem_conf := cP_sys (s1: state) (terms: g_multiset).
```

In order to facilitate the representation of cP terms with large numbers, we consider natural numbers to be ground terms. For example, $a(b(c)d(1^{1931}))$ can be

[1] https://github.com/YezhouLiu/cP-Coq

represented as: Term a [Term b [Atom c]; Term d [Num 1931]]. This design can also help us to inductively prove cP system properties related to natural numbers.

Polymorphic lists are used to represent multisets/bags. Several comparators and sorting functions are provided in cP-Coq, which can be used to keep lists of terms sorted. Thus, standard library functions such as *eq* can be directly used when necessary. Fig. 5.1 shows how to represent cP system components and configurations in Coq.

cP system component	**cP system representation**	**cP-Coq type**	**cP-Coq example**
atom	a	atom	a
variable	V	variable	V
functor	f	functor	f
simple term	a, X, or 1^3	atom, variable, nat	a, X, or 3
compound term	$a(a(1^2)b)$	Term, Num, Atom	Term a [Term a [Num 2]; Atom b]
g-multiset	$a\ b\ a$	bag g-term	[Atom a; Atom b; Atom a]
state	s_1	state	s 1
system configuration	a cP system at s_1 with no term	cPsystem_conf	cP_sys (s 1) nil.

Figure 5.1: Representing cP system components in Coq

5.1.2 Modelling cP system rules

Rules in cP systems are defined as types in cP-Coq. By applying a rule, a cP system will transit from one configuration to another.

```
Definition cP_rule : Type := cPsystem_conf -> cPsystem_conf.
Definition cP_ruleset : Type:= list cP_rule.
```

Since there is no straightforward way to implement the unification of labelled multiset based-terms in Gallina, human intervention is needed for representing generic rules. A set of system operations is provided by cP-Coq, which can be used to construct cP rules (Fig. 5.2).

A recommended way to represent a generic cP rule is by using a group of functions $f_1, f_2, ..., f_n$ to describe its rewriting logic, and using higher-order functions – such as map or filter – to apply $f_1, f_2, ..., f_n$ to all the system terms.

To simulate the behaviour of a cP system, we can either apply a rule or ruleset to a cP system configuration as one step, or we can keep applying a ruleset until the system terminates. In the code, a looping limit is introduced to guarantee the fixpoint function eventually terminates.

Rules	**cP system notation**	**Operations in cP-Coq**	**cP-Coq example**
producing terms	$\rightarrow ab$	ProduceTerms	ProduceTerms [a;b] sys
consuming terms	$ab \rightarrow$	ConsumeTerms	ConsumeTerms [a;b] sys
checking a promoter	$\mid p$	TermInSystemB	TermInSystemB p sys
changing system state	$s_1... \rightarrow s_2...$	ChangeState	ChangeState s2 sys
rewriting	$s_1... \rightarrow s_2 m_2$	NewConf	NewConf (s 2) m2

Figure 5.2: Modelling cP system rules in Coq

```
Fixpoint ApplyARuleset (sys: cPsystem_conf) (rs: cP_ruleset) : cPsystem_conf :=
    match rs with
    | h1 :: t1 => ApplyARuleset (h1 sys) t1
    | _ => sys
    end.

Fixpoint RunUntilTerminated (sys: cPsystem_conf) (rs: cP_ruleset) (limit1: nat) :
    cPsystem_conf :=
    match limit1, SystemIsTerminatedRSB sys rs with
    | O, _ => sys
    | _, true => sys
    | S n', false => RunUntilTerminated (ApplyARuleset sys rs) rs n'
    end.
```

A variety of strategies and tactics can be selected in Coq, the most common ones include mathematical induction and case analysis. A common way to prove a theorem is to recursively break its proof obligations (goals) into simpler subgoals, and then to prove each subgoal one by one. In addition to performing backward reasoning, tactics such as "apply" can conduct forward reasoning, which suits certain cases. As an interactive theorem prover, all the proofs need to be written manually in Coq.

As a supplementary approach, we implemented several functions in cP-Coq to perform model checking. System properties including deadlockfreeness, termination, loopingfreeness and rule validation can bechecked in cP-Coq.

5.2 Case studies

We modelled and verified several cP systems using cP-Coq. This include simple cP systems which perform minimum finding and gcd finding, and complex cP systems that solve NP-complete problems. In this section we will discuss the formal verification of two cP systems.

5.2.1 Verifying a minimum finding cP system using Coq

Suppose a cP system Π_{min} (at state s_1) contains n terms $a(X_1)$, $a(X_2)$, ..., $a(X_n)$. The following ruleset (Fig. 5.3) can find the minimum of X_1, X_2, ..., X_n in two max-parallel steps.

$$
\begin{array}{llll}
s_1 & \rightarrow_+ & s_2 \quad b(X) \mid a(X) & (R1) \\
s_2\ b(XY1) & \rightarrow_+ & s_3 \quad \mid a(X) & (R2)
\end{array}
$$

Figure 5.3: The ruleset of Π_{min}

R1 produces $b(X_k)$ for each $a(X_k)$, $k \in [1..n]$ and changes the system state from s_1 to s_2. *R2* consumes $b(X_i)$ if there exists $a(X_j)$, $X_j < X_i$, $i, j \in [1..N]$. By applying *R2*, the system state will be changed to s_3, then no rule is applicable, and the system will terminate.

The Coq representation of *R1* and *R2* is shown in Fig. 5.4. The rewriting logic of *R1* and *R2* is manually intepreted as functions including MakeB, R1, BIsNotGreaterThanA, and R2.

To represent *R1*, a function MakeB is defined, which can produce a term $b(X)$ from $a(X)$. By applying MakeB to all the system terms using the map function, R1 can model *R1* in a sequential way.

For *R2*, a function BIsNotGreaterThanA is implemented to represent the promoter (condition). By applying BIsNotGreaterThanA to all the system terms using the filter function, R2 can properly represent *R2*. After calling R2, if a b term is greater than any a term in the system, it will be consumed; otherwise it will be kept. A Coq simulation of the minimum finding cP system is shown in Fig. 5.5, where the term "b @num 3" in the halting configuration indicates that the min value of $\{3, 7, 6, 8\}$ is 3.

Using a pre-defined min function, we can specify the correctness property of the system, when it contains two initial terms $a(X_1)$ and $a(X_2)$ (Fig. 5.6).

```
Definition MakeB (t1: g_term) : g_term :=
    match t1 with
    | a @num x1 => b @num x1
    | _ => Atom e
    end.

Definition R1 (sys:cPsystem_conf) : cPsystem_conf :=
    match sys with
    | cP_sys (s 1) terms => NewConf (s 2) ((map MakeB terms) ++ terms)
    | _ => sys
    end.

Fixpoint BIsNotGreaterThanA (m1: g_multiset) (t1: g_term) : bool :=
    match t1, m1 with
    | b @num x1, a @num x2 :: t2 => if x1 <=? x2 then (BIsNotGreaterThanA t2 t1) else
        false
    | _, _ :: t3 => BIsNotGreaterThanA t3 t1
    | _, _ => true
    end.

Definition R2 (sys:cPsystem_conf) : cPsystem_conf :=
    match sys with
    | cP_sys (s 2) terms => NewConf (s 3) (filter (BIsNotGreaterThanA terms) terms)
    | _ => sys
    end.
```

Figure 5.4: The cP-Coq representation of *R1* and *R2*

```
Notation "f @num x" := (Term f [Num x]) (at level 50).
Definition cPsys1 := cP_sys (s 1) [a @num 3;a @num 7;a @num 6;a @num 8].
Definition cPsys2 := R1 cPsys1.
Definition cPsys3 := R2 cPsys2.
Compute cPsys1.
Compute cPsys2.
Compute cPsys3.

Output:
cP_sys (s 1) [a @num 3; a @num 7; a @num 6; a @num 8] : cPsystem_conf
cP_sys (s 2) [b @num 3; b @num 7; b @num 6; b @num 8; a @num 3; a @num 7; a @num 6; a
    @num 8] : cPsystem_conf
cP_sys (s 3) [b @num 3; a @num 3; a @num 7; a @num 6; a @num 8] : cPsystem_conf
```

Figure 5.5: A Coq simulation of Π_{min}

In Lemma SystemCorrectness, GetB is a function which can extract the resulting natural number from the halting configuration. In the proof, LETrivial1, LETrivial2, LETrivial3, and MinTrivial1 are simple lemmas that are predefined and proved in cP-Coq. In performing a case analysis on x_1 <=? x_2 and x_2 <=? x_1, we separated the proof obligation into several subgoals and proved

```
Lemma SystemCorrectness: forall (x1 x2: nat),
    GetB (R2 (R1 (cP_sys (s 1) [a @num x1; a @num x2])) = min x1 x2.
Proof.
    intros. destruct (x1 <=? x2) eqn: e1. try simpl.
    try rewrite LETrivial1; try rewrite LETrivial2;
    try rewrite e1. apply MinTrivial1 in e1. rewrite <- e1. reflexivity.
    destruct (x2 <=? x1) eqn: e2; try simpl.
    repeat(try rewrite LETrivial1; try rewrite LETrivial2;
    try rewrite e1; try rewrite e2). rewrite Nat.min_comm.
    apply MinTrivial1 in e2. rewrite <- e2. reflexivity.
    rewrite LETrivial3 in e1. discriminate e1.
    rewrite e2. reflexivity.
Qed.
```

Figure 5.6: A correctness proof of Π_{min} with two initial terms

them separately. Similar strategies can be applied to Π_{min} with more than two initial terms.

Π_{min} is expected to terminate in two steps (by applying *R1* and *R2* once) despite the number k of $a(X_k)$ terms. The specifications and proof of this system property is shown in Fig. 5.7, where functions such as SystemIsTerminatedRS and ApplyARuleset are pre-defined in cP-Coq.

```
Lemma SystemTerminatesInTwoSteps: forall (sys: cPsystem_conf),
    SystemIsTerminatedRS (R2 (R1 sys)) [R1; R2].
Proof.
    unfold SystemIsTerminatedRS. unfold ApplyARuleset. destruct sys. destruct s1.
    repeat (destruct n; try reflexivity; try discriminate).
Qed.
```

Figure 5.7: Proving Π_{min} terminates in two steps

Other system properties can be specified and proven as needed. For the cP system Π_{min}, both its correctness and complexity are proven for all the valid instances of Π_{min}.

5.2.2 Verifying Π_{SSP} using Coq

As mentioned in Chapter 4, Π_{SSP} is a cP system that solves SSP in linear time ($n + 2$ steps). Five rules are included in Π_{SSP}, which are shown in Fig. 5.8. In the ruleset, we modified certain term labels (compared to the ruleset shown in Fig. 4.1) to fit the functor definition in cP-Coq, where the algorithm remains the same.

s_0	$a(M)$	$\rightarrow_1$	s_1	$b(c()d(M)e())$	$(R1)$
s_1		$\rightarrow_1$	s_2	$g(X) \mid b(c(X)\_e(T))\ f(T)$	$(R2)$
s_1		$\rightarrow_1$	s_2	$g(\lambda) \mid b(\_d(\lambda))$	$(R3)$
s_1		$\rightarrow_+$	s_1	$b(c(Xa(Y))d(Z)e(SY)) \mid b(c(X)d(a(Y)Z)e(S))$	$(R4)$
s_0	$b(\_)$	$\rightarrow_+$	s_1		$(R5)$

Figure 5.8: The ruleset of Π_{SSP}

The system initially contains two terms: $a(a(i_1)a(i_2)...a(i_n))$ and $f(T)$. The first term $a(a(i_1)a(i_2)...a(i_n))$ represents the original multiset S, and $f(T)$ stores the target integer T. The initial state of the system is s_0, by applying *R1*, it consumes $a(a(i_1)a(i_2)...a(i_n))$, produces a new term $b(c()d(a(i_1)a(i_2)...a(i_n))e())$, and changes its state to s_1. In cP-Coq, *R1* can be directly modelled using the operation NewConf.

```
Definition R1 (sys: cPsystem_conf) : cPsystem_conf :=
    match sys with
    | cP_sys (s 0) [Term a x1; t1] => NewConf (s 1) [t1; Term b [Term c nil; Term d x1 ; e
          @num 0]]
    | _ => sys
    end.
```

Each occurrence of the term $b(c(X)d(Y)e(Z))$ in the cP system describes a subset of S. X includes elements of the subset, Y stores unused elements (compared to S), and Z stores the sum of the elements in X.

R2 and *R3* describe two terminating states of the system. If a target subset (whose sum of elements equals T) is found, the system generates a goal term $g(X)$, which contains that subset and changes its state to s_2. Otherwise, if all subsets have already been checked and none of them have a sum of elements which equals T, the system generates an empty goal term $g(\lambda)$ and changes its state to s_2.

```
Fixpoint MakeG2 (t1: g_term) (m1: g_multiset) : g_term :=
    match t1, m1 with
    | f @num v1, Term b [ Term c c1; _; e @num v2] :: t2 => if v1 =? v2 then Term g c1 else
          MakeG2 t1 t2
    | _, _ :: t2 => MakeG2 t1 t2
    | _, _ => Atom e
    end.
```

```
Definition R2 (sys: cPsystem_conf) : cPsystem_conf :=
    match sys with
    | cP_sys (s 1) (t1 :: terms) => match MakeG2 t1 terms with
        | Atom e => sys
        | x1 => NewConf (s 2) (x1 :: t1 :: terms)
        end
    | _ => sys
    end.
```

To represent *R2*, a recursive function named MakeG2 is implemented to generate the output $g(X)$ term. By comparing the sum of the elements in each subset of S with the target integer T, MakeG2 can output a solution to the problem. The function R2 applies MakeG2 to all the system terms, which simulates the rewriting logic of *R2*.

A similar design can be used to represent *R3*, which is shown as follows. In the code, the recursive function MakeG3 checks the set of unused elements $d(\_)$. If the length of the set equals zero, which means the set is empty, MakeG3 will generate a term $g(\lambda)$ as a system output. R3 applies MakeG3 to all the system terms, which represents the rewriting logic of *R3* .

```
Fixpoint MakeG3 (m1: g_multiset) : g_term :=
    match m1 with
    | Term b [ _; Term d d1; _] :: t1 => if (length d1) =? 0 then Term g nil else MakeG3 t1
    | _ :: t1 => MakeG3 t1
    | _ => Atom e
    end.

Definition R3 (sys: cPsystem_conf) : cPsystem_conf :=
    match sys with
    | cP_sys (s 1) terms => match MakeG3 terms with
        | Atom e => sys
        | x1 => NewConf (s 2) (x1 :: terms)
        end
    | _ => sys
    end.
```

R4 describes the core algorithm of the cP system. In each step, it moves exactly one element $a(Y)$ from $d(\_)$ to $b(\lambda)$, and recomputes the sum $e(\lambda)$ of the subset. Running *R4* in max-parallel model can generate all subsets of S in a layer-by-layer manner. Additionally, *R5* runs together with *R4*, which can clean out-of-date terms and save system memory.

Multiple supporting functions are implemented to represent *R4* and *R5*. The function MakeB is designed to move one element from $d(\_)$ to $b(\lambda)$. MakeBAll

is used to simulate the max-parallel generation of all the subsets, which applies MakeB to a multiset.

```
Fixpoint MakeB (t1: g_term) (m1: g_multiset) : g_multiset :=
    match t1, m1 with
    | _, nil => nil
    | Term b [Term c c1; Term d d1; e @num e1], (a @num x1) :: t2 => if TermInBagB (a
          @num x1) d1 then [Term b [Term c (c1 ++ [a @num x1]); Term d (RemoveATerm (a
           @num x1) d1); e @num (e1 + x1)]] ++ (MakeB t1 t2) else MakeB t1 t2
    | _, _ :: t2 => MakeB t1 t2
    end.
```

```
Fixpoint MakeBAll (m1: g_multiset) : g_multiset :=
    match m1 with
    | h1 :: t1 => (MakeB h1 (GetUnused h1)) ++ (MakeBAll t1)
    | _ => nil
    end.
```

Since *R4* and *R5* commits to the same target state, they will apply in the same step, which means they can be modelled in the same Gallina function. R4n5 describes the rewriting logic of *R4* and *R5*, which applies NotB and MakeBAll to all the system terms, where NotB is a function which checks if a term's label is not b.

```
Definition NotB (t1: g_term) : bool :=
    match t1 with
    | Term b _ => false
    | _ => true
    end.

Definition R4n5 (sys: cPsystem_conf) : cPsystem_conf :=
    match sys with
    | cP_sys (s 1) terms => NewConf (s 1) ((filter NotB terms) ++ (MakeBAll terms))
    | _ => sys
    end.
```

To prove the correctness of Π_{SSP} in Coq is non-trivial. The subset sum problem is NP-complete, in other words, we cannot guarantee solving it without checking all the subsets of S. Although cP systems are assumed to have unlimited computational resources, the real life implementations may run out of memory. In cP-Coq, we bound the problem when verifying certain system properties. By defining the system's validity, its correctness can be proven (the detailed proof can be found in cP-Coq's verification examples).

```
Lemma SystemCorrectness: forall (sys: cPsystem_conf), ValidSystem sys = true ->
    GetFValue (RunNSteps sys rs (psize+2)) = SetSum (GetG (RunNSteps sys rs (psize+2))).
```

Similarly, the system's complexity can be verified. By proving SystemTerminatesInNPlusTwoSteps, we can verify that the worst time complexity of the system is: psize (problem size) + 2 steps.

```
Lemma SystemTerminatesInNPlusTwoSteps: forall (sys: cPsystem_conf), ValidSystem sys =
     true ->
    SystemIsTerminatedRS (RunNSteps sys rs (psize+2)) rs.
```

When the problem size increases, the subset sum cP system may take a large amount of memory. We can write a lemma to specify and prove an upper bound of the memory usage.

```
Lemma SystemMemoryUse: forall (sys: cPsystem_conf) (n1: nat), (ValidSystem sys = true)
     /\ (0 <? n1 = true) ->
    SystemMemory (RunNSteps sys rs n1) <=? pow 2 (n1 + 1) = true.
```

Some system properties can be proven without bounding the problem. For example, we can prove that the system terminates at s_2.

```
Lemma SystemTerminatesAtS2: forall (sys: cPsystem_conf),
    SystemState sys = s 2 -> SystemIsTerminatedRS sys rs.
Proof.
    intros. destruct sys. destruct s1.
    repeat (try destruct n; try discriminate H; try reflexivity).
Qed.
```

Model checking can also be used to verify the system. For example, we can verify the properties of an instance of Π_{SSP} as follows.

```
Lemma SystemTerminated: SystemIsTerminatedRS cPsys6 rs.
Lemma SystemTerminatedAtS2: SystemState cPsys6 = s 2.
Lemma Loopingfreeness: LoopingCheckB cPsys1 rs 6 = false.
Lemma Deadlockfreeness: DeadlockCheckB cPsys1 rs 5 = false.
```

By grouping several problem instances together, we can verify their properties at once. Other lemmas and proofs of the system can be found in cP-Coq's example files.

5.3 Discussion

In using Coq we can either conduct theorem proving or model checking on cP systems. For certain system properties, we can prove them for all the problem instances using mathematical induction. For other properties, we can prove them by setting different upper bounds, or by conducting model checking instead.

Properties of cP systems can be verified in a divide-and-conquer manner. We can describe complex properties as multiple theorems or lemmas, prove them one by one, and then combine the results together. cP-Coq includes a number of basic theorems on cP systems' data structures and operations that can be used to construct proofs.

Compared to the model checking approach, which verifies system properties of certain cP system instances, in Coq we can verify certain properties for all the instances of a cP system. For example, we proved that the complexity of Π_{SSP} is linear in Coq, which is hard to verify by only checking a few instances of Π_{SSP}.

A comparison of verifying cP systems in Coq, PAT3 and ProB is shown in Table 5.1. A check mark in the figure indicates that certain properties can be verified by the corresponding tool. A cross mark indicates that some properties cannot be straightforwardly verified by the tool.

Table 5.1: A comparison of verifying cP systems in different formal tools

Properties	**Coq**	**PAT3**	**ProB**
Correctness	✓	✓	✓
Deadlockfreeness	✓	✓	✓
Invariant violation	✓	✗	✓
Terminating	✓	✓	✓
Nondeterministic	✗	✓	✗
Complexity	✓	✗	✗
Verification objects	All the instances	A small number of instances	

To effectively and efficiently simulate cP systems in modern computers is a long-term challenge. cP systems can solve NPC or even PSPACE-complete problems in linear time by trading memory for time. Thus, memory explosion is a common issue for all the cP system implementations in real life. By using cP-Coq, certain system properties can be verified by conducting theorem proving, which does not require the entire statespace of a cP system to be explored.

Modelling cP systems in Gallina requires human intervention, which is a major limitation of this work. Representing cP systems in different ways may significantly

affect the difficulty of proving theorems, and the representations themselves also need to be certified.

To completely simulate non-deterministic cP systems with max-parallel rules in the current version of Coq is extremely difficult. Fortunately, many well-designed cP systems are suggested to be confluent, so non-deterministic properties are often undesired for cP systems.

Model checking and interactive theorem proving are two complementary formal verification approaches, which can be combined together. Model checkers can be used to verify a cP system that has a relatively small statespace. Proof assistants can be used to mathematically prove certain properties of a cP system with a large or even unlimited statespace.

5.4 Summary

Deductive verification aims to verify a system's properties by conducting logical inference. In this chapter, we introduced how to formally verify cP systems using Coq proof assistant. To model cP systems in Gallina, we implemented a library named cP-Coq, and proposed two mapping guidelines to help transform cP systems into Gallina models. Compared to model checking, we can verify certain cP system properties for all of a system's instances without encountering memory explosion.

This study demonstrates the great potential of verifying membrane systems using interactive theorem provers. Following the mapping guidelines, we implemented a Gallina translator, which can be used to transform ground cP systems into cP-Coq models (Appendix A).

In the next chapter, we will introduce our software implementation of a cP-specific simulation and verification framework, which can simulate and verify cP systems in a natural way.

Chapter 6

cPV – a Formal Verification Framework for cP Systems

To formally verify cP systems, three major approaches are considered in this study. These include:

- Manually modelling and verifying cP systems using existing formal verifiers.
- Automately translating cP systems to modelling languages supported by existing formal verifiers, and then obtaining the verification results.
- Designing and implementing a domain-specific formal verifier for cP systems.

The first approach was discussed in Chapters 4 and 5. By using general purpose verifiers including PAT3, ProB, and Coq, system properties of several cP systems were successfully verified. In this chapter, we will discuss the other two approaches and introduce the software tools we built for each of them.

In order to fully automate the cP system formal verification process, we implemented the second approach with multiple translators, which can translate cP systems into certain modelling languages, and invoke existing formal verifiers in the backend to get verification results. To properly represent cP models in the tool, a corresponding domain-specific language (DSL) for cP systems was proposed. The implementation of translators follows the mapping guidelines that are proposed in Chapter 4.

In addition to using existing formal verifiers, we designed and implemented the third approach of a cP system-specific simulation and formal verification framework, namely cPV. cPV includes functionalities such as language parsing, system

simulation, verification algorithms, reduction techniques, counterexample generation, and different display options including a graphical user interface (GUI). Several important cP system properties including deadlockfreeness, confluence, termination, determinism, and goal reachability can be verified in cPV[1].

The chapter is structured as follows. Section 6.1 introduces the Python software tool, which can achieve fully automated cP system verification using existing formal verifiers. Section 6.2 presents the design and implementation of cPV. Section 6.3 concludes the chapter and discusses its contributions.

6.1 Automatically verifying ground cP systems using PAT3 and ProB

Manually verifying cP systems using existing formal tools requires in-depth understanding of formal verification techniques. This actually prevents many membrane computing experts from successfully applying formal verification to their cP models. To solve this issue, we implemented a software tool that is integrated with multiple existing formal verifiers that can automatically verify certain cP systems without human intervention.

6.1.1 cPVJ – a DSL for cP systems

In order to represent cP systems in computers, we defined a DSL named *cPVJSON* (*cPVJ* for short) by extending cP systems' syntax, this is fully supported by our implementations. Major differences between cPVJ and cP systems' syntax include:

- Delimiters in terms such as commas and whitespaces, which have frequently been accepted in previous cP studies, are not supported in cPVJ. For example, $f(g(a), b)$ and $f(g(a)\ b)$ in cPVJ must be written as f(g(a)b).
- cPVJ supports natural numbers rather than the symbol *1*. For example, $a(1)$, $b(11)$, and $c(1^7)$ need to be written as a(1), b(2), and c(7) in cPVJ.
- cPVJ does not support the use of superscripts or subscripts in terms. For example, the term $f(a^2b^3)$ needs to be written as f(aabbb).

[1]The implementations mentioned above can be found in the project `https://github.com/YezhouLiu/cP-Verifier`.

- States in cPVJ can be written as any string beginning with the letter "s'. Furthermore, as a convention, using "s" with a natural number to represent states is highly recommended, for example, "s0", "s1", or "s2".
- Underscores are not supported in cPVJ, variables can only be represented as uppercase letters. Using the variable "A" to represent anonymous variables is suggested.
- The symbol λ is not supported in cPVJ. However, a label with a pair of empty parentheses can be used to represent an empty labelled multiset, such as $f()$ or $g()$.
- In cPVJ, a optional "name" field is added for each cP system to describe its behaviours, this can be an arbitrary string.

A cP system in cPVJ is represented as a JSON object with four items (key-value pairs), these are "terms", "state", "ruleset" and "name".

The item "terms" stores a dictionary, which describes the system terms of a cP system and their multiplicities. Suppose a cP system has four system terms: a, $b(1^3)$, $b(1^3)$, and $f(g(1)k(1))$, these would be represented as "terms": {"a": 1, "b(3)": 2, "f(g(1)k(1))"}. Similar to cP systems' syntax, the order of terms and their subterms are irrelevant.

The item "ruleset" stores the rules of a cP system. A rule with the application model exactly-once is written as $lhs -> 1\ rhs$, and a max-parallel rule is written as $lhs -> +\ rhs$. Note that the application models are not written as subscripts and there is no whitespace between the arrow $(->)$ and "1" or "+". States, terms, arrows with application models ($-> 1$ or $-> +$), and the promoter symbol (|) in a rule need to be seperated by whitespaces, for example, s1 a(XY1) $->$ + s1 a(Y1) | a(X).

A cPVJ example is shown in Fig. 6.1, which describes a cP system that can find the greatest common divisor (GCD) of the two natural numbers 144 and 88. The cP system is named as "GCD cP system", and it starts at state s_1, contains two system terms $a(1^{512})$ and $a(1^{144})$, and has two rules: $s_1\ a(XY1) \rightarrow_+ s_1\ a(Y1) \mid a(X)$ and $s_1\ a(X)\ a(X) \rightarrow_1 s_2\ b(X)$. Note that the order of rules in a cP system matters – as mentioned, rules in cP systems are sequentially considered in the top–down order.

```
{"ruleset": ["s1 a(XY1) −>+ s1 a(Y1) | a(X)",
"s1 a(X) a(X) −>1 s2 b(X)"],
"terms": {"a(512)": 1, "a(144)": 1},
"state": "s1",
"name": "GCD cP system"}
```

Figure 6.1: A cPVJ example of a cP system which describes the Euclidean algorithm

6.1.2 An internal representation of cP systems

In having cP systems described in cPVJ, our tool can parse them into internal representations. Several object classes are implemented to represent and process cP systems, these include CPSystem, Term, and Rule (Fig. 6.2).

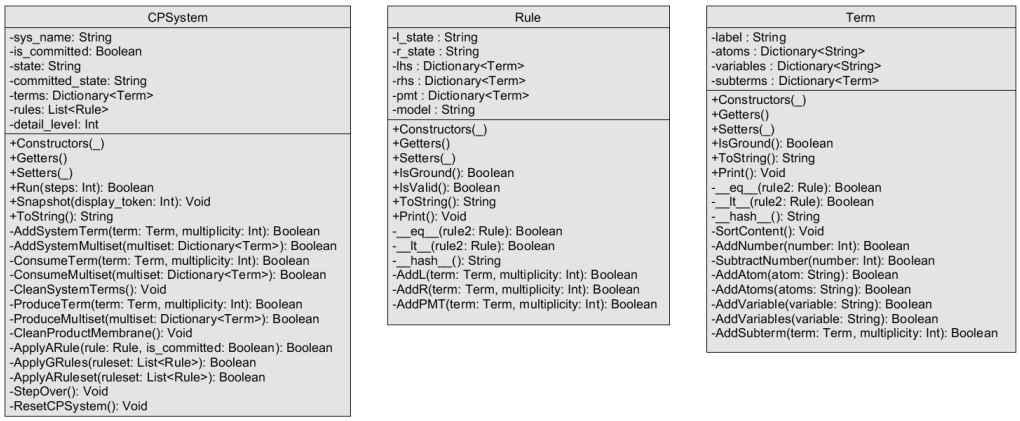

Figure 6.2: Object classes for internally modelling cP systems

In the implementation, atoms and variables in cP systems are defined as strings such as "a" and "X". The class *Term* is used to represent compound terms, this is recursively defined. Four member variables are defined in *Term*, these are *label*, *atoms*, *variables*, and *subterms*. Multisets of terms are defined as dictionaries of *Term* objects.

The class *CPSystem* is used to represent cP system configurations, this includes several member variables that describe the system's name, state, system terms, rules, and some other tokens such as *is_committed* and *detail_level*.

Rule objects are used to represent cP system rules, this includes six member variables including *l_state* (left state), *r_state* (right state), *lhs*, *rhs*, *pmt* (promoters), and *model* (application model).

For cP systems described in cPVJ, several parsers including *SystemParser*, *TermParser*, and *RuleParser* are implemented. These can parse a cPVJ object into a CPSystem object that contains a number of Term and Rule objects.

In addition to member variables, each class contains a rich set of methods, which are used to validate cP systems, simulate their behaviours, and perform formal verification.

6.1.3 B-translator

To automatically verify cP systems using ProB, following the transformation guideline shown in Table 4.2, we implemented a B-translator, which can translate certain cP systems from internal representations (mentioned in the above section) into B machines. B machines that are built by the B-translator consist of the following components:

- Machine headers to specify cP systems' names.
- Machine sections which include VARIABLES, INVARIANT, and INITIALISATION.
- Operations to represent cP system rules.

The B-translator works for all cP systems that contain ground rules (similar to classical P systems). The machine name can be obtained from the cP system's name. Terms of the cP system will be collected and translated into variables in B. For a term variable, the B-translator will also generate an invariant clause to guarantee its multiplicity is non-negative. After collecting all terms in the cP system, the B-translator will instantiate all the terms following their multiplicities.

For cP system rules, each rule is translated into an operation, with the name "r" plus an integer identifier. The state and lhs term check of the rules are written as preconditions (in PRE-clauses) of the operation. State change, term production, and term consumption are written in THEN clauses.

The core function of the B-translator (*cPtoB*) is to generate the string content of a B-machine(Fig. 6.3). Another function named *CreateBFile* is implemented to transform the string content into a B-machine with a ".mch" extension.

An example of a B-machine automatically generated by the B-translator is shown in Fig. 6.4. The B-machine represents a cP system (at s_1) named "simplecp", which contains 10 copies of term a and has two rules $s_1\ a^2\ \rightarrow_1\ s_1\ b$ and $s_1\ b^2\ \rightarrow_1\ s_1\ c\ d^2$.

```
def cPtoB(str_ruleset, system_terms, system_state, system_name):
  atoms = set()
  ruleset = []
  for str_rule in str_ruleset:
    rule = ParseRule(str_rule)
    ruleset.append(rule)
    for a1 in rule.LHS(): atoms.add(a1)
    for a2 in rule.RHS(): atoms.add(a2)
    for a3 in rule.PMT(): atoms.add(a3)
  B_file = 'MACHINE ' + system_name + '\nVARIABLES state'
  for ch in atoms: B_file += ',' + ch
  B_file += '\nINVARIANT state >= 0'
  for ch in atoms: B_file += ' & ' + ch + ' >= 0'
  B_file += '\nINITIALISATION state := ' + system_state[1:]
  for ch in atoms:
    if ch in system_terms: B_file += '; ' + ch + ' := ' + str(system_terms[ch])
    else: B_file += '; ' + ch + ' := 0'
  B_file += '\nOPERATIONS\n'
  i = 1
  for rule in ruleset:
    B_file += 'r' + str(i) + ' = PRE state = ' + rule.LState()[1:]
    LP = lnmu.MultisetUnion(rule.LHS(), rule.PMT())
    for atom in LP:
      B_file += ' & ' + atom + ' >= ' + str(LP[atom])
    B_file += ' THEN '
    for atom in rule.LHS():
      B_file += atom + ':=' + atom + '-' + str(rule.LHS()[atom]) + ';'
    for atom in rule.RHS():
      B_file += atom + ':=' + atom + '+' + str(rule.RHS()[atom]) + ';'
    B_file += 'state := ' + rule.RState()[1:] + ' END;\n'
    i += 1
  B_file = B_file[:-2]
  B_file += '\nEND'
  return B_file
```

Figure 6.3 The core function of the B-translator

```
MACHINE simplecp
VARIABLES state,d,b,a,c
INVARIANT state >= 0 & d >= 0 & b >= 0 & a >= 0 & c >= 0
INITIALISATION state := 1; d := 0; b := 0; a := 10; c := 0
OPERATIONS
r1 = PRE state = 1 & a >= 2 THEN a := a - 2; b := b + 1;
  state := 1 END;
r2 = PRE state = 1 & b >= 2 THEN b := b - 2; c := c + 1;
  d := d + 2; state := 1 END
END
```

Figure 6.4 An example of a B-machine generated by the B-translator

As discussed in Chapter 2, rules in cP systems are sequentially considered following a weak-priority order. If there exist multiple rules that commit to the same target state, they can be applied in the same step. However, in ProB, operations (cP system rules) will be considered independently. Thus, cP system properties related to number of steps cannot be properly verified via this approach.

Several verification functions such as *ProBMC*, *ProBMCCustom*, *ProBMCBreathFirst*, and *ProBMCTimeout* are implemented, these can be used to verify a cP system's properties in different manners using ProB.

6.1.4 CSP-translator

Similar to the B-translator, we implemented a CSP-translator following the transformation guideline shown in Table 4.1, which can translate ground cP systems to CSP#. As mentioned, CSP# is a language extension of CSP which supports shared variables, asynchronous communication channels and event associated programs.

CSP# code generated by the CSP-translator consists of global variables and processes. Here each global variable represents a ground term or state, and each process is used to model a rule. Names of cP systems will be written as comments.

A token named *applied* is used in the CSP# code to simulate the cP system rule applications. If a rule is succesfully applied, the system will commit to a new state, and all other rules which commit to different states will be disabled in the same step.

Ground terms in cP systems (global variables) will be declared with the *applied* token and system state, and will be initiated in a process named *P0*. The values of terms indicate their multiplicities.

A process generated by the CSP-translator will be named as "r" plus an integer identifier corresponding to a rule's index in the cP system. In the body of a process, both the state and lhs check will be placed in IF-clauses. Term consumption and production, state change, and token manipulation are written as statements. The core fuction of the CSP-translator is shown in Fig. 6.5, two extra processes *P_CHECK* and *P_NEXT* are implemented to loop rules in the cP system.

Fig. 6.6 shows an example CSP# file that is automatically generated by the CSP-translator, which also describes the cP system "simplecp". In the CSP# file, *P0* is the entry process, assertions such as *#assert P0() nonterminating* or *#assert P0() deadlockfree* can be used to verify the cP system's properties.

```
def cPtoCSP(str_ruleset, system_terms, system_state, system_name):
  atoms = set()
  ruleset = []
  for str_rule in str_ruleset:
    rule = ParseRule(str_rule)
    ruleset.append(rule)
    for a1 in rule.LHS():
      atoms.add(a1)
    for a2 in rule.RHS():
      atoms.add(a2)
  CSP_file = '//' + system_name + '\nvar applied = false; \n'
  for ch in atoms:
    CSP_file += 'var ' + ch + ';\n'
  CSP_file += 'var state = ' + system_state[1:] + ';\n\n'
  CSP_file += 'P0() = cp_init{\n'
  for ch in atoms:
    if ch in system_terms:
      CSP_file += ch + ' = ' + str(system_terms[ch]) + ';\n'
    else:
      CSP_file += ch + ' = 0;\n'
  CSP_file += '}-> P1();\n'
  i = 1
  for rule in ruleset:
    pn = str(i)
    CSP_file += 'P' + pn + '() = r' + pn + '{\n'
    CSP_file += 'if (state == ' + rule.LState()[1:]
    for atom in rule.LHS():
      CSP_file += ' && ' + atom + ' > 0'
    CSP_file += '){\napplied = true;\n'
    for atom in rule.LHS():
      CSP_file += atom + '=' +atom+ '-' +str(rule.LHS()[atom])+ ';\n'
    for atom in rule.RHS():
      CSP_file += atom + '=' +atom+ '+' +str(rule.RHS()[atom])+ ';\n'
    CSP_file += 'state = ' + rule.RState()[1:] + ';\n'
    if i < len(ruleset):
      CSP_file += '}\n}-> P' + str(i + 1) + '();\n'
    else:
      CSP_file += '}\n}-> P_CHECK();\n'
    i += 1
  CSP_file += 'P_CHECK() = if(applied == true){P_NEXT()}else{Skip};\n'
  CSP_file += 'P_NEXT() = {applied = false;}-> P1();\n'
  return CSP_file
```

Figure 6.5 The core function of the CSP-translator

Different from the code generated by the B-translator, in using the *applied* token, PAT3 can simulate cP systems in a desired way. Thus, cP system properties related to number of steps can be properly verified in PAT3.

Multiple verification functions such as *PAT3MC* and *PAT3MCCustom* are included in the implementation. These can be used to verify predefined system properties and custom LTL or CTL properties.

```
//simplecp
var applied = false;
var b;
var d;
var a;
var c;
var state = 1;

P0() = cp_init{
  b = 0;
  d = 0;
  a = 10;
  c = 0;
}-> P1();
P1() = r1{
  if (state == 1 && a > 0){
    applied = true;
    a = a - 2;
    b = b + 1;
    state = 1;
  }
}-> P2();

P2() = r2{
  if (state == 1 && b > 0){
    applied = true;
    b = b - 2;
    c = c + 1;
    d = d + 2;
    state = 1;
  }
}-> P_CHECK();

P_CHECK() = if(applied == true){P_NEXT()} else {Skip};

P_NEXT() = {applied = false;}-> P1();
```

Figure 6.6 An example CSP# file generated by the CSP-translator

6.1.5 Connecting to back-end verifiers

The model checkers ProB and PAT3 are integrated in our tool. Files generated by the B-translator or CSP-translator can be directly sent to ProB and PAT3, and the verification results can be obtained by running the two model checkers at the back-end.

To use ProB as a back-end verifier, a ProB client with version 1.10.x or higher needs to be installed and configured as an environment variable of the operating system. For each verification task, a subprocess will be created to run the executable command *probcli*. Suppose verification commands are collected and stored

in a list object *prob_commands*, the following line of code could then be used to obtain the ProB verification results of a cP system:

```
result = subprocess.run(prob_commands, stdout=subprocess.PIPE)
```

In the implementation, several frequently-checked properties including deadlockfreeness, invariant violations and new errors can be quickly verified using the *ProBMC* function. Custom properties can be checked via the *ProBMCCustom* function, this function collects custom commands, parameters, and other verification options from users and passes them onto ProB. The verification result will be formatted, displayed, and output as a file named "prob_verification_result.txt".

Suppose there a cP system (at s_1) named "simplecp2" that has 100 copies of term a, 90 copies of b, and two rules *R1*: $s_1\ a^3\ b\ \rightarrow_1\ s_1\ b$ and *R2*:$s_1\ b\ \rightarrow_1\ s_1\ a$. The result of verifying "simplecp2' using *ProBMC* is shown in Fig. 6.7. In the example, a deadlock is detected after applying the rule *R2*.

```
CompletedProcess(args=['probcli', 'simplecp2.mch'],
returncode=0, stdout=b'
Deadlock reached after 154 steps (after r2).
( state=1 & a=1 & b=0 )
% Runtime for --execute: 78 ms (with gc: 78 ms, walltime: 82 ms);
time since start: 3901 ms')
```

Figure 6.7 The result of verifying the deadlockfree property of a cP system in ProB

To automately verify cP systems using PAT3, a console version of PAT3 (version 3.5 or higher) needs to be installed and properly configured. Similar to the ProB approach, a subprocess will be created for each verification task.

Four frequently-checked properties including deadlockfreeness, termination, divergencefreeness, and determinism are predefined in the *PAT3MC* function, which can be directly verified. Custom properties can be specified and verified with the *PAT3MCCustom* function. By default, PAT.Console.exe will run in verbose mode in *PAT3MC* and *PAT3MCCustom*. The following line of code can be used to obtain the PAT3 verification results of a cP system. This will be formatted, displayed, and output as a file named "pat3_verification_result.txt".

```
result = subprocess.run(['PAT3.Console.exe', '-csp', '-v', file_path,
                                        output_path], stdout=subprocess.PIPE)
```

An example of the verification result of "simplecp2" using PAT3 is shown in Fig. 6.8, this shows that "simplecp2" is terminating.

```
Assertion: P0() nonterminating
********Verification Result********
The Assertion (P0() nonterminating) is NOT valid.
The following trace leads to a terminating situation.
<init -> cp_init -> r1 -> r2 -> [if((applied == true))]
... -> r1 -> r2 -> ... -> terminate>
********Verification Setting********
Admissible Behavior: All
Search Engine: First Witness Trace using Depth First Search
System Abstraction: False
********Verification Statistics********
Visited States:365
Total Transitions:365
Time Used:0.0256232s
Estimated Memory Used:9019.608KB
```

Figure 6.8 The result of verifying the terminating property of a cP system in ProB

6.1.6 Discussion

cP systems can be verified using different formal tools. However, it is often time-consuming for cP system experts to learn different third-party formal verification tools and their corresponding language syntaxes. To properly model a cP system in a general purpose verification tool is also non-trivial. The tool introduced in this section alleviates the aforementioned issues by providing a one-stop solution for cP system formal verification. It also demonstrates that the transformation guidelines proposed in Section 4 and 5 are feasible.

By having the translators, cP systems can be automatically translated from cPVJ to different modelling languages including B and CSP#. The models generated by the translators can be automatically verified by back-end verifiers. Users can verify cP systems' properties directly without spending too much time learning the grammar syntaxes of back-end verifiers' modelling languages.

In addition to model checkers, a Gallina-translator (Appendix A) is also included in the tool, this can translate cPVJ files to Gallina (following the mapping guideline introduced in Chatper 5). By having the cP-Coq library properly installed and configured, files generated by the Gallina-translator can be verified in Coq. Since Coq is an interactive proof assitant, the verification tasks (proof obligations) need to be specified and proven manually.

A major challenge of using existing general purpose formal verification tools to verify cP systems is that existing formal verifiers are usually insufficient for fully modelling cP systems. This is due to the lack of language features. For example, ProB requires all deferred sets to be given a finite cardinality, and integers can only

be enumerated within MININT to MAXINT. Data structures such as trees are only supported by the String type. CSP# does not support generic containers, only integer arrays can be used to model terms. Additionally, compound terms can only be indirectly modelled using multiple integer arrays. Other limitations of the approach include: 1) cP systems are highly parallel and distributed, thus to manually describe and specify some of their properties in LTL or CTL could be time-consuming or error-prone; 2) the one-way unification supported by cP system rules is extremely hard to totally represent in existing tools.

To solve the issues mentioned above, we designed and implemented a cP system-specific formal verifier named cPV. This can verify several safety and fairness properties of cP systems. In the next subsection, we will introduce the design and implementation of cPV.

6.2 cPV – a simulation and formal verification framework for cP systems

cPV is an open-source cP system simulation and formal verification framework in Python. It includes functionalities such as language parsing, system simulation, verification algorithms, reduction techniques, counterexample generation, and different display options including a graphical user interface (GUI). For valid cP systems, their properties including deadlockfreeness, confluence, termination, determinism, and goal reachability can be verified in cPV.

6.2.1 Overall architecture

cPV is highly modularised, it contains several independent and interchangeable modules. The overall design of cPV is shown in Fig. 6.9.

cPV accepts cPVJ as its data input language. Having a proper cP system written as a cPVJ file means that the parser module in cPV can interpret it and cooperate with the system module to produce an internal represtation, then send it to the simulation or verification module. Both the simulation and verification module contains multiple submodules including engines and algorithm implementations. By performing computing on the internal represtation, the simulation or verification results can be obtained, this can be displayed by the display engine.

The object classes CPSystem, Term, and Rule introduced in the previous section are included in the system module of cPV. Labelled nested multisets in cP systems are implemented by nested hybrid dictionaries. A set of comparators and

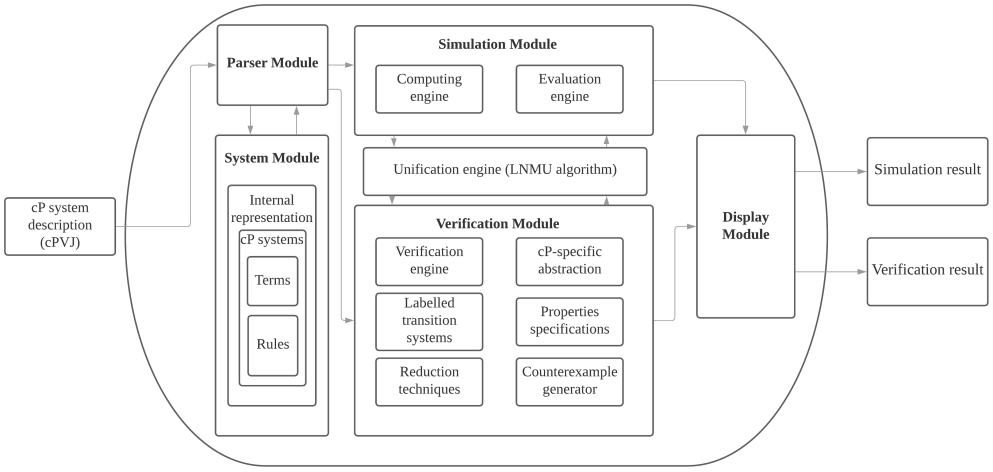

Figure 6.9: The design of cPV

hash functions are implemented for each class to make it a valid key for dictionaries. Although multisets are unordered theoretically, cPV continuously sorts all the dictionaries by keys to figure out identical terms and reduce the statespace.

The parser module in cPV contains several parsing, translating, and validation functions. At the beginning of the simulation or verification, valid cPVJ files will be transformed into a CPSystem object, then they will be passed onto the simulation or the verification module.

For simulation rule applications, a core algorithm is LNMU, this was introduced in Chapter 3. In the unification engine, LNMU is implemented in a sequential way, which can efficiently find all the unifiers between two multisets of terms. As one of the most important components of cPV, the unification engine is shared by both the simulation and verification modules.

In the simulation module, computation needs such as rule validation, term validation, the weak priority order of rules, virtual ground rules, nondeterministic rule application, virtual product membrane, and application models are handled by the computing engine. Additionally, an evaluation engine is implemented to track the simulation information such as running time and memory usage.

cP systems will be transformed into labelled transition systems (LTS) in the verification module. A fundamental data structure for cP system verification is CPNode, which includes terms of the cP systems, state, a set of labels, and a list of visted nodes (trace). Since rules in cP systems cannot be dynamically changed, the verification engine only stores one copy of the rules, rather than repeatedly storing them in every node.

Transitions in the LTS are (potential) virtual ground rule applications. Compared to the simulation engine which randomly selects a group of unifiers for nondeterministic rules, the verification engine considers all the compatible unifier groups. Several search methods are implemented in cPV to traverse its statespace, which includes breadth-first search (BFS), depth-first search (DFS), and heuristic search.

Multiple important properties of cP systems are predefined in the verification module, this includes both safety and liveness properties such as deadlockfreeness, the Church–Rosser property (confluence), termination, determinism, and reachability properties. By specifying the expected terminating states of a cP system, deadlocks (nodes without outgoing edges) can be detected by cPV. As nondeterministic computing models, the Church–Rosser property is expected to be held by most cP systems. cPV performs the confluence check by comparing all the possible halting configurations of a cP system. Some cP systems may be non-terminating, this can be identified by setting a running time/steps limit. A unifier check can be used to verify if a cP system is deterministic. Reachability properties of states and terms are important for some cP models, which can be checked by searching the corresponding statespace.

Basic verification and statespace reduction algorithms are implemented in the verification module. cP systems can work both in a synchronized and asynchronized manner. By applying reduction techniques, the orderings in interleaved transitions will be ignored. Similar to several existing formal verifiers, the statespace of cPV will be generated on the fly.

For instance, to verify a cP system, if a safety property is not held by a configuration, the counterexample generator will save the corresponding node and its trace, then send them to the display module. Several detail levels are defined to display the full or compressed simulation/verification information. A simple GUI for windows is provided, this is implemented using the PyQt5 library.

Several example files including ADD, SUB, MUL, GCD, the subset sum, and Hamiltonian cycle are provided in cPV. Users can either modify these cP models to achieve different computation tasks, or simply use them to get familiar with the syntax of cPVJ.

As an extensible framework, multiple interfaces are provided in cPV, new features can also be added into corresponding modules as required.

6.2.2 The unification engine

To simulate rule applications in cP systems, one of the most challenging tasks is to handle the cP-style unification. In the unification engine of cPV, the LNMU algorithm is implemented in an optimised way (Fig. 6.10).

```
#G: all equations; S: one subtitution; SS: all substitutions.
DEF LNMU(G, S, SS):
  FOR equation IN G:
    equation = GROUND(equation)
    IF FAIL1(equation) OR FAIL2(equation):
      return False
    IF DELETE(equation):
      G = G \ equation
    IF len(G) == 0: #success, all the equations are solved
      return True
    #handle one equation in G
    (mv, mg) = G[0]
    IF HAS_FUNCTOR(mv): #apply FUNCTOR
      succ = False
        FOR t1 IN mv:
          FOR t2 IN mg:
          IF label(t1) == label(t2) AND mv[t1] <= mg[t2]:
            (mv',mg') = (mv,mg)
            mg'[t2] -= mv[t1]
            G1 = [(content(t1), content(t2))] + G[1:]
            G1.append((mv', mg'))
            LNMU(G1, S, SS)
            succ = True
          break #handle one compound term at a time
        return succ
      ELSE: #apply VARIABLE
        FOR v1 IN mv:
          all_comb = []
          FOR t2 IN mg:
            IF mg[t2] >= mv[v1]:
              multiplicity = int(mg[t2] / mv[v1])
              term2_mappings = []
              FOR i = 0; i <= multiplicity; i++:
                bag = {}
                if i > 0:
                  bag[t2] = i
                term2_mappings.append(bag)
              all_comb.append(term2_mappings)
          expanded = expand_combinations(all_comb)
          FOR binding1 IN expanded:
            S' = S + {var -> binding1}
            G' = []
            succ = False
            FOR equation IN G:
              lhs = apply_binding(equation[0], S')
              rhs = equation[1]
              G'.append((lhs, rhs))
            IF (LNMU(G', S1, SS)):
              succ = True
        return succ
```

Figure 6.10: Pseudocode of the LNMU implementation

By properly selecting the order of operations in LNMU (which are unordered and nondeterministic), cPV can find all the unifiers of nested multisets efficiently.

For each equation in set G, cPV first applies the operation GROUND: $G_p = G \cup \{mm_v = mm_g\} \Longrightarrow G'_p = G \cup \{m_v = m_g\}$, which can eliminate the shared ground multiset m from both the lhs and rhs of the equation. For instance, by applying GROUND, the equation $abXf(Y) = abcf(a)f(b)$ will be reduced to $Xf(Y) = cf(a)f(b)$. Equations that satisfy DELETE: $G_p = G \cup \{\lambda = \lambda\} \Longrightarrow G'_p = G$ will be deleted from G.

After reducing all the equations in G, cPV checks if there exists any equation that satisfies the operation FAIL1: $G_p = G \cup \{mm_v = m_g\} \Longrightarrow \perp,\ m \not\subset m_g$ or FAIL2: $G_p = G \cup \{f(m'_v)m_v = m_g\} \Longrightarrow \perp,\ f \notin \varphi(m_g)$. Here $\varphi(m_g)$ is the set which contains all the functors of m_g's first-level compound terms, for example, $\varphi(a1f(b)g(h(c))) = \{f, g\}$. If FAIL1 or FAIL2 is applicable, which means the two multisets are not unifiable, cPV terminates the unification process immediately, and generate an empty output.

For equations which contain compound terms, the operation FUNCTOR: $G_p = G \cup \{f(m'_v)m_v = f(m'_g)m_g\} \Longrightarrow G'_p = G \cup \{m_v = m_g\} \cup \{m'_v = m'_g\}$ will be applied. Compared to LNMU's design, where FUNCTOR is nondeterministically applied once, cPV applies FUNCTOR multiple times to get all possible new equations. For example, to solve the equation $f(X)Y = f(a)f(b)f(c)$, all possible bindings for the variable X including $X \mapsto a$, $X \mapsto b$, and $X \mapsto c$ will be found by cPV. Here corresponding bindings for Y are $Y \mapsto f(b)f(c)$, $Y \mapsto f(a)f(c)$, and $Y \mapsto f(a)f(b)$, respectively.

For equations that do not contain compound terms, cPV applies VARIABLE: $G_p = G \cup \{X^n m_v = m^n m_g\} \Longrightarrow G'_p = G\vartheta_X \cup \{m_v\vartheta_X = m_g\} \cup \vartheta_X,\ \vartheta_X = \{X \mapsto m\},\ X \in V,\ n \in \mathbb{N}^*$. Similar to the implementation of FUNCTOR, all possible bindings of each variable will be found.

The LNMU implementation is used in both the simulation and verification module, and is closely linked to several system properties. For example, a cP system is deterministic if and only if one unifier at most can be found for each rule application.

If a cP system is nondeterministic, all the unifiers found by cPV will be further computed until the system halts. If only one halting configuration can be found, the system is said to be confluent; otherwise, it is NOT confluent.

In having the optimised LNMU implementation, cPV can completely simulate rule applications of cP systems. Most existing cP systems contain well-formed

rules (defined in Section 3.4), which means cPV can unify rules against system terms in linear time using limited memory.

In addition to the unification algorithm implementation, other necessary functions such as binding application, multiset manipulation, and comparators are also included in the unification engine. These functions are also frequently called by other modules of cPV.

6.2.3 The simulation module

The simulation module contains two major engines and a set of supporting functions. cP systems' rule application and the validation of CPSystem, Term, and Rule objects are performed in the computing engine. System information collection, such as running time and memory usage tracking, is done by the evaluation engine.

Fig. 6.11 shows the core function of the simulation engine, which is used to perform rule application. The function APPLY_RULE is recursively defined. To apply a rule it will first perform the state check, if the rule's left state is different from the cP system's state, the rule application will be terminated.

If before applying a rule, the system is already committed to a target state, APPLY_RULE will compare the rule's right state with the committed state. The rule will only be applied if its right state is identical to the committed state.

After passing the state check, if a rule does not contain any lhs terms or promoters, the rule will be directly applied and the rhs terms of the rule will be produced in the product membrane, since no unification is needed.

If the rule is ground, APPLY_RULE will perform the term check and apply it. A rule in the exactly-once model will be applied only once, and max-parallel rules will be applied as many times as possible.

For variable rules, APPLY_RULE will call LNMU function from the unification engine to get all valid unifiers, it will then handle them in different application models. If multiple groups of compatible unifiers exist, following the nondeterminism of cP systems, only one group will be randomly selected.

In the code, the function APPLY_G_RULES is a stateless and looping version of APPLY_RULE, which is defined in Fig. 6.12. APPLY_G_RULES is only used to handle the ground rules that are generated by a group of compatible unifiers.

In having the simulation module implemented, all valid cP systems can be effectively simulated in cPV. Different from modelling and simulating cP systems in

```
DEF APPLY_RULE(cP_system, r1, is_committed = False):
  IF r1.LState() != cP_system.state: return False
  ELIF is_committed AND r1.RState() != cP_system.committed_state:
    return False
  ELIF len(r1.LHS()) == 0 AND len(r1.PMT()) == 0:
    cP_system.ProduceMultiset(r1.RHS())
    return True
  ELIF r1.IsGround():
    IF r1.Model() == '1':
      ms_to_check = lnmu.MultisetUnion(r1.PMT(), r1.LHS())
      IF lnmu.MultisetIn(ms_to_check, cP_system.terms):
        cP_system.ConsumeMultiset(r1.LHS())
        cP_system.ProduceMultiset(r1.RHS())
        return True
      ELSE: return False
    ELSE: #model = '+'
      ms_to_check = lnmu.MultisetUnion(r1.PMT(), r1.LHS())
      ms_to_check_2 = deepcopy(ms_to_check)
      mult = 1
      WHILE lnmu.MultisetIn(ms_to_check_2, cP_system.terms) DO:
        mult += 1
        ms_to_check_2 = lnmu.MultisetTimes(ms_to_check, mult)
      IF mult == 1: return False
      ELSE: mult -= 1
        cP_system.ConsumeMultiset(lnmu.MultisetTimes(r1.LHS(), mult))
        cP_system.ProduceMultiset(lnmu.MultisetTimes(r1.RHS(), mult))
        return True
  ELIF not is_committed
      OR (is_committed AND r1.RState() == cP_system.committed_state):
    ms_to_process = lnmu.MultisetUnion(r1.PMT(), r1.LHS())
    G = []
    G.append((ms_to_process, cP_system.terms, 'in'))
    SS = [] #the set of unifiers
    S = {}
    lnmu.LNMU(G, S, SS)
    IF len(SS) == 0: return False
    ELIF: r1.Model() == '1': #exact-once model
      num_g_rules = len(SS)
      rd_rule = rd.randint(0, num_g_rules - 1)
      unifier = SS[rd_rule]
      lhs2 = lnmu.ApplyBindingMultiset(r1.LHS(), unifier)
      rhs2 = lnmu.ApplyBindingMultiset(r1.RHS(), unifier)
      pmt2 = lnmu.ApplyBindingMultiset(r1.PMT(), unifier)
      r2 = Rule(r1.LState(), r1.RState(), '1') #a unified, ground rule
      r2.SetLHS(lhs2)
      r2.SetRHS(rhs2)
      r2.SetPMT(pmt2)
      IF r2.IsGround(): return APPLY_RULE(cP_system, r2)
      ELSE: return False
    ELSE: #max-parallel model
      rd.shuffle(SS) #no need to keep original SS
      ruleset = []
      FOR unifier IN SS:
        lhs2 = lnmu.ApplyBindingMultiset(r1.LHS(), unifier)
        rhs2 = lnmu.ApplyBindingMultiset(r1.RHS(), unifier)
        pmt2 = lnmu.ApplyBindingMultiset(r1.PMT(), unifier)
        r2 = Rule(r1.LState(), r1.RState(), '+') #a unified, ground rule
        r2.SetLHS(lhs2)
        r2.SetRHS(rhs2)
        r2.SetPMT(pmt2)
        IF r2.IsGround(): ruleset.append(r2)
        return APPLY_G_RULES(cP_system, ruleset)
```

Figure 6.11: The core function to perform rule application in the computing engine

third-party tools which requires human intervention, cPV can perform cP system simulation fully automatically.

```
DEF APPLY_G_RULES(cP_system, ruleset):
  succ = False
  FOR r1 IN ruleset:
    IF APPLY_RULE(cP_system, r1): succ = True
  return succ
```

Figure 6.12: The function APPLY_G_RULES

6.2.4 The verification module

cP systems are modelled as LTS. However in cPV, the labels in the LTS models are not atomic – string, list, and even nested labelled multisets are used as labels of CPNode objects. This design guarantees the information of cP configurations are completely kept, while also creating a huge number of potential branches and a large statespace, which exactly describes how computations are performed in cP systems.

Following the design of cP systems, the trasitions in the LTS models are virtual ground rule application attempts. A generic rule in cP systems can be unified to an arbitrary number of virtual ground rules depending on the terms in the system. In the max-parallel model, a group of compatible virtual ground rules will be applied sequentially, the order of applying these virtual ground rules is unimportant. To reduce the size of the statespace, an algorithm which is inspired by partial order reduction is implemented in the verification module.

6.2.4.1 The core verification algorithm of cPV

cP systems can either be used to perform sequential computations, or describe parallel systems. Several safety and liveness properties are predefined in cPV, these can be verified using corresponding verification algorithms. Fig. 6.13 describes the core verification algorithm in cPV, notably checking functions for different system properties such as deadlockfreeness and goal reaches are placed in the CHECK_HOLD() function. By using different search strategies, the statespace can be checked exhaustively. Using a node list or heap, common graph traversal algorithms such as BFS, DFS, and heuristic search can be applied. Different heuristics can be implemented in the HEAPIFY() function as needed. In the actually implementation, the function names and organisations may be slightly different from the pseudocode.

For a certain node, if the CHECK_HOLD() function returns FALSE, which means a safety propety is not held by the node, the VERIFY() function will

```
DEF VERIFY:
 c = CPNode object for the initial configuration
 node_list = [c]
 WHILE NOT node_list = empty DO:
  IF BFS or Heuristic search:
   current_node = node_list.first()
  ELIF DFS:
   current_node = node_list.last()
  node_list.pop()
  IF NOT CHECK_HOLD(current_node, property):
   return FALSE, current_node //counterexample with trace
  ELSE:
   succ = SUCCESSORS(current_node) //by performing unification
   FOR s IN succ:
    node_list.push(s)
    IF Heuristic search:
     HEAPIFY(node_list)
 return TRUE

DEF SUCCESSORS(node):
 r = ruleset[node.next_rule]
 successors = []
 STATE_CHECK()
 unifiers = LNMU(r.lhs union r.promoters, node.terms)
 cu = COMPATIBLE_UNIFIERS(unifiers)
 cu = REDUCTION(cu)
 FOR u IN cu:
  v_g_rule = APPLY_BINDINGS(r, u)
  succ_node = APPLY_RULE(v_g_rule, node)
  successors.push(succ_node)
 return successors
```

Figure 6.13: The core verification algorithm in cPV

terminate and return FALSE with a counterexample. Otherwise, after the entire statespace has been searched, the VERIFY() function will return TRUE, which indicates the property is held by the system. For example, to detect deadlocks, in the CHECK_HOLD() function, cPV checks if there exists any rule that can be applied to the current node (cP system configuration). If none of the rules can be successfully unified against system terms in the node or match the node's cP state – which indicates a node without any outgoing edge in the LTS – cPV will collect the node as a counterexample and return a deadlock.

Similar to safety properties, liveness properties can also be checked exhaustively. Predefined properties in cPV include the verification needs mentioned in previous cP system studies, furthermore, new properties can also be defined and added to the framework easily. For example, if we expect that a cP system always contains a term $f(1)$, we can add a few lines of code in the CHECK_HOLD()

function. Additionally, if a node's system terms contains $f(1)$, cPV will go to check the next node; otherwise, it will collect the node as a counterexample and return the verification result.

6.2.4.2 Reducing the statespace of cP systems

As one of the P variants, cP systems trade memory for time. Many cP systems can generate exponential terms in each computational step, and during the computation, each rule application may generate exponential groups of unifiers.

For example, to apply a simple max-parallel rule $s_1\ a(X)a(Y) \rightarrow_+ s_2\ b(XY)$ with four terms $a(c)$, $a(d)$, $a(e)$, $a(f)$, the system will first unify the rule against terms where C_4^2 ground rules can be obtained: $s_1\ a(c)a(d) \rightarrow_+ s_2\ b(cd)$, $s_1\ a(c)a(e) \rightarrow_+ s_2\ b(ce)$, $s_1\ a(c)a(f) \rightarrow_+ s_2\ b(cf)$, $s_1\ a(d)a(e) \rightarrow_+ s_2\ b(de)$, $s_1\ a(d)a(f) \rightarrow_+ s_2\ b(df)$, and $s_1\ a(e)a(f) \rightarrow_+ s_2\ b(ef)$. Following this a compatible group of the ground rules will be randomly selected and applied.

A naive approach to finding all the groups of compatible unifiers can be directly obtained by following the design of cP systems. Suppose that for a rule application, k unifiers can be found, this means we can generate the permutations (arrangements) of $\{1, 2, \ldots, k\}$ including $[1, 2, \ldots, k-1, k]$, $[1, 2, \ldots, k, k-1]$, ..., and $[k, k-1, \ldots, 2, 1]$. For cP system simulation, one of the arrangements will be selected and applied; for verification, all of them need to be checked. For each arrangement of unifiers, a group of corresponding ground rules will be applied following the top-down priority order.

The naive approach works properly, however, its space complexity is factorial, since $P_k^k = k!$. In cPV, we optimised this approach by eliminating the internal orders. When k unifiers ϑ_1, ϑ_2, ..., ϑ_k are applied, the order of applying the corresponding ground rules is actually irrelevant. In other words, it is important to know the truth that $s_1\ a(c)a(d) \rightarrow_+ s_2\ b(cd)$ and $a(e)a(f) \rightarrow_+ s_2\ b(ef)$ are compatible and will be applied. However, we do not need to know which of them will be applied first.

Thus, for each rule application, we use binary numbers to label the unifiers: each of them can be chosen (labelled as 1) or ignored (labelled as 0). For k unifiers, we will have at most 2^k groups of unifiers. The pseudocode is shown in Fig. 8.5. By looping all the unifier labels from total $(2^k - 1)$ to 0, the system can check compatible unifiers from larger sets to smaller sets. If a set of unifiers is successfully applied, all of its subsets can be skipped. For example, if a set of uni-

fiers $\{\vartheta_1, \vartheta_2, \vartheta_3\}$ is compatible, we do not need to recompute its subsets $\{\vartheta_1, \vartheta_2\}$, $\{\vartheta_1, \vartheta_3\}$, ..., $\{\vartheta_3\}$ as they must be compatible, too.

```
DEF MAX_PARALLEL_RULE():
  total = pow(2, len(SS)) − 1
  applied_unifiers = []
  FOR x IN range(total,−1,−1):
    needed = True
    FOR x1 IN applied_unifiers:
      IF self.IsSubset(x, x1):
        needed = False
        break
    IF not needed: continue
    ...
    unifier_ids = []
    position = len(SS) − 1
    WHILE position >= 0:
      IF x % 2 == 1: unifier_ids.append(position)
      position −= 1
      x = int(x / 2)

    all_applied = True
    FOR i IN unifier_ids:
      ...
      IF can_be_applied(i): apply(i)
      ELSE:
        all_applied = False
        break

    IF all_applied:
      applied_unifiers.append(x)
      ...

DEF ISSUBSET(child, parent):
  IF parent < child: return False
  ELIF parent == child: return True
  IF parent % 2 == 0 and child % 2 == 1: return False
  ELSE: return self.IsSubset(int(child / 2), int(parent / 2))
```

Figure 6.14: The statespace reduction pseudocode for rule unification

In having the reduction function, for max-parallel rules, as opposed to the original cP system design, cPV can reduce the statespace from $\mathcal{O}(n!)$ to $\mathcal{O}(2^n)$. This can drastically boost the speed of cP system verification.

6.2.5 Display module

cPV includes a GUI and a set of display functions. Here the GUI was implemented using the PyQt5 library, which has a main display window with multiple interactive components.

The display module loads the system modules to read and write cPVJ files, it also loads other modules during the cP system initialization, simulation and verification.

The GUI provides several cP system simulation and verification options, and custom properties can be manually input via interactive textboxes.

cP systems including ADD, SUB, MUL, GCD, the subset sum, and Hamiltonian cycle can be quickly imported from the menu button. Users can also define, save, and load their own cP systems with custom rules using the textboxes and menu buttoms provided by the GUI.

By selecting different detail levels, cP systems' simulation and verification results will be displayed differently.

Fig. 6.15 shows a screenshot of cPV, this shows a verification result of the Church–Rosser property of a cP system that solves the Hamiltonian cycle problem. Since the cP system does not satisfy the Church–Rosser property, a counterexample is displayed.

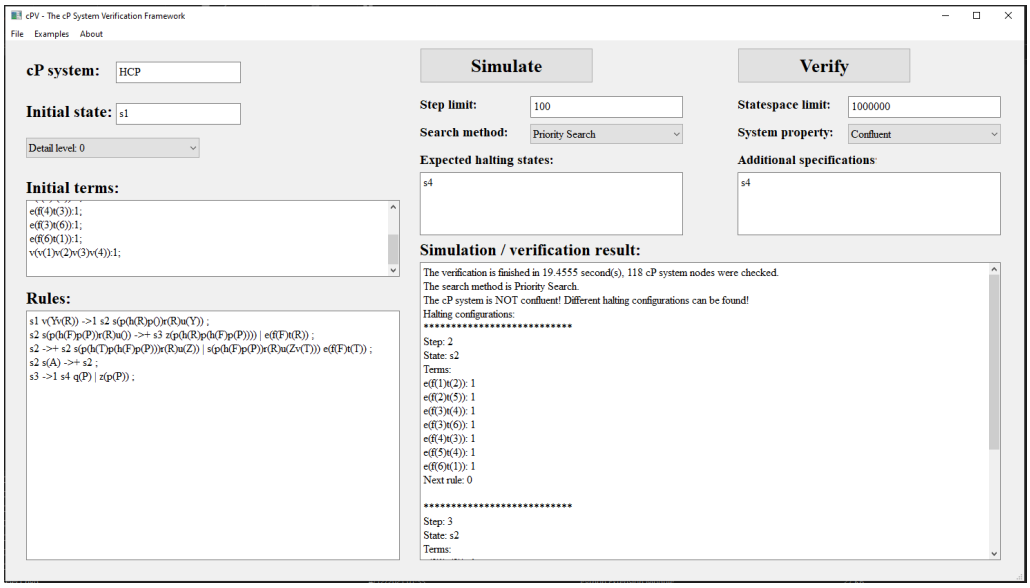

Figure 6.15: A cPV screenshot of verifying a cP system that solves the Hamiltonian cycle problem

6.2.6 Summary

Different from other P variants, cP systems support complex symbols and generic rules. This significantly increases its representational power, whilst also making them more difficult to simulate or verify. Most cP systems proposed in previous studies were not formally verified – the evaluation and verification of the cP systems were often done by performing manual system simulation during case studies.

Our previous work (Chapter 4 and 5) introduced how to verify particular cP systems using ProB, PAT3 and Coq, however, these approaches are done manually and require human intervention. In this section, we introduced two different implementations of cP system formal verification: (1) a software tool which is integrated with existing formal tools, and (2) a cP system-specific simulation and verification framework (cPV).

cP systems that solve NPC problems or model parallel and/or distributed algorithms are often expected to satisfy several important system properties. These include deadlockfreeness, confluence, termination, and goal reachability. cP systems that model parallel and/or distributed systems are expected to be deadlockfree. For a cP system that performs sequential computing, terminal states may occur as a natural phenomenon; thus apart from the expected terminal states, the rest of the system still needs to be deadlockfree. Most cP systems are nondeterministic, to make sure they can consistently solve certain problems correctly, furthermore, they are highly recommended to be confluent. Similarly, to make sure problem solving cP systems can always generate outputs, they are expected to be terminating. All the system properties described above can be easily verified in cPV automatically.

cPV includes functionalities such as language parsing, system simulation, verification algorithms, reduction techniques, counterexample generation, and different display options including a simple GUI. As a highly modularised extensible framework, different modules in cPV are loosely coupled, new functionalities and algorithms can be easily added to cPV to meet different verification requirements. Major contributions of this chapter follow.

- The intergrated tool proves that the cP system formal verification approaches and transformation guidelines proposed in our previous studies are feasible.
- As the first effective cP system simulator and verifier, cPV supports the automated analysis of all valid cP systems.
- A domain specific language for cP systems (cPVJ) is proposed, which is fully supported by cPV.
- cPV can be easily extended to meet different verification requirements.
- The entire approach introduced in this chapter can be adopted by other verification frameworks.

In the next chapter, we will conduct both quantitative and qualitative evaluations of the cPV framework, and compare it with other membrane computing software implementations.

Chapter 7

Evaluation of cPV

Having the cPV framework implemented, we can simulate cP systems' behaviours and verify their properties. In this section, we will first conduct a case study, to illustrate the cP system modelling and simulation in cPV, and then evaluate cPV from both functional and performance aspects.

In the case study, a cP system that solves the Hamiltonian cycle problem (HCP) will be represented as a cPVJ file and then will be simulated in cPV.

In the evaluation of cPV, two benchmark cP system datasets will be created and verified, these can also be used in future studies. The experiment result shows that it is feasible to correctly verify cP systems' properties including deadlockfreeness, confluence, termination, determinism, and goal reachability using cPV.

The chapter is organised as follows. Section 7.1 shows the modelling, simulation, and verification of a cP system in cPV; Section 7.2 introduces the evaluation of cPV; Section 7.3 compares cPV to other P system simulation and verification implementations; and Section 7.4 summarises the chapter.

7.1 A case study

HCP is long-standing, well-known computationally hard problem, it is a special case of the travelling salesman problem (TSP), which determines whether a Hamiltonian cycle exists in a given graph.

A cP system Π_{HCP} was proposed in [5], it consists of five rules, and solves HCP in linear time (Fig. 7.1). In Π_{HCP}, all the vertices in a graph are represented as a term $v(v(X), v(Y), v(Z), \ldots)$, where X, Y, and Z are unique IDs of the vertices. An arc from vertex I to vertex J is encoded as $e(f(I)t(J))$, and the initial state of the system is s_1.

$$
\begin{array}{lcll}
s_1\ v(v(R)Y) & \to_1 & s_2\ s(r(R)u(Y)p(h(R)p())) & (R1) \\
s_2\ s(r(R)u()p(h(F)p(P))) & \to_+ & s_3\ z(p(h(R)p(h(F)p(P)))) \mid e(f(F)t(R)) & (R2) \\
s_2 & \to_+ & s_2\ s(r(R)u(Z)p(h(T)p(h(F)p(P)))) & (R3) \\
 & & \mid s(r(R)u(v(T)Z)p(h(F)p(P)))\ e(f(F)t(T)) & \\
s_2\ s(\_) & \to_+ & s_2 & (R4) \\
s_3 & \to_1 & s_4\ p'(P) \mid z(p(P)) & (R5)
\end{array}
$$

Figure 7.1: The ruleset of Π_{HCP}

R1 selects an arbitrary vertex R from $v(v(R)Y)$ to become the starting point of the cycle, and create a term $s(r(R)u(Y)p(h(R)p()))$. The subterm $u(Y)$ stores the remaining unexplored vertices in the graph, and $p(h(R)p())$ tracks the cycle's path so far.

The term $z(p(h(R)p(h(F)p(P))))$ in *R2* is used to store the Hamiltonian path, and *R3* generates all the valid paths from current vertex following existing edges $e(f(F)t(R))$ in the system. As max-parallel rules, all the valid paths will be generated by *R2* and *R3*.

In each step, *R4* consumes $s(\_)$ terms which are already expanded. *R5* terminates the cP system and outputs a Hamiltonian cycle. When multiple Hamiltonian cycles can be found by the ruleset, one of them will be randomly selected. In total, the cP system can find a Hamiltonian cycle of a graph in $n + 3$ steps.

An example cPVJ representation of Π_{HCP} is shown in Fig. 7.2. In the example, the input graph contains 6 vertices and 7 edges, which is shown in Fig. 7.3.

```
{"ruleset": ["s1 v(v(R)Y) −>1 s2 s(r(R)u(Y)p(h(R)p()))",
"s2 s(r(R)u()p(h(F)p(P))) −>+ s3 z(p(h(R)p(h(F)p(P)))) | e(f(F)t(R))",
"s2 −>+ s2 s(r(R)u(Z)p(h(T)p(h(F)p(P)))) |
    s(r(R)u(v(T)Z)p(h(F)p(P))) e(f(F)t(T))",
"s2 s(A) −>+ s2",
"s3 −>1 s4 q(P) | z(p(P))"],
"terms": {"e(f(1)t(2))": 1, "e(f(2)t(5))": 1,
"e(f(5)t(4))": 1, "e(f(3)t(4))": 1,
"e(f(4)t(3))": 1, "e(f(3)t(6))": 1,
"e(f(6)t(1))": 1, "v(v(1)v(2)v(3)v(4)v(5)v(6))": 1},
"state": "s1",
"name": "HCP"}
```

Figure 7.2: A cPVJ example of the cP solution to HCP

As mentioned, in the cPVJ representation, no anonymous variable is allowed, thus the $s(\_)$ in *R4* is replaced by $s(A)$. Each rule of the cP system is written as a string object in the field "ruleset", and initial system terms are stored in the

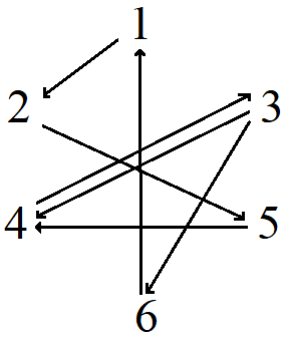

Figure 7.3: The input graph of the cP system

"terms" field. The cP system starts at s_1, and a string name "HCP" is given to the system.

By saving the example cPVJ as a JSON file, cPV can read, simulate, and verify it. A screenshot of simulating Π_{HCP} in cPV is shown in Fig. 7.4

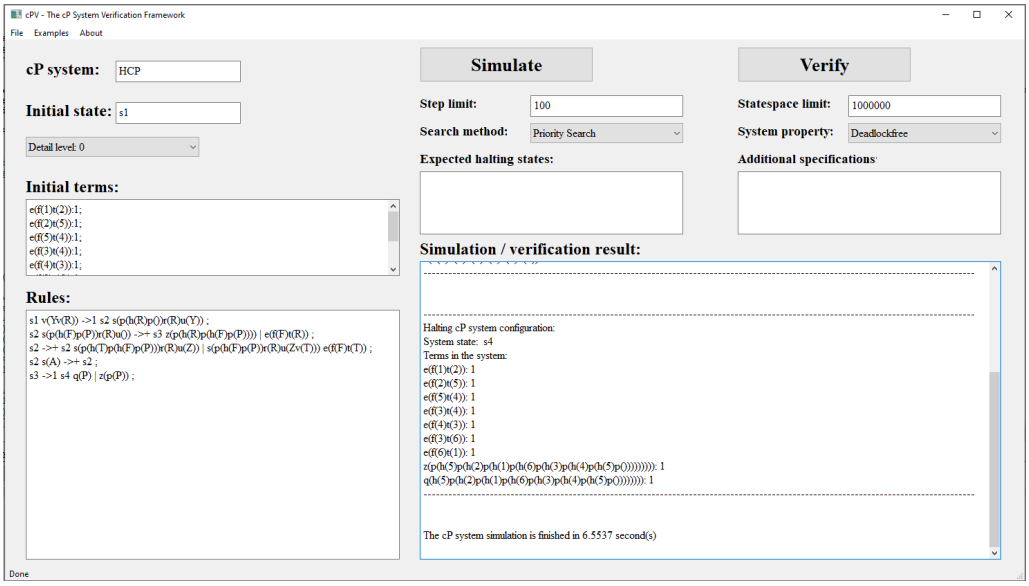

Figure 7.4: A screenshot of simulating Π_{HCP} system in cPV

Since the cP system is non-deterministic, different computation results can be obtained by simulating the cP systems multiple times. In the example, a Hamiltonian cycle $5 \to 2 \to 1 \to 6 \to 3 \to 4 \to 5$ is found, which is indicated by the term $q(h(5)p(h(2)p(h(1)p(h(6)p(h(3)p(h(4)p(h(5)p())))))))$.

By choosing the detail level 2 in cPV, system information including configurations, rule attempts, rule unifications, and the production and consumption of

terms will be tracked and displayed in cPV. In having $n = 6$ vertices, the system found the Hamiltonian cycle in 9 steps, which is $n + 3$.

In having specified the target state s_4 in cPV, it can verify the system's deadlock property. The screenshot in Fig.7.5 shows that the system is deadlock free. To perform simulation, only one random branch of the computation tree needs to be expanded; and to conduct verification, all possible applications of non-deterministic rules need to be verified. Thus, the running time of cP system verification is often longer than simulation.

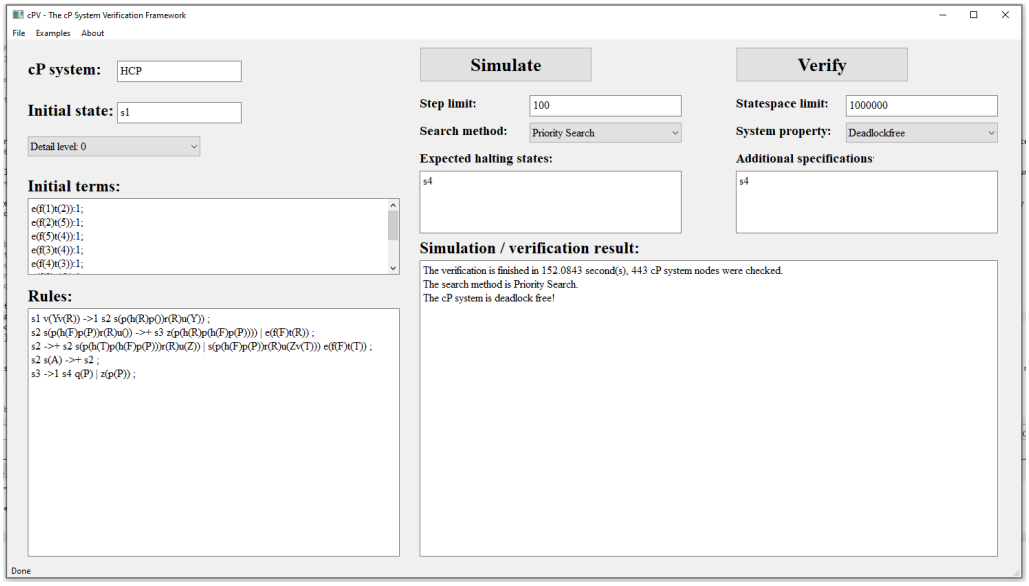

Figure 7.5: The deadlock verification result of Π_{HCP}

The verification results of other system properties is shown in Fig.7.6. The system is terminating, nondeterministic, and the expected halting state s_4 is eventually reached. For the Church–Rosser property, since it is not held by the system, a counterexample is given by cPV, this includes two Hamiltonian cycles starting from different vertices: $1 \to 2 \to 5 \to 4 \to 3 \to 6 \to 1$ and $2 \to 5 \to 4 \to 3 \to 6 \to 1 \to 2$.

Throughout the case study, we can find that it is feasible to simulate and verify complex cP systems such as Π_{HCP} using cPV. By specifying the expected target state or goal terms, several cP system properties can be effectively verified. When a cP system does not hold certain properties such as the Church–Rosser property, counterexamples will be found and displayed. In the next section, we will introduce how to evaluate the performance of cPV.

```
The cP system is NOT confluent!
Different halting configurations can be found!

Halting configurations:
————————————————————————
State: s4
Terms: ...
z(p(h(1)p(h(6)p(h(3)p(h(4)p(h(5)p(h(2)p(h(1)p()))))))))): 1
————————————————————————
State: s4
Terms: ...
z(p(h(2)p(h(1)p(h(6)p(h(3)p(h(4)p(h(5)p(h(2)p()))))))))): 1

The cP system is terminating!
The cP system is nondeterministic!
The target state s4 is reachable!
The target state s4 is eventually reached!
```

Figure 7.6: The verification result of Π_{HCP} (other system properties)

7.2 Evaluation of cPV

We evaluate cPV from both functional and performance aspects. cP systems are theoretically computationally efficient. However, in the actual implementation of cP systems, without having unbounded memory, simulation and verification can be slow. This is due to different reasons, such as generation of a large number of terms, production of highly nested terms, or exponential unifiers found by unification processes.

By conducting the evaluation, the following two questions will be answered:

- Can cPV effectively verify existing cP systems' properties including deadlockfreeness, confluence, termination, determinism, and goal reachability?
- How effective is the performance of simulating different cP systems and verifying their properties?

Two cP system datasets are constructed to evaluate cPV. These cover different aspects of cP systems, such as cP systems that solve NPC problems, cP systems producing highly nested terms, and cP systems which generate exponential terms in each step.

For cP systems in each dataset, we verify several major properties that were emphasised in multiple previous cP system studies. Most published cP systems are designed to solve certain problems, these are usually expected to be deadlockfree, terminating, confluent, and able to reach the expected goal states.

7.2.1 Experiments setup

In order to answer the evaluation questions, we constructed two cP system datasets *D1* and *D2*, which are used for functionality and performance evaluation, respectively. The two datasets include several cP systems published in previous studies, and two cP systems which were particularly created for the evaluation.

In this study, a laptop with an Intel i7-7700HQ CPU (2.80GHZ, 2.81GHZ) and 16G RAM will be used to conduct the evaluation.

7.2.1.1 Dataset for the functionality evaluation

The first dataset *D1* includes the following four cP systems, which are used to perform the functionality evaluation.

- Π_1: a cP system that computes GCD of two natural numbers using only ground rules [105].
- Π_2: a cP system that computes GCD of two natural numbers using generic rules.
- Π_3: a cP system that solves SSP [4].
- Π_4: Π_{HCP}.

In *D1*, Π_1 computes GCD of two natural numbers, these are represented as multiplicities of two atoms a and b, for instance, a^{144} and b^{88}. The cPVJ representation of Π_1 is shown in Fig. 7.7. Two max-parallel ground rules $s_1\ ab\ \rightarrow_+\ s_1\ b$ and $s_1\ b\ \rightarrow_+\ s_1\ a$ are included in the system, these describe the Euclidean algorithm. In Π_1, neither compound terms nor generic rules are used. Thus, it can be easily transformed into a traditonal P system.

```
{"ruleset": ["s1 a b ->+ s1 b",
"s1 b ->+ s1 a"],
"terms": {"a":144, "b":88},
"state": "s1",
"name": "GCD1"}
```

Figure 7.7: The cPVJ representation of Π_1

Π_2 shows a cP system version of the Euclidean algorithm. The cPVJ representation of Π_2 is shown in Fig. 7.8. Using the generic rules $s_1\ a(XY1)\ \rightarrow_1\ s_1\ a(Y1)\ |\ a(X)$ and $s_1\ a(X)\ a(X)\ \rightarrow_1\ s_2\ b(X)$, only two compound terms are

needed in the system. The first rule computes the difference of the two numbers, and uses the difference to overwrite the larger number. The system halts when the two numbers in the cP system are the same, then the second rule will generate the computation result. To compute the GCD of 144 and 88, as opposed to Π_1 which needs 144 copies of a and 88 copies of b in the system, Π_2 only needs two compound terms $a(1^{144})$ and $a(1^{88})$ in its initial configuration.

```
{"ruleset": ["s1 a(XY1) −>1 s1 a(Y1) | a(X)",
"s1 a(X) a(X) −>1 s2 b(X)"],
"terms": {"a(144)": 1, "a(88)": 1},
"state": "s1",
"name": "GCD2"}
```

Figure 7.8: The cPVJ representation of Π_2

Π_3 is the cP solution to SSP that is mentioned in Chapter 4. As a NPC solution, an exponential number of terms can be generated in every step, this is a challenge to implement the cP system in real life. Compound terms with fixed nesting depths are used in Π_3, the cPVJ representation of the cP system is shown in Fig. 7.9.

```
{"ruleset": ["s0 −>1 s1 p(n(M)s()u()) | m(M)",
"s1 −>1 s2 o(X) | p(As(T)u(X)) t(T)",
"s1 −>1 s3 o() | p(An())",
"s1 −>+ s1 p(n(Z)s(SY)u(Xm(Y))) | p(n(Zm(Y))s(S)u(X))",
"s1 p(A) −>+ s1"],
"terms": {"m(m(1)m(2)m(3))":1, "t(6)":1},
"state": "s0",
"name": "SSP1"}
```

Figure 7.9: The cPVJ representation of Π_3

Π_4 is the solution to HCP (Fig. 7.2). Compound terms in Π_4 can be highly nested, which challenges the simulation and verification speed of cPV. Since the number of terms that can be generated for a complete graph is extremely large, to evaluate cPV, we choose graphs with random edges (arcs).

7.2.1.2 Dataset for the performance evaluation

The second dataset *D2* is designed for the performance evaluation of cPV, which also includes four cP systems. *D2* also includes Π_3 and Π_4, which are published cP system solutions to NPC problems. The search space of SSP is $\mathcal{O}(2^n)$, and the worst-case search space of HCP is $\mathcal{O}(n!)$.

- Π_3.
- Π_4.
- Π_5: a cP system which generates exponential terms in each step.
- Π_6: a cP system which generates highly nested terms.

To evaluate the performance of cP system simulation, we choose problem sizes $n = 4, 6, 8$ for Π_3, and $n = 3, 4, 5$ for Π_4. Complete graphs are considered for Π_4.

To conduct the evaluation of cP system verification, we choose problem sizes $n = 3, 4, 5$ for both Π_3 and Π_4. Instead of complete graphs, randomly connected graphs are chosen.

Π_5 is a "cell division" cP system – given a natural number n, it can generate 2^n terms in n steps (Fig. 7.10). The system term $a(\_)$ is used to generate exponential new terms, and $b(\_)$ is used to indicate the problem size. For the simulation of Π_5, we choose $b = 15, 20, 25$; and for the verification, we choose $b = 10, 15, 20$.

```
{"ruleset": ["s1 b(X) −>1 s2 o(X) | a(XY) ",
"s1 a(X) −>+ s1 a(X1) a(X1) "],
"terms": {"a(1)":1, "b(10)":1},
"state": "s1",
"name": "EXP1"}
```

Figure 7.10: The cPVJ representation of Π_5

Π_6 can generate compound terms with arbitrary nesting depths (Fig. 7.11). The first rule $s_1\ b(X)\ c(X)\ \rightarrow_1\ s_2\ o(X)$ terminates the computation when the nesting depth reaches the limit $b(X)$. Otherwise, the cP system will apply the second rule $s_1\ a(X)\ c(Y)\ \rightarrow_+\ s_1\ a(a(X))\ c(Y1)$ and increase the nesting depth by one. For the simulation of Π_6, we choose $b = 5, 10, 15$; and for the verification, we choose $b = 10, 11, 12$.

```
{"ruleset": ["s1 b(X) c(X) −>1 s2 o(X) ",
"s1 a(X) c(Y) −>+ s1 a(a(X)) c(Y1) "],
"terms": {"a(1)":1, "c(1)":1, "b(10)":1},
"state": "s1",
"name": "NES1"}
```

Figure 7.11: The cPVJ representation of Π_6

Given a cP system in *D2*, the corresponding LTS in cPV may contain a large number of nodes depending on the system's rules. Furthermore, each node may

contain an exponential or factorial number of compound terms. By increasing the problem sizes of the cP systems in *D2*, the memory usage will be increased drastically, which is ideal for the performance evaluation.

7.2.2 Evaluation results

To perform the functionality evaluation of cPV, we verify properties of cP systems in *D1*, and manually check the correctness of the experiment results.

To conduct performance evaluation, we simulate and verify cP systems in *D2*, analyse the experiment results, and discuss each cP system in detail.

7.2.2.1 Functionality evaluation result

For each cP systems in *D1*, we choose three instances and verify their properties. These include deadlockfreeness, confluence, termination, determinism, and goal reachability. Counterexamples displayed by cPV will be checked manually.

The functionality evaluation is shown in Table 7.1. Π_1 describes the Euclidean algorithm, which is expected to be terminating. The system should halts at s_1 with a multiset of goal terms a^x, where x is the GCD of the two input numbers. If the two numbers are coprime, the goal term will be a. Since the system only contains ground rules with a top-down priority order, it is deterministic. During the computation of GCD, the system should not include any rules which may cause a deadlock. The verification results of Π_1 meets our expectations, which shows that Π_1 is deadlockfree, confluent, terminating, deterministic, and can eventually reach the goal state.

Table 7.1: The verification results of cP systems in *D1*

Systems	deadlockfree	confluent	terminating	deterministic	goal reachability
Π_1	✓	✓	✓	✓	✓
Π_2	✓	✓	✓	✓	✓
Π_3	✓	✗*	✓	✗	✓
Π_4	✓	✗*	✓	✗	✓

(*): counterexample displayed

Although Π_2 uses generic rules, the system only contains two compound terms, and the rules' unification should generate unique unifiers. Thus, the system is expected to be deterministic. The system should halts at s_2 with a goal term $o(\_)$, which indicates the GCD of the two input numbers. If the two numbers are coprime, the goal term will be $o(1)$. As in the description of the Euclidean algorithm, Π_2 should be terminating. For each pair of input natural numbers, there

only exist one GCD, thus, Π_2 is supposed to meet the Church–Rosser property. The verification result also meets our expectations – Π_2 is deadlockfree, confluent, terminating, deterministic, and can eventually reach the goal state.

Π_3 is the solution to SSP. If multiple subsets A_1, A_2, ... A_m of an original set S satisfy $\Sigma_{x \in A_i} x = T, i \in [1, m]$, where T is the target number; Π_3 will find all these subsets and keep them as $p(\_)$ terms in the system. Since all the instances of Π_3 we chose for the experiment have at least two solutions in the same layer, Π_3 should not be confluent (Fig. 7.12). The system is not expected to be deterministic, since the unification of its rules can generate different groups of mappings during the computation. However, it has to be deadlockfree and terminating – it should eventually halt at s_2 or s_3, and generates a goal term $o(\_)$. The experiment results shows that Π_3 is deadlockfree, confluent, terminating, nondeterministic, and can eventually reach one of the goal states, this fulfils our expectations.

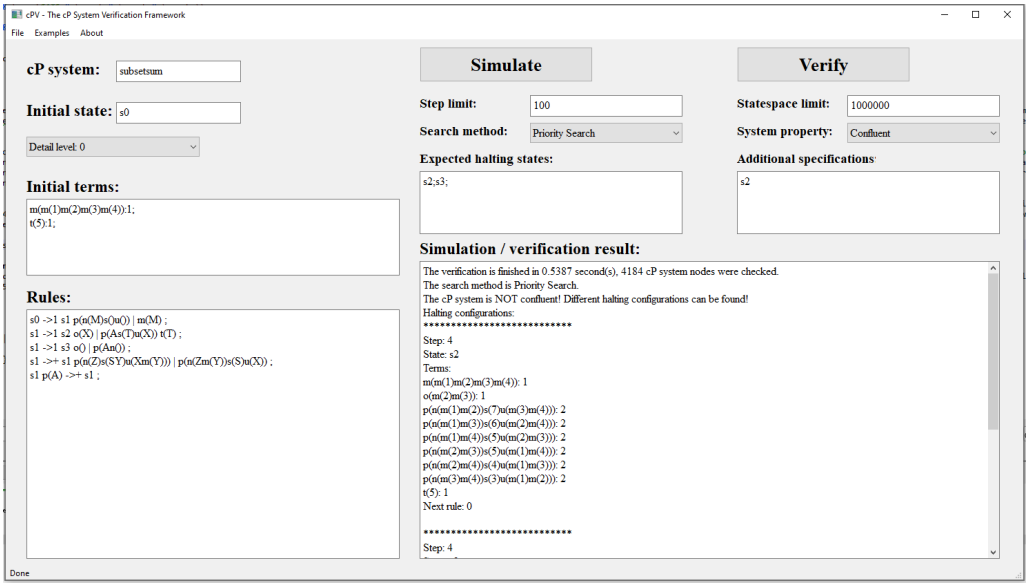

Figure 7.12: The Church-Rosser property verification result of Π_3

Π_4 does not hold the Church–Rosser property. If multiple Hamiltonian cycles can be found in a graph, it will randomly select one of them. Thus, goal terms generated by Π_4 can be different. The system is nondeterministic since multiples rules in Π_4 process random terms in the system. However, as a problem solution, Π_4 is still expected to be deadlockfree and terminating. If a Hamiltonian cycle can be found, the system will halt at s_4, otherwise it will halt at s_2. The evaluation results correctly verifies all the aforementioned properties.

In conclusion, all the system properties of cP systems shown in *D1* were correctly verified in cPV. For properties which are not held by certain cP systems, counterexamples were correctly found and displayed.

7.2.2.2 Performance evaluation result

Using cPV, we simulated all the cP systems in *D2*. The running time is shown in Table 7.2. In the experiment, worst-case scenarios were considered for each cP system. For example, we chose problems instances of Π_3 with zero solutions, and chose complete (fully-connected) graphs for Π_4.

As NPC solutions, the simulation time of Π_3 and Π_4 drastically increases with the problem sizes. For Π_5 and Π_6, the simulation becomes relatively slow when the cP system attempts to generate around 2^{25} terms in one step, or generate terms with a nesting depth of 15.

Table 7.2: Simulation time of cP systems in *D2*

Systems	Problem Size	Simulation Time (s)
Π_3	n=4	0.6234
Π_3	n=6	7.7435
Π_3	n=8	233.8670
Π_4	n=3	0.5562
Π_4	n=4	7.3641
Π_4	n=5	264.8018
Π_5	b=15	0.1908
Π_5	b=20	5.2841
Π_5	b=25	189.0755
Π_6	b=5	0.0185
Π_6	b=10	2.6434
Π_6	b=15	674.2326

If multiple groups of unifiers are found in a rule application, the simulation module of cPV will non-deterministically select one of them. Thus, only one branch of the computation tree will be explored during the system simulation. However, when a cP system generates exponential or highly nested terms, a large amount of memory is required, thus the running time significantly increases with the problem sizes.

For cP system verification, the following random instances of cP systems in *D2* were chosen:

- Π_3, $n = 3, S = \{1, 2, 3\}, T = 6$.
- Π_3, $n = 4, S = \{1, 2, 3, 4\}, T = 9$.
- Π_3, $n = 5, S = \{1, 2, 3, 4, 5\}, T = 14$.

- Π_4, $n = 3$, a complete graph.
- Π_4, $n = 4, E = \{1 \to 2, 1 \to 3, 1 \to 4, 2 \to 1, 2 \to 3, 2 \to 4, 3 \to 1, 3 \to 2, 4 \to 3\}$.
- Π_4, $n = 5, E = \{1 \to 2, 1 \to 3, 1 \to 4, 1 \to 5, 2 \to 1, 2 \to 3, 2 \to 4, 3 \to 1, 3 \to 2, 4 \to 3, 5 \to 4\}$.
- Π_5, $b = 10, 15, 20$.
- Π_6, $b = 10, 11, 12$.

The property verfication result is shown in Table 7.3, where the search method "Priority" was chosen. Using different search method such as BFS or DFS may affect the verification time for certain properties, but the worst-case scenario will remain the same. For example, to verify a cP system which holds a safety property, since no counterexample can be found, the entire statespace needs to be traversed despite the search strategy.

For the instances of Π_3, we can find that the number of internal nodes (potential rule applications) drastically increases with the problem size. This is due to the unification of max-parallel rules. For example, if k unifiers $\vartheta_1, \vartheta_2, \ldots, \vartheta_k$ can be found for a max-parallel rule, the total groups of possible compatible unifiers are 2^k, this includes $g_1 = \{\vartheta_1\}, g_2 = \{\vartheta_2\}, \ldots, g_{2^k} = \{\vartheta_1, \vartheta_2, \ldots \vartheta_k\}$. All the unifier groups $g_i, i \in [1, 2^k]$ need to be verified exhaustively, thus, cPV may need to verify exponential internal nodes for each max-parallel rule application. Even for the problem size $n = 5$, the number of internal nodes generated by the cP system exceeded the statespace limit of cPV, which is 1000000. The experiment also shows why cP systems can solve NPC problems in a small number of steps; because in every step, exponential memory and "processors" will be used to perform the computation parallelly.

To verify Π_4, cPV needs to process an exponential number of highly nested terms such as $z(p(h(2)p(h(1)p(h(6)p(h(3)p(h(4)p(h(5)p(h(2)p())))))))))$. Since the worst-case scenario for HCP is factorial, considering Π_4 uses multiple max-parallel rules, it is impossible to verify complete graphs with a large number of vertices. We used graphs with around $n * (n-1)/4$ edges to conduct the evaluation, and found that processing highly nested terms is much slower than handling compound terms with a nesting depth of 1, 2, or 3. Highly nested terms are not friendly to cPV's nested object structures.

Table 7.3: Property verification time of cP systems in *D2*

Systems	deadlockfree	confluent	terminating	deterministic	goal reachability
Π_3 (n=3)	✓	✓	✓	✗	✓
Time (s)	0.2245	0.2282	0.2291	0.0191	0.2220
Nodes visited	129	139	139	14	129
Π_3 (n=4)	✓	✓	✓	✗	✓
Time (s)	0.7555	0.6787	0.6777	0.0309	0.6551
Nodes visited	8321	8331	8331	14	8321
Π_3 (n=5)	✓	✓	✓	✗	✗
Time (s)	8.1454	8.2526	8.2834	0.0555	8.2281
Nodes visited	1048267*	1048267*	1048267*	14	1048267*
Π_4 (n=3)	✓	✗	✓	✗	✓
Time (s)	3.7566	6.8857	7.6740	0.0076	4.9720
Nodes visited	136	182	226	1	161
Π_4 (n=4)	✓	✗	✓	✗	✓
Time (s)	15.3510	16.9211	17.7229	0.0125	15.3904
Nodes visited	297	315	337	1	294
Π_4 (n=5)	✓	✗	✓	✗	✓
Time (s)	72.6052	80.0983	85.2717	0.0277	72.3948
Nodes visited	781	798	831	1	777
Π_5 (b=10)	✓	✓	✓	✗	✓
Time (s)	0.0714	0.0697	0.0675	0.0053	0.0693
Nodes visited	161	165	165	8	161
Π_5 (b=15)	✓	✓	✓	✗	✓
Time (s)	0.2369	0.2478	0.2513	0.0054	0.2716
Nodes visited	2342	2346	2346	8	2342
Π_5 (b=20)	✓	✓	✓	✗	✓
Time (s)	0.9713	1.0872	0.9681	0.0054	0.9758
Nodes visited	65835	65839	65839	8	65835
Π_6 (b=10)	✓	✓	✓	✓	✓
Time (s)	3.0985	3.2475	3.2507	3.3566	2.9663
Nodes visited	31	35	35	35	31
Π_6 (b=11)	✓	✓	✓	✓	✓
Time (s)	9.3661	10.6740	9.9801	9.9602	8.9828
Nodes visited	34	38	38	38	34
Π_6 (b=12)	✓	✓	✓	✓	✓
Time (s)	27.7904	33.4176	32.2899	33.0137	27.0599
Nodes visited	37	41	41	41	37

(*): exceeding cPV's default statespace limit (1000000)

Π_5 and Π_6 cover the two major challenges of cP system verification separately: (1) an exponential number of system terms, and (2) highly-nested compound terms. From the experiment results we can find that cPV can efficiently verify cP systems with around 2^{20} terms (around 1 second). However, to verify terms with a nesting depth of 12, cPV was relatively slow, which needs around 30 seconds to verify a system property.

In the results, the verification of the deterministic property can often terminate early. Following the definition of cP systems, if one generic rule can be unified against multiple groups of terms, one group will be randomly selected, which will make the cP system nondeterministic. For example, in Π_4, the first rule will randomly select a vertex from a graph as a starting point of a path, which makes the system nondeterministic. By checking one rule, cPV already knows that Π_4 is

nondeterministic, thus it can terminate early and output the verification result.

The experiment results shows that it is feasible to use cPV to correctly verify different cP systems' properties. cPV can check more than 1000000 internal states within 9 seconds, or verify properties of cP systems that contain terms with a nesting depth of 12 within 34 seconds. As the first cP system-specific framework, both the simulation and verification experiment results fulfil our expectations.

7.3 Comparison to related work

After P systems was proposed as a theoretical computational model, researchers considered multiple software approaches to implement, simulate, or verify P systems in real life. One of the earliest simulation works in P systems was conducted by Malița (2000) [106], here they used Prolog to simulate transition P systems (the implementation was named ProMem 0.1). Later, several P system simulation studies were published, this includes: (1) Balbontín et al. (2002) proposed a MzScheme implementation of transition P systems [107]; (2) Baranda et al. (2001) simulated transition P systems using Haskell [108], which was further extended by Arroyo et al. (2002) [109]; (3) Syropoulos et al. (2003) proposed a distributed Java simulation of transition P systems in the NOP_2(coo, tar) family [110]; (4) Nepomuceno-Chamorro (2004) proposed a Java simulator, which can simulate basic transition P systems with dissolution and priority rules [111]; (5) Cordón-Franco et al. (2004) proposed a Prolog simulator for deterministic P systems with active membranes [112]; and (6) Gutiérrez-Naranjo et al. (2005) implemented a simulator for confluent P systems [113].

In addition to the standard version of P systems, the simulation work of other P system variants such as tissue P systems, spiking neural P systems, and kernel P systems include: (1) Bianco and Castellini (2007) designed and implemented a simulation tool for metabolic P systems (MP systems), namely Psim [114]; (2) Castellini and Manca (2008) proposed a software architecture called Meta-Plab for MP systems [115]; (3) Pérez-Hurtado et al. (2010) proposed MeCoSim, which is a generic P system simulator that supports a P system DSL named P-Lingua [56, 57]; (4) Martínez-del-Amor et al. (2010) proposed a simulator for tissue P systems based on P-Lingua [116]; (5) Macías-Ramos et al. (2011) proposed a simulator for spiking neural P systems; (6) Buiu et al. (2011) developed a software tool named SNUPS for modelling and simulating numerical P systems [117], this was further refined by Arsene et al. (2011) [118]; (7) Pérez-Hurtado et al.

(2014) proposed a P-Lingua based simulator for tissue P systems with cell separation [119]; (8) Guo et al. (2019) implemented UPSimulator for cell-like, tissue-like and neural-like P systems [120], and (9) Konur et al. (2020) developed kPWorkbench, which is a simulation and verification framework of kernel P systems [58].

For P system formal verification, model checkers were widely used in previous studies. These include Omega [121], SPIN [121, 83, 85, 87, 122], NuSMV [122, 82, 123], ProB [84], UPPAAL [86], and PRISM [123]. For cP system verification, we used model checkers including PAT3 [8, 4, 6] and ProB [8, 4], and the Coq proof assistant [9] to verify several cP systems.

In addition to directly modelling and verifying P systems using third-party formal verifiers, kPWorkbench is an intergrated verification framework for Kernel P systems [123, 122, 58]. Two model checkers SPIN and NuSMV are integrated in the verification module of kPWorkbench, these can be used as back-end verifiers. The design of kPWorkbench enables the automated verification of Kernal P systems, however, developers are not able to modify or optimise the verification algorithms that are implemented in SPIN or NuSMV.

Implementation	P System Variant	Simulation	Verification	DSL	Comment
MeCoSim [56, 57]	P systems	✓	✗	P-Lingua	
MetaPlab [115]	MP systems	✓	✗	MP store	
UPSimulator [120]	P systems tP systems SN P systems	✓	✗	UPLanguage	
kPWorkbench [58]	kP systems	✓	✓	kP-Lingua	intergrated with SPIN and NuSMV
cPV (this work)	cP systems	✓	✓	cPVJ	built-in formal verifier

Figure 7.13: A comparison P system simulators and verifiers that are in development

Major P system software simulators or verifiers that are in development are listed in Fig. 7.13. cPV is not only the first P system verification framework that has its own verifier with corresponding algorithms, but also the only software framework which can effectively handle generic rules and compound terms. Existing P verification frameworks such as kPWorkbench use translators to transform P models into modelling languages of certain model checkers, where behaviours of complex P systems can only be partially modelled. In other words, the translation from cP syntax to third-party modelling languages is often incomplete. By supporting cPVJ, cPV can completely model cP systems and their behaviours. For cP system verification, since cPV has a built-in verifier, all the verification and reduction algorithms are tailored and optimised for cP systems. This significantly improves the running speed of cPV.

To verify a complex cP system using PAT3 or ProB, the system needs to be manually translated to CSP# or B, this is time consuming and error prone. To

Formal Tool	Verification Objects	Automated Translation	Automated Verification	Requirement of cP Systems
PAT3 [8, 4, 6]	a cP system instance	✗	✓	term nesting depth ≤ 2
ProB [8, 4]	a cP system instance	✗	✓	term nesting depth ≤ 2
Coq [9]	all the instances of a cP system	✗	✗	deterministic cP systems
cPV (this work)	a cP system instance	✓	✓	no special requirement

Figure 7.14: A comparison of different formal tools for verifying cP systems

prove the equivalence of the translated model and the orginal cP system is also a challenge. Using Coq to verify a cP system is even harder as the system needs to be manually modelled in Gallina, and the properties need to be specified and proven manualy. Nondeterministic properties of cP systems cannot be directly verified in Coq.

Compared to previous studies which used third-party tools to conduct cP system formal verification, cPV can verify all the cP systems without any special requirements. Both the translation from cP system to internal models and formal verification can be done automatically.

7.4 Summary

The case study and evaluation that were introduced in this section illustrate that it is feasible to model and simulate complex cP systems, and verify their properties in cPV. By choosing proper problem sizes, cPV can work efficiently on different cP systems that solve NPC problems.

Given a cP system represented in cPVJ, its system properties including deadlockfreeness, confluence, termination, determinism, and goal reachability can be effectively verified using cPV.

We built two datasets *D1* and *D2* for the evaluation of cPV, these can be used as benchmark datasets for cP-specific simulation and verification tools. *D1* and *D2* cover several aspects of complex cP systems including nondeterminism, max-parallelism, large numbers of terms, highly nested terms, and difficult unifications.

Membrane systems such as cP systems can perform computation much more efficiently than traditional computers, whose efficiency is often obtained by trading memory for time. Thus, to simulate or verify membrane systems in real life often requires a large amount of memory. We implemented several statespace traversing and reduction algorithms in cPV which significantly improved its running speed.

As the first cP system simulation and verification framework, cPV can effectively help researchers to study, design, and verify cP systems. Using cPV, domain

experts in membrane computing can verify cP systems without in-depth understanding of formal verification tools, techniques, and algorithms. All the model transformations are automatically handled by cPV, which also prevents incomplete translations from cP systems to other modelling languages. The design and implementation of cPV was inspired by several existing general purpose and domain specific formal tools including PAT3, ProB and Coq.

Chapter 8

Towards Automated Deduction in cP systems

Automated deduction examines how computer programs can help people prove formal theorems. Deduction in equational theory is fundamental in many research areas including automated theorem proving, formal verification, symbolic computation, and logic programming [124]. Emphasised in several studies, many interesting and important logics are built on top of equational logic, and all the computable functions and data structures can be defined in equational logic [125].

Introduced by Knuth-Bendix [60], equational theories can be proven by generating canonical rewrite systems. If a rewrite system is convergent, then using it to perform equational deduction is sound and complete.

In this chapter, we propose a cP system Π_d to perform automated deduction on equational logic [10]. Given a set of axioms, Π_d can compute all the critical pairs among the axioms in logarithmic time. To reduce a term of size m to a normal form, Π_d can be $\mathcal{O}(2^m)$ times faster than traditional rewrite systems. By slightly modifying the rules of Π_d, it can be extended to reason about equational theories with n-ary operators in the same time complexity.

To distinguish with terms and rules in equational theories, in the rest of the chapter we will add a "cP-" prefix to terms, rules, rulesets, states, and steps in cP systems. For example, in a cP system, a compound term $f(ab)$ will be called as a "cP-term", and the rule $s_1\ a(X) \to_+ s_1\ b(X)$ is a "cP-rule".

The chapter is organised in the following way. Section 8.1 introduces equational deduction. Section 8.2 presents the cP system encoding to the problem. Section 8.3 explains the cP-rulesets of Π_d. Section 8.4 includes a case study, to illustrate

how Π_d works. Section 8.5 provides a further discussion on Π_d, and Section 8.6 summarises the chapter.

8.1 Equational deduction

Let $\mathcal{F}$ and $\mathcal{V}$ be two disjoint sets, which include function symbols and variables. Suppose the collection of all terms is denoted by $\mathcal{T}$, and a *term* can either be a variable v, $v \in \mathcal{V}$ or an expression $f(t_1 t_2 \ldots t_k)$, $f \in \mathcal{F}$, $t_1, t_2, \ldots, t_k \in \mathcal{T}$. Furthermore, each function symbol has an *arity*, and nullary function symbols are also called *constants*. A term s is said to be a *subterm* of a term t if: (1) $s = t$, or (2) $t = f(t_1 t_2 \ldots t_k)$ and s is a subterm of t_i, $i \in [1\,..\,k]$. A subterm s of t is said to be *proper* iff (if and only if) $s \neq t$.

A *substitution* ϑ is a mapping from variables to terms. The application of ϑ to a term t is denoted by $t\vartheta$. The composition of two substitutions ϑ and τ is denoted by $\vartheta\tau$. For two terms s and t, if there exists a substitution ϑ that satisfies $s\vartheta = t\vartheta$, ϑ is called a *unifier* of s and t. If σ unifies s and t, and for any other unifier ω, there exists a unifier τ, such that $\omega = \sigma\tau$, we call σ an *mgu* of s and t. Several unification algorithms were proposed to compute an mgu of two terms, such as [68, 71, 72].

An *equation* is written as $s = t$, where $s, t \in \mathcal{T}$. Given a set of equations E, $E \vDash s = t$ denotes $s = t$ is true for every model in E. The *Rewrite relation* induced by E is denoted by $\rightarrow_E$. For instance, $s \rightarrow_E t$ indicates s rewrites to t. The *equational theory* induced by E is the symmetric-transitive-reflexive closure $\leftrightarrow_E^*$ of $\rightarrow_E$. Terms s and t are equivalent in E iff $s \leftrightarrow_E^* t$.

The following five properties are held by equations, these can be used to deduce new equations from old ones [59, 126]:

- *Reflexivity*: $s = s$.
- *symmetry*: if $s = t$ then $t = s$.
- *transitivity*: if $r = s$ and $s = t$ then $r = t$.
- *congruence*: if $s_i = t_i$ for $i \in [1\,..\,n]$ then $f(s_1, s_2, \ldots, s_n) = f(t_1, t_2, \ldots, t_n)$.
- *substitutivity*: if $s = t$, then $s\vartheta = t\vartheta$ for all substitutions ϑ.

Directed equations such as $s \rightarrow t$ are called *axioms* or *Rewriting rules*. A *Rewrite system* contains a set of axioms. The binary relation $\rightarrow$ is *confluent* or

Church-Rosser iff for all $s = t$, there exists a term r such that $s \to^* r$ and $t \to^* r$. Terminating confluent relations are also called *convergent* [127].

Terms are in *normal forms* or *irreducible forms* if they cannot be rewritten any further. A rewrite system $\mathcal{R}$ is said to be *Reduced* iff all the terms in its axioms are in normal forms (irreducible by $\mathcal{R}$).

Rewrite systems are one of the most effective tools to reason about equations. In a convergent rewrite system, an equation $s = t$ is valid iff s and t can be rewritten to an identical normal form. Proofs that consist of a chain of rewritings which gradually transforms an expression into another are called *equational proofs*.

A *completion procedure* aims to build a convergent rewrite system from a set of equations or axioms, which can also be seen as a process of simplifying equational proofs [91]. The complexity of an equational proof can be reduced by axioms in a corresponding convergent rewrite system, and the normal-form proof of the equational proof is called a *Rewrite proof*.

The classical Knuth-Bendix completion procedure was reformulated by Bachmair as *the standard completion* [127], this is shown in Fig. 8.1. E and R denote the set of equations and the set of axioms. The symbol $\succ$ refers to the reduction ordering, and $\triangleright$ is the encompassment ordering: $s \triangleright t$ means a subterm of s is an instance of t.

DELETE:	$(E \cup \{s = s\}; R) \vdash (E; R)$	
COMPOSE:	$(E; R \cup \{s \to t\}) \vdash (E; R \cup \{s \to u\})$	$if\ t \to_R u$
SIMPLIFY:	$(E \cup \{s = t\}; R) \vdash (E \cup \{s = u\}; R)$	$if\ t \to_R u$
ORIENT:	$(E \cup \{s = t\}; R) \vdash (E; R \cup \{s \to t\})$	$if\ s \succ t$
COLLAPSE:	$(E; R \cup \{t \to s\}) \vdash (E \cup \{u = s\}; R)$	$if\ l \to r \in R, t \to_{\{l \to r\}} u, t \triangleright l$
DEDUCE:	$(E; R) \vdash (E \cup \{s = t\}; R)$	$if\ s \leftarrow_R u \to_R t$

Figure 8.1: The standard completion

8.2 cP system encodings

Our cP system-based deduction approach works for general equational theories, the cP-rulesets will be slightly different for each theory. For illustration purposes, in this chapter we will focus on the well-known left group theory whose corresponding convergent set contains ten axioms.

The left group is closed under an associative binary operation $(+)$ with a left inverse $(-)$ and a left identity element e (Fig. 8.2). Lowercase letters x, y, and z

represent variables. The unary operator $(-)$ is assumed to have a higher precedence than $(+)$, so the expression $-x + -y$ has the same meaning as $(-x) + (-y)$.

(1) $e + x \to x$
(2) $-x + x \to e$
(3) $(x + y) + z \to x + (y + z)$

Figure 8.2: The left group theory

Given an axiom $s \to t$ (or an equation $s = t$), its terms s and t in $s \to t$ will be encoded in two different ways in cP systems. These two ways include tree forms and linear forms. Tree forms are used to process terms in a layer-by-layer manner, and linear forms are used to model axioms (rewriting rules) in equational theories. The two forms can be transformed between each other as needed.

8.2.1 The tree form of a term

To represent terms in a tree-like structure, two kinds of nodes (cP-terms) are needed, these are: internal nodes – *n(ID)(Content)*, and leaves – *l(ID)(Content)*. In the tree form of a term, internal nodes represent operators and leaves represent variables.

Each node has a content object and an ID which indicates its path from the root. For example, to represent $-x + x$ in tree form, two internal nodes and two leaves are needed, these are $n(a)(+)$, $n(a(a))(-)$, $l(b(a))(x)$, and $l(a(a(a)))(x)$ (Fig 8.3 (i)). The shape of a term's tree form is similar to the term's binary expression tree (Fig 8.3 (ii)). Given the term t, we use $tree(t)$ to denote its tree form in cP systems.

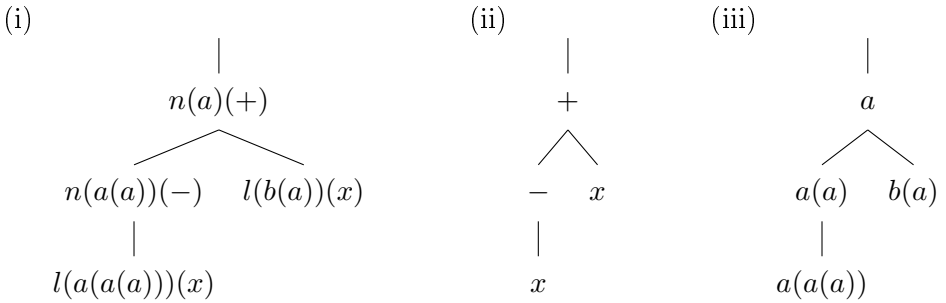

Figure 8.3 For $t = -x + x$, (i) is $tree(t)$, (ii) is the expression tree of t, (iii) shows the IDs in $tree(t)$

Actual contents in cP systems are cP-terms. Arcs that appear in Fig 8.3 (i) are only virtual, these can be indicated by matching node IDs using particular cP-rules.

The node IDs are represented as nested cP-terms (Fig. 8.3 (iii)). The root of a tree has an ID a. For each node with ID K, its first child has an ID $a(K)$ and its second child has an ID $b(K)$. For instance, a node or leaf with an ID $b(a(a))$ is the second child of the first child of the root.

A node stores its parent's ID, but not vice versa. cP-rule fragments such as $n(a(X))(\_) \to X$ or $l(b(X))(\_) \to X$ can be used to extract the parent ID of a child node.

8.2.2 The linear form of a term

In addition to tree forms, equational terms also need to be represented in a linear way in cP systems. Technically, the linear form of a term is one single leaf-like node containing the linear contents of $tree(t)$. By traversing $tree(t)$ in pre-order, we can obtain the linear form of t, denoted $linear(t)$.

The operator symbols $+$ and $-$ are atoms in cP systems, which can be used as cP-term functors (labels). For example, $+(x)(y)$ is a cP-term labelled $+$ which has two subterms x and y.

The linear form of a term – *l(ID)(LinearContent)* – is similar to a leaf node, but with linear content. For example, given the term $-x + (x + y)$, its linear form is $l(a)(+(-(x))(+(x)(y)))$. In the linear form of its subterm $-x$ is $l(a(a))(-(x))$, the ID $a(a)$ indicates that $-x$ is the first child (subterm) of the root $(-x+(x+y))$. Similarly, the linear form of $(x + y)$ is $l(b(a))(+(x)(y))$ – the ID $b(a)$ shows that $(x + y)$ is the second child of $-x + (x + y)$.

8.2.3 Transforming terms between tree and linear forms

Consider the term t and suppose its tree form – $tree(t)$ – is given. cP-rules *R1* and *R2* describe a bottom-up approach, which can transform $tree(t)$ into $linear(t)$.

$$
\begin{array}{llcll}
s_1 & l(a(K))(X),\ l(b(K))(Y),\ n(K)(+) & \to_+ & s_1 & l(K)(+(X)(Y)) \quad (R1) \\
s_1 & l(a(K))(X),\ n(K)(-) & \to_+ & s_1 & l(K)(-(X)) \quad (R2)
\end{array}
$$

Given $linear(t)$, cP-rules *R3* and *R4* describe a top-down approach, which can transform $linear(t)$ into $tree(t)$.

$$\begin{array}{llll} s_1 \;\; l(K)(+(X)(Y)) & \to_+ & s_1 \;\; n(K)(+),\, l(a(K))(X),\, l(b(K))(Y)) & (R3) \\ s_1 \;\; l(K)(-(X)) & \to_+ & s_1 \;\; n(K)(-),\, l(a(K))(X) & (R4) \end{array}$$

An example of transforming $t : -x + (x + y)$ between $tree(t)$ and $linear(t)$ is shown in Fig. 8.4. Starting from $tree(t)$, a cP system (at s_1) has two cP-rules *R1* and *R2*, and contains six system terms $n(a)(+)$, $n(a(a))(-)$, $l(a(a(a)))(x)$, $n(b(a))(+)$, $l(a(b(a)))(x)$, and $l(b(b(a)))(y)$. Since *R1* and *R2* commit to the same target state s_1, they can be applied in the same step.

In the first step, by applying *R1*, $n(b(a))(+)$, $l(a(b(a)))(x)$, and $l(b(b(a)))(y)$ will be consumed, and a new leaf $l(b(a))(+(x)(y))$ will be generated. By applying *R2*, $n(a(a))(-)$ and $l(a(a(a)))(x)$ will be consumed and $l(a(a))(-(x))$ will be produced.

In the second step, by applying *R1*, $n(a)(+)$, $l(a(a))(-(x))$, and $l(b(a))(+(x)(y))$ will be consumed and $l(a)(+(-(x))(+(x)(y)))$ will be produced. As a result neither *R1* nor *R2* is applicable and the system halts. The cP-term $l(a)(+(-(x))(+(x)(y)))$ left in the cP system is $linear(t)$.

Starting from $linear(t)$, a cP system (at s_1) has two cP-rules *R3* and *R4*, and contains a system term $l(a)(+(-(x))(+(x)(y)))$. In the first step, *R3* is applicable, this consumes $l(a)(+(-(x))(+(x)(y)))$ and produces three new cP-terms $n(a)(+)$, $l(a(a))(-(x))$, and $l(b(a))(+(x)(y))$.

When committing to the same target state, *R3* and *R4* can be applied in the same step. In the second step, *R3* will consume $l(b(a))(+(x)(y))$ and produce $n(b(a))(+)$, $l(a(b(a)))(x)$, and $l(b(b(a)))(y)$. Furthermore, *R4* will consume $l(a(a))(-(x))$ and produce $n(a(a))(-)$ and $l(a(a(a)))(x)$. So far neither *R3* or *R4* is applicable, thus the cP system halts and $tree(t)$ is obtained, which contains six cP-terms $n(a)(+)$, $n(a(a))(-)$, $l(a(a(a)))(x)$, $n(b(a))(+)$, $l(a(b(a)))(x)$, and $l(b(b(a)))(y)$.

Lemma 8.2.1. *Suppose a term t contains m symbols. If $tree(t)$ is balanced, then: using $R1 - R4$ to transform t between $tree(t)$ and $linear(t)$ takes $\mathcal{O}(\log m)$ steps.*

Proof. Let the depth of $tree(t)$ be d, since $tree(t)$ is balanced, d is $\mathcal{O}(\log m)$.

We use t' to denote the working tree, which includes the cP-terms during the transformations. The depth of t' is denoted by d'.

To transform $tree(t)$ into $linear(t)$, at the beginning, $t' = t$ and $d' = d$. When $d' > 1$, at least one of *R1* or *R2* is applicable. The leaves with the longest path

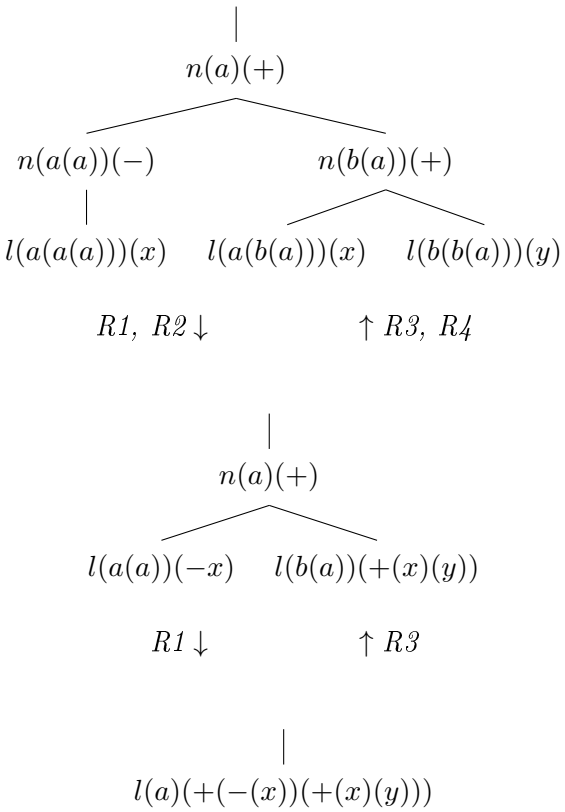

Figure 8.4 Transforming $-x + (x + y)$ between its tree and linear forms

(whose IDs are the deepest nested) in the tree must be consumed by *R1* or *R2*. In other words, when $d' > 1$, in each step, *R1* or *R2* will strictly decrease d' by 1. When $d' = 1$, neither *R1* or *R2* is applicable, the system halts and $linear(t)$ is obtained. Thus, to transform t from $tree(t)$ to $linear(t)$ takes $d - 1$ steps, which is $\mathcal{O}(\log m)$.

To transform $linear(t)$ into $tree(t)$, d' starts from 1. When $d' < d$, in each step, *R3* and *R4* will attempt to generate children for all the linear leaves in a maximally parallel manner, where d' is exactly increased by one. When $d' = d$, the transformation is finished and $tree(t)$ is obtained. Thus, to transform $linear(t)$ into $tree(t)$ also takes $d - 1$ steps, which is $\mathcal{O}(\log m)$. □

8.3 cP system rulesets for equational deduction

Given an equational theory, we propose a cP system Π_d, which can deduce new axioms from existing ones by performing completion. After obtaining a convergent

set of axioms, the cP system can prove equations by reducing their lhs and rhs terms to normal forms and comparing them.

Our cP completion procedure is similar to the standard completion shown in Fig. 8.1. While the rules COMPOSE, SIMPLIFY, and COLLAPSE are covered by a term reduction cP-ruleset, these need not be handled explicitly. If an equational theory can be represented as a convergent system, given a proper ordering (e.g. a weight function), the cP procedure will succeed. Several studies discussed how to choose weights for the Knuth-Bendix completion procedure [128, 96, 129].

In this study, we assume a well-founded ordering function called ORD(t_1,t_2) is given. Suppose ORD() accepts two cP-terms t_1 and t_2 as parameters. If t_1 has a larger weight than t_2, ORD(t_1,t_2) will return a positive number; otherwise it will return a negative number. For the left group theory, ORD() can be Knuth-Bendix order [60].

For illustration purposes, the cP-rulesets that appear in this section may be represented in a relatively simple way. To practically combine the cP-rules together, some of them may need to be slightly modified.

8.3.1 The superposition process

Given a set of axioms, by superposing an axiom onto itself or another axiom, a new term can be generated. By reducing the new term using different axioms or using one axiom in different ways a critical pair (new equation) can be obtained.

8.3.1.1 Computing compound subterms of a term

Terms containing at least one operator symbol are called *compound* terms. For example, $x + y$, $-z$, and $-x + -y$ are compound terms. A term that does not contain any operator symbol is called a *simple* term. For instance, x, y, and z are simple terms. Given a term $-x+(y+z)$, its compound subterms are $-x+(y+z)$, $-x$, and $y + z$; and its simple subterms are x, y, and z.

Given the linear form of a term t, *R5* and *R6* can generate subterms of t in a layer-by-layer manner. Compound subterms of t will be stored as cP-terms $s(\_)(\_)$, and simple subterms of t will be ignored.

$$
\begin{array}{llll}
s_2 \;\; l(K)(+(X)(Y)) & \rightarrow_+ & s_2 \;\; l(a(K))(X),\, l(b(K))(Y),\, s(K)(+(X)(Y)) & (R5) \\
s_2 \;\; l(K)(-(X)) & \rightarrow_+ & s_2 \;\; l(a(K))(X),\, s(K)(-(X)) & (R6)
\end{array}
$$

If the outermost operator symbol of a subterm (leaf) is $(+)$, *R5* will save the leaf $l(K)(+(X)(Y))$ as a compound subterm $s(K)(+(X)(Y))$, consume $l(K)(+(X)(Y))$, and produce its children (subterms) $l(a(K))(X)$ and $l(b(K))(Y)$ as new leaves.

Similarly, if the outermost operator symbol of a leaf is $(-)$, *R6* saves the leaf $l(K)(-(X))$ as a compound subterm $s(K)(-(X))$, consumes $l(K)(-(X))$, and produces its subterm $l(a(K))(X)$ as a new leaf.

After $d-1$ steps (d is the depth of $tree(t)$), all the compound subterms of t will be generated and saved. Leaves without any operator, such as $l(b(a))(e)$ or $l(a(a(a)))(x)$, will be ignored by *R5* and *R6*.

R5 and *R6* work in a destructive way, thus $linear(t)$ will be consumed after generating all the subterms of t. By using a set of cP-terms ($w(\_)$) to track the working nodes, *R5* and *R6* can be modified to *R7* and *R8* which are non-destructive.

$$s_2 \quad w(K) \quad \rightarrow_+ \quad s_2 \quad l(a(K))(X),\ l(b(K))(Y),\ s(K)(+(X)(Y)),\ w(a(K)),\ w(b(K)) \quad (R7)$$
$$\mid l(K)(+(X)(Y))$$
$$s_2 \quad w(K) \quad \rightarrow_+ \quad s_2 \quad l(a(K))(X),\ s(K)(-(X)),\ w(a(K)) \quad (R8)$$
$$\mid l(K)(-(X))$$

In having an initial working node $w(a)$, *R7* and *R8* use $l(K)(+(X)(Y))$ and $l(K)(-(X))$ as promoters – all the linear forms for t's subterms will be kept in the system. In every step, *R7* and *R8* check all the working nodes parallelly. If a working node with an ID K has operator symbols, *R7* and *R8* will store it as a compound subterm $s(K)(\_)$, consume $w(K)$, and add its children to working nodes (by generating $w(a(K))$, $w(b(K))$). If some working nodes are simple subterms of t, they will be ignored by *R7* and *R8*.

In addition to generating all the compound subterms of t from $linear(t)$, *R7* and *R8* will also generate linear forms of all the (compound and simple) subterms of t which can be used later. The complexity of *R7* and *R8* (running steps) is exactly the same to *R5* and *R6*.

Lemma 8.3.1. *Suppose a term t contains m symbols. Given $linear(t)$, if $tree(t)$ is balanced, then: using R7 and R8 to generate linear forms of all the subterms of t takes $\mathcal{O}(\log m)$ steps.*

Proof. Let the depth of $tree(t)$ be d, since $tree(t)$ is balanced, d is $\mathcal{O}(\log m)$.

If t is a simple term, then neither *R7* nor *R8* is applicable, and the cP system is terminated.

If t is a compound term, then at least one of *R7* or *R8* is applicable. In every step, *R7* or *R8* will store the compound subterms of t in one layer, and expand their children in the next layer. After $d-1$ steps, all the nodes in $tree(t)$ are generated, all the compound subterms of t are stored, and linear forms of all the subterms of t are generated.

Thus, to compute all the subterms of t takes $d-1$ steps, which is $\mathcal{O}(\log m)$. □

8.3.1.2 The unification process

Unification plays an essential role in the superposition process. Given a set of m axioms $s_i \to t_i$, $i = 1, 2, \ldots, m$, if s_j ($j \in [1\,..\,m]$) can be successfully unified with a compound subterm of s_k ($k \in [1\,..\,m]$), then an mgu σ can be found. By applying σ to s_k, a superposed term $s_k\sigma$ can be obtained. By using $s_j \to t_j$ and $s_k \to t_k$ to reduce $s_k\sigma$ in different ways, a critical pair will be generated. In the superposition process, $j = k$ is allowed, this means an axiom can superpose onto itself.

For an axiom $-x + (x + y) \to y$, its lhs term $-x + (x + y)$ can be successfully unified with its lhs subterm $x+y$ (an mgu: $\sigma = \{x \mapsto -x, y \mapsto x+y\}$). A new term can be obtained by applying σ to $(-x + (x + y))$, which is $--x + (-x + (x + y))$. By reducing the new term using the axiom in two different ways, we can get a critical pair $< x + y, --x + y >$.

To process or manipulate axioms in equational theories in cP systems, their variables such as x, y, z are actually encoded as atoms – only generic cP-rules may contain variables, and system terms must be ground. Thus, using cP systems to unify terms in equational theories is actually non-trivial.

For axioms $s_i \to t_i$ and $s_j \to t_j$, suppose s_{s_i} is a compound subterm of s_i, and all the m subterms of s_j are s_{s_j1}, s_{s_j2}, $\ldots s_{s_jm}$. Given $tree(s_{s_i})$, $tree(s_j)$, $linear(s_{s_j1})$, $linear(s_{s_j2})$,..., $linear(s_{s_jm})$ (linear forms of all the subterms of s_j can be obtained by *R7* and *R8*), suppose nodes in $tree(s_{s_i})$ are labelled as n_i and l_i; nodes in $tree(s_j)$ are labelled as n_j and l_j; nodes in $linear(s_{s_j1})$, $linear(s_{s_j2})$, $\ldots linear(s_{s_jm})$ are labelled as l'_j; the root ID of $tree(s_{s_i})$ is adjusted to a; and IDs of all the other nodes of $tree(s_{s_i})$ are also adjusted according to their parent-child relationships; cP-rules $R9 - R16$ can compute an mgu of s_{s_i} and s_j.

s_3	$w(K)$	$\rightarrow_+$	s_3	$w(a(K)),\ w(b(K)) \mid n_i(K)(+),\ n_j(K)(+)$	$(R9)$
s_3	$w(K)$	$\rightarrow_+$	s_3	$w(a(K)) \mid n_i(K)(-),\ n_j(K)(-)$	$(R10)$
s_3	$w(K)$	$\rightarrow_+$	s_3	$m(X)(Y) \mid\ l_i(K)(X),\ l'_j(K)(Y)$	$(R11)$
s_3		$\rightarrow_1$	s_4	$w(\_)$	$(R12)$
s_3		$\rightarrow_1$	s_5		$(R13)$
s_5	$m(X)(Y)$	$\rightarrow_+$	s_5	$\mid m(X)(Y)$	$(R14)$
s_5	$m(X)(Y),\ m(X)(Z)$	$\rightarrow_1$	s_4		$(R15)$
s_5		$\rightarrow_1$	s_6		$(R16)$

The working nodes are tracked by cP-terms $w(K)$. The initial working node is $w(a)$, which refers to the root ID of $tree(s_{s_i})$ and $tree(s_j)$. *R9* and *R10* compare the contents X and Y of the (internal) working nodes $n_i(K)(X)$ and $n_j(K)(Y)$. If they are the same ($X = Y = +$ or $X = Y = -$), *R9* or *R10* will consume $w(K)$, and set its children $w(a(K))$, $w(b(K))$ as new working nodes. If $X \neq Y$, neither *R9* or *R10* is applicable, and $w(K)$ will be kept in the cP system (s_{s_i} and s_j are not unifiable).

If a leaf $l_i(K)(X)$ is reached, *R11* will create a variable mapping $m(X)(Y)$ for X, by checking the linear form of the s_j's subterm with an ID K.

Since *R9*, *R10*, and *R11* commit to the same target state (which do not change the system state), they will be applied together as many times as possible, before the cP system considers *R12* and *R13*.

If neither *R9*, *R10*, or *R11* is applicable, the system will check *R12*, too see if there are $c(K)$ cP-terms left in the system. Unhandled $c(K)$ cP-terms indicate mismatches between s_{s_i} and s_j. If the cP system has at least one $c(K)$ cP-term, it will apply *R12* and change its state to s_4, which means the two terms are not unifiable. If *R12* is not applicable, the system will apply *R13* and change its state to s_5, this indicates all the variable mappings $m(X)(Y)$ are successfully generated.

In having all the variable mappings in the cP system, *R14* can be used to eliminate duplicated copies of the variable mappings. For instance, if the system contains three copies of $m(x)(+(y)(z))$, after applying *R14*, only one copy will be left in the system.

After applying *R14*, if there exists different mappings for the same variable such as $m(x)(+(y)(z))$ and $m(x)(-(w))$, *R15* will change the system state to s_4 to indicate the failure of the unification process. If *R15* is not applicable, *R16* will change the system state to s_6, which means the unification succeeded.

Lemma 8.3.2. *Suppose two terms t and t' contain m and m' symbols respectively. Given linear(t) and linear(t'), if tree(t) and tree(t') are balanced, then: using $R9 - R16$ to unify t with t' takes $\mathcal{O}(\log(\max(m, m')))$ steps.*

Proof. Applying lemma 8.2.1, to obtain $tree(t)$ and $tree(t')$ from $linear(t)$ and $linear(t')$ takes $\mathcal{O}(\log m) + \mathcal{O}(\log m')$ steps.

Applying lemma 8.3.1, to compute linear forms of all the subterms of t' takes $\mathcal{O}(\log m')$ steps.

Thus, the pre-computing of the unification takes $\mathcal{O}(\log m) + \mathcal{O}(\log m')$ steps, which is $\mathcal{O}(\log(\max(m, m')))$.

Suppose the depth of $tree(t)$ is d, which is $\mathcal{O}(\log m)$. In each step, *R9*, *R10* or *R11* will check exactly one layer of $tree(t)$, after d steps, none of them will be applicable, and *R12* or *R13* will be applied .

R12, *R13*, *R14*, *R15* and *R16* each take one step, after applying them, the system will terminate.

Summing up the discussion, to unify t with t', including the pre-computing, the cP-ruleset $(R9 - R16)$ takes $\mathcal{O}(\log(\max(m, m'))) + \mathcal{O}(\log m) + \mathcal{O}(1)$ steps, which is $\mathcal{O}(\log(\max(m, m')))$. □

8.3.2 Orientation and term reduction

To apply a unifier σ to a term t in cP systems is straightforward. Only leaves in $tree(t)$ contain variables, and *R17* can apply σ to all the leaves of $tree(t)$ in one step.

$$s_7 \quad l(K)(X) \quad \rightarrow_+ \quad s_7 \quad l(K)(Y) \mid m(X)(Y) \qquad (R17)$$

Suppose we have two axioms $s_1 \rightarrow t_1$ and $s_2 \rightarrow t_2$, let s_1' be a subterm of s_1. If σ unifies s_1' and s_2, then by applying σ to s_1, a superposed term $s_1\sigma$ can be obtained. Then we can get a critical pair p by reducing $s_1\sigma$ using $s_1 \rightarrow t_1$ and $s_2 \rightarrow t_2$.

For a critical pair $p =< c_1, c_2 >$, to make the rewrite system convergent, the equation $c_1 = c_2$ needs to be added into the system. Using the well-founded ordering function ORD(), we can orient $c_1 = c_2$ to an axiom $c_1 \rightarrow c_2$ (when ORD(c_1,c_2) > 0) or $c_2 \rightarrow c_1$ (when ORD(c_1,c_2) < 0).

Only non-trivial critical pairs need to be considered. If c_1 and c_2 are identical, the critical pair $< c_1, c_2 >$ is said to be *trivial*. Adding a trivial axiom such as $x \rightarrow x$ to a rewrite system will cause the system to be non-terminating – it is also unnecessary to rewrite a term by itself.

An optimisation of this approach is to reduce c_1 and c_2 to normal forms before considering them as a new equation. Suppose the normal form of c_1 and c_2 are

c'_1 and c'_2, if c'_1 and c'_2 are identical, we can simply delete the trivial critical pair $< c'_1, c'_2 >$ from the system. If c'_1 and c'_2 are different, for instance $c'_1 = x + e$ and $c'_2 = x$, we can use ORD() to orient them and add the new axiom $x + e \to x$ to the system. In having a new axiom, the system will also use it to reduce other existing axioms. When all the axioms are in normal forms, duplicate axioms can be deleted.

cP systems use lowercase letters as atoms and uppercase letters as variables, notably this can only be contained by rules. Thus, it is impossible to convert a lowercase letter to uppercase using cP-rules. As mentioned, to manipulate axioms in a theory, all their variables are encoded as atoms in cP systems, for example x, y, z. There is no direct way to use these axioms as cP-rules to reduce other terms. In addition to case converting, the current version of cP systems cannot generate new rules at run time. Thus, to use newly generated axioms as cP-rules (to reduce other terms), human intervention is needed.

Fig. 8.5 lists the convergent set of axioms that can be obtained from the left group theory. The cP-rules *R18* – *R27b* represent axioms (1) - (10). When none of the cP-rules in Fig. 8.5 are applicable, *R1* and *R2* can be used to fold the tree one level up. By looping a cP-ruleset consisting of *R18*, *R19*, ..., *R27b*, *R1*, and *R2*, a term can be reduced to a normal form. Compared to traditional rewrite systems, for a term t of size m, the cP-ruleset can simultaneously rewrite at most $\mathcal{O}(2^m)$ compound subterms of t in each step.

To detect the termination of the completion process in the cP system, we can use two counters $c_1(\_)$ and $c_2(\_)$. Every time a new axiom is added to the system we increase $c_1(\_)$ by one; and when a (duplicated) axiom is deleted from the system, we increase $c_2(\_)$ by one. By applying all the cP-rules, if neither $c_1(\_)$ nor $c_2(\_)$ is changed, then the completion process is terminated.

By having a convergent rewrite system, the correctness of an equation $t_1 = t_2$ can be proven or disproven by reducing t_1 and t_2 to normal forms and checking if they are identical.

(1)	$e + X$	$\rightarrow$	X		
s_8	$l(K)(+(e)(X))$	$\rightarrow_+$	s_8	$l(K)(X)$	$(R18)$
(2)	$-X + X$	$\rightarrow$	e		
s_8	$l(K)(+(-(X))(X))$	$\rightarrow_+$	s_8	$l(K)(e)$	$(R19)$
(3)	$(X + Y) + Z$	$\rightarrow$	$X + (Y + Z)$		
s_8	$l(K)(+(+(X)(Y))(Z))$	$\rightarrow_+$	s_8	$l(K)(+(X)(+(Y)(Z)))$	$(R20)$
s_8	$l(K)(+(X)(+(Y)(-(Y))))$	$\rightarrow_+$	s_8	$l(K)(X)$	$(R20a)$
s_8	$l(K)(+(X)(+(-(Y))(Y)))$	$\rightarrow_+$	s_8	$l(K)(X)$	$(R20b)$
(4)	$-X + (X + Y)$	$\rightarrow$	Y		
s_8	$l(K)(+(-(X))(+(X)(Y)))$	$\rightarrow_+$	s_8	$l(K)(Y)$	$(R21)$
(5)	$X + e$	$\rightarrow$	X		
s_8	$l(K)(+(X)(e))$	$\rightarrow_+$	s_8	$l(K)(X)$	$(R22)$
(6)	$-e$	$\rightarrow$	e		
s_8	$l(K)(-(e))$	$\rightarrow_+$	s_8	$l(K)(e)$	$(R23)$
(7)	$--X$	$\rightarrow$	X		
s_8	$l(K)(-(-(X)))$	$\rightarrow_+$	s_8	$l(K)(X)$	$(R24)$
(8)	$X + -X$	$\rightarrow$	e		
s_8	$l(K)(+(X)(-(X)))$	$\rightarrow_+$	s_8	$l(K)(e)$	$(R25)$
(9)	$X + (-X + Y)$	$\rightarrow$	Y		
s_8	$l(K)(+(X)(+(-(X))(Y)))$	$\rightarrow_+$	s_8	$l(K)(Y)$	$(R26)$
(10)	$-(X + Y)$	$\rightarrow$	$-Y + -X$		
s_8	$l(K)(-(+(X)(Y)))$	$\rightarrow_+$	s_8	$l(K)(+(-(Y))(-(X)))$	$(R27)$
s_8	$l(K)(+(-(-(X)))(Y))$	$\rightarrow_+$	s_8	$l(K)(+(X)(Y))$	$(R27a)$
s_8	$l(K)(+(X)(-(-(Y))))$	$\rightarrow_+$	s_8	$l(K)(+(X)(Y))$	$(R27b)$

Figure 8.5: The cP-rules to perform reduction on the left group theory

8.4 A case study

In this section, we use the left group theory as an example to demonstrate how to deduce new axioms from existing ones.

Suppose axioms in the left group theory are encoded as cP-terms labelled a. For instance, $a(2)(l(a)(+(-(x))(x)))(l(a)(e))$ represents the axiom $-x + x \rightarrow e$ with and ID 2, whereas $(l(a)(+(-(x))(x)))$ and $(l(a)(e))$ are its lhs and rhs. At the beginning of the computation, the initial three axioms exist in the cP system. The initial state of the cP system is s_1.

state: s_1
$a(1)(l(a)(+(e)(x)))(l(a)(x))$
$a(2)(l(a)(+(-(x))(x)))(l(a)(e))$
$a(3)(l(a)(+(+(x)(y))(z)))(l(a)(+(x)(+(y)(z))))$

The cP system needs to compute each axiom's compound subterms, here we use $a(3)(l(a)(+(+(x)(y))(z)))(l(a)(+(x)(+(y)(z))))$ as an example. The system will first apply $s_1\ a(X)(Y)(Z) \rightarrow_+ s_2\ a'(X)(Y)$ to make copies for all the axioms' lhs terms. In the same step, the rule $s_1\ \rightarrow_1 s_2\ w(a)$ can be applied to produce an initial working node $w(a)$.

state: s_2
$a(1)(l(a)(+(e)(x)))(l(a)(x))$
$a(2)(l(a)(+(-(x))(x)))(l(a)(e))$
$a(3)(l(a)(+(+(x)(y))(z)))(l(a)(+(x)(+(y)(z))))$
$a'(1)(l(a)(+(e)(x)))$
$a'(2)(l(a)(+(-(x))(x)))$
$a'(3)(l(a)(+(+(x)(y))(z)))$
$w(a)$

For readability, in the rest of the section, we will only show the cP-terms directly related to the example – axiom (3), other cP-terms in the system will be represented as dots (...).

R7': $s_2\ w(K) \rightarrow_+ s_2\ a'(A)(l(a(K))(X)),\ a'(A)(l(b(K))(Y)),\ s(A)(K)(+(X)(Y)),$
$w(a(K)),\ w(b(K)) \mid a'(A)(l(K)(+(X)(Y)))$
R8': $s_2\ w(K) \rightarrow_+ s_2\ a'(A)(l(a(K))(X)),\ s(A)(K)(-(X)),\ w(a(K))$
$\mid a'(A)(l(K)(-(X)))$
R_ n1: $s_2\ w(\_) \rightarrow_+ s_3$
R_ n2: $s_2 \rightarrow_1 s_3\ w(a)$

R7' and *R8'* are modified from *R7* and *R8*. By applying *R7'* once, $w(a)$ will be consumed and $a'(3)(l(a(a))(+(x)(y))))$, $a'(3)(l(b(a))(z))$, $s(3)(a)(+(+(x)(y))(z))$, $w(a(a))$, and $w(b(a))$ will be generated.

state: s_2
$a'(3)(l(a)(+(+(x)(y))(z)))$
$a'(3)(l(a(a))(+(x)(y))))$
$a'(3)(l(b(a))(z))$
$s(3)(a)(+(+(x)(y))(z))$
$w(a(a))\ w(b(a))$
...

In having the new working nodes $w(a(a))$ and $w(b(a))$, the system will apply *R7'* again, consume $w(a(a))$, and produce $a'(3)(l(a(a(a)))(x)))$, $a'(3)(l(b(a(a)))(y)))$, $s(3)(a(a))(+(x)(y))$, $w(a(a(a)))$, and $w(b(a(a)))$. Then neither *R7'* or *R8'* is applicable, thus *R_ n1* and *R_ n2* will be applied. Following this the system state will be changed to s_3, and the working node will be reset to $w(a)$.

state: s_3
$a'(3)(l(a)(+(+(x)(y))(z)))$
$a'(3)(l(a(a))(+(x)(y))))$
$a'(3)(l(b(a))(z))$
$a'(3)(l(a(a(a)))(x)))$
$a'(3)(l(b(a(a)))(y)))$
$s(3)(a)(+(+(x)(y))(z))$
$s(3)(a(a))(+(x)(y))$
$w(a)$
...

cP-terms $s(3)(a)(+(+(x)(y))(z))$ and $s(3)(a(a))(+(x)(y))$ are compound subterms of the lhs of axiom (3), in short, $(x+y)+z$ and $x+y$ are compound subterms of $(x + y) + z$.

In having the compound subterms of axiom (3), the system will unify them against all the axioms' lhs. Let us choose axiom (2) as an example – suppose the system is unifying t_s: $s(3)(a(a))(+(x)(y))$ against t_a: $a(2)(l(a)(+(-(x))(x)))$. The system will use $s_3\ a(X)(Y)(Z) \rightarrow_+ s_4\ a'(X)(Y)$ to make a copy of the lhs of axiom (2).

state: s_4
$a'(2)(l(a)(+(-(x))(x)))$
$s(3)(a(a))(+(x)(y))$
$w(a)$
...

For the compound subterm $s(3)(a(a))(+(x)(y))$, the system will adjust its path to a by applying $s_4\ s(X)(\_)(Y) \rightarrow_1 s_5\ s(X)(a)(Y)$. The old path $a(a)$ can be stored with the axiom ID as needed, in this example we will simply omit it.

state: s_5
$a'(2)(l(a)(+(-(x))(x)))$
$s(3)(a)(+(x)(y))$
$w(a)$
...

Linear forms of t_a's subterms are already generated in previous steps by *R7'* and *R8'*. The system can use a slightly modified version of *R3* and *R4* to generate tree forms of t_s and t_a.

state: s_5
$w(a)$

$a'(2)(l(a)(+(-(x))(x)))$
$n_a(a)(+) \quad l'_a(a)(+(-(x))(x))$
$n_a(a(a))(-) \quad l'_a(a(a))(-(x))$
$l_a(a(a(a)))(x) \quad l'_a(a(a(a)))(x)$
$l_a(b(a))(x) \quad l'_a(b(a))(x)$

$s(3)(a)(+(x)(y))$
$n_s(a)(+)$
$l_s(a(a))(x)$
$l_s(b(a))(y)$
...

Suppose cP-terms labelled n_s and l_s represent $tree(t_s)$; cP-terms labelled n_a and l_a represent $tree(t_a)$, and cP-terms labelled l'_a represent the linear forms of t_a's subterms. The following cP-ruleset can find an mgu of t_s and t_a.

R9': $s_5\ w(K) \rightarrow_+ s_5\ w(a(K)),\ c(b(K)) \mid n_s(K)(+),\ n_a(K)(+)$
R10': $s_5\ w(K) \rightarrow_+ s_5\ w(a(K)) \mid n_s(K)(-)\ n_a(K)(-)$
R11': $s_5\ w(K) \rightarrow_+ s_5\ m(X)(Y) \mid l_s(K)(X)\ l'_a(K)(Y)$
R12': $s_5 \rightarrow_1 s_6 \mid w(\_)$
R13': $s_5 \rightarrow_1 s_7$
R14': $s_7\ m(X)(Y) \rightarrow_+ s_7 \mid m(X)(Y)$
R15': $s_7\ m(X)(Y),\ m(X)(Z) \rightarrow_1 s_6$
R16': $s_7 \rightarrow_1 s_8$

After checking the promoters $n_s(a)(+)$ and $n_a(a)(+)$, the system will apply *R9'*, consume $w(a)$, and produce $w(a(a))$ and $w(b(a))$. For $w(a(a))$, *R11'* is applicable, and by checking $l_s(a(a))(x)$ and $l'_a(a(a))(-(x))$, a variable mapping $m(x)(-(x))$ will be generated and $w(a(a))$ will be consumed. For $w(b(a))$, *R11'* is also applicable, and by checking $l_s(b(a))(y)$ and $l'_a(b(a))(x)$ another mapping $m(y)(x)$ will be generated, and $w(b(a))$ will be consumed.

So far all the cP-terms labelled w in the system were consumed, and rules *R9'*, *R10'*, *R11'*, and *R12'* are not applicable. Thus, the system will apply *R13'* and change its state to s_7.

state: s_7
$a'(2)(l(a)(+(-(x))(x)))$
$s(3)(a)(+(x)(y))$
$m(x)(-(x))$
$m(y)(x)$
...

The system does not contain conflict or redundant mappings, thus *R14'* and *R15'* will not be applicable. *R16'* will thus change the system state to s_8.

The variable mappings for x and y are $m(x)(-(x))$ and $m(y)(x)$, which represent an mgu $\sigma = \{x \mapsto -x, y \mapsto x\}$.

By applying σ to $a'(3)(l(a)(+(+(x)(y))(z)))$, we can obtain a new cP-term $a''(3)(l(a)(+(+(-x)(x))(z)))$. The cP-rule $s_8\ a''(\_)(X) \rightarrow_1 s_8\ X$ can extract the linear representation $l(a)(+(+(-x)(x))(z))$, and a slightly modified version of *R3* and *R4* can transform it into a tree form. Following this the cP system can make a copy of the tree form – suppose nodes and leaves in the copy are labelled as n' and l'.

state: s_8
$n(a)(+)$
$n(a(a))(+)$
$l(b(a))(z)$
$n(a(a(a)))(-)$
$l(b(a(a)))(x)$
$l(a(a(a(a))))(x)$

$n'(a)(+)$
$n'(a(a))(+)$
$l'(b(a))(z)$
$n'(a(a(a)))(-)$
$l'(b(a(a)))(x)$
$l'(a(a(a(a))))(x)$
...

The cP-rules *R19'*, *R1'*, and *R2'* are used to process cP-terms with labels n and l; and cP-rules *R20'*,*R20a'*, *R20b'*, *R1"*, and *R2"* are used to process cP-terms with labels n' and l'. By using them to reduce the two copies of the superposed term $l(a)(+(+(-x)(x))(z))$ in different ways, a critical pair can be obtained.

R19': $s_8\ l(K)(+(-(X))(X)) \rightarrow_+ s_8\ l(K)(e)$
R1': $s_8\ l(a(K))(X)\ l(b(K))(Y)\ n(K)(+) \rightarrow_+ s_8\ l(K)(+(X)(Y))$
R2': $s_8\ l(a(K))(X)\ n(K)(-) \rightarrow_+ s_8\ l(K)(-(X))$
R20': $s_8\ l'(K)(+(+(X)(Y))(Z)) \rightarrow_+ s_8\ l'(K)(+(X)(+(Y)(Z)))$
R20a': $s_8\ l'(K)(+(X)(+(Y)(-(Y)))) \rightarrow_+ s_8\ l'(K)(X)$
R20b': $s_8\ l'(K)(+(X)(+(-(Y))(Y))) \rightarrow_+ s_8\ l'(K)(X)$
R1": $s_8\ l'(a(K))(X)\ l'(b(K))(Y)\ n'(K)(+) \rightarrow_+ s_8\ l'(K)(+(X)(Y))$
R2": $s_8\ l'(a(K))(X)\ n'(K)(-) \rightarrow_+ s_8\ l'(K)(-(X))$

Let us choose the application of *R19'*, *R1'*, and *R2'* as an example. At the beginning, cP-terms that match $l(K)(+(-(X))(X))$ cannot be found in the

system, so *R19'* is not applicable. Since *R1'* is also not applicable, the system will apply *R2'*, consume $l(a(a(a(a))))(x)$ and $n(a(a(a)))(-)$, and produce $l(a(a(a)))(-(x))$. Then *R1'* is applicable, $l(a(a(a)))(-(x))$, $l(b(a(a)))(x)$, and $n(a(a))(+)$ will be consumed, and $l(a(a))(+(-(x))(x))$ will be generated. In having $l(a(a))(+(-(x))(x))$, *R19'* is applicable and will produce $l(a(a))(e)$ and consume $l(a(a))(+(-(x))(x))$. Then *R1'* is applicable, thus the system will consume $l(a(a))(e)$, $l(b(a))(z)$, and $n(a)(+)$, and produce $l(a)(+(e)(z))$.

In addition to axiom (2) and axiom (3), the system will also use axiom (1) to reduce terms. Axiom (1) can be represented as *R18'* : $s_8\ l(K)(+(e)(X)) \rightarrow_+ s_8\ l(K)(X)$. By applying *R18'*, $l(a)(+(e)(z))$ will be reduced to $l(a)(z)$.

state: s_8
$l(a)(z)$
$l'(a)(+(-x)(+(x)(z)))$
...

Similarly, $l'(a)(+(+(-x)(x))(z))$ can be reduced to $l'(a)(+(-x)(+(x)(z)))$. Thus a critical pair z and $+(-x)(+(x)(z))$ will be generated by the cP system. By orienting them, we can get an new axiom $a(4)(+(-x)(+(x)(z)))(z)$, which refers to $-x + (x + z) \rightarrow z$. After axiom (4) being generated, it will be used to reduce other existing axioms.

By making use of unbounded computational resources in cP systems, we can modify the cP-rules to consider all the combinations of axioms and their subterms simultaneously. Using lhs terms of axioms and their subterms as promoters, they can be shared and will not be consumed. By setting up a proper ID for each cP-term, newly generated cP-terms can be well distinguished. For example, to unify $a'(2)(l(a)(+(-(x))(x)))$ and $s(3)(a(a))(+(x)(y))$, we can use their axiom IDs and paths to generate a unification ID $i(2)(a)(3)(a(a))$. By carrying this ID with the generated terms, their unifier will not be confused with unifiers generated by other pair of terms.

8.5 Discussion

The cP-rulesets of Π_d shown in the previous sections are designed for the left group theory. To handle other equational theories, some cP-rules need to be modified, while the general algorithm remains the same. If an equational theory can be represented as a convergent system, and it has a well-founded ordering, the

cP solution will succeed. This will mean all the new equations are orientable and only a finite number of critical pairs will be generated.

Given an arbitrary number of axioms, suppose the size of the largest axiom is m, here the cP system can compute all the critical pairs among the axioms in $\mathcal{O}(\log m)$ steps (applying lemmas 8.2.1, 8.3.1, and 8.3.2). To reduce a term t of size m' to a normal form, our cP system can simultaneously apply at most $\mathcal{O}(2^{m'})$ rewriting rules in one step. This is significantly faster than traditional rewrite systems (which apply rewriting rules sequentially).

The cP-rules for the left group theory can be easily extended to equational theories with n-ary operators without affecting the complexity of the superposition process. This is because the cP system processes (tree forms of) terms layer-by-layer (no matter how many subterms are in the same layer, they will be handled parallelly). For example, given a term t with an n-ary operator f, for each parent node with and ID K, its first child's ID is $c_1(K)$, its second child's ID is $c_2(K)$, and so on. The following two cP-rules can be used to transform t between $tree(t)$ and $linear(t)$.

Tree_to_linear n-ary:
$s_1\ l(c_1(K))(X_1),\ l(c_2(K))(X_2), \ldots, l(c_n(K))(X_n),\ n(K)(f)$
$\rightarrow_+ s_1\ l(K)(f(X_1)(X_2)\ldots(X_n))$
Linear_to_tree n-ary:
$s_1\ l(K)(f(X_1)(X_2)\ldots(X_n))$
$\rightarrow_+ s_1\ n(K)(f),\ l(c_1(K))(X_1),\ l(c_2(K))(X_2), \ldots, l(c_n(K))(X_n)$

Rewrite systems work well in many equational theories, while for non-trivial theorems in other logics they may encounter the combinatorial explosion. This is common for forward chaining methods. Thus, most of the practical theorem provers are interactive and require humans to work together with machines. In using an interactive theorem prover, a user can often reduce an intended problem into simpler subproblems and solve them separately. However, it is often non-trivial for a human, especially a non-expert user, to make proof guidance decisions.

Unlike traditional rewrite systems, cP systems can perform completion much more efficiently by using unbounded memory. Although finding efficient software or hardware implementations of P system variants such as cP systems is still a long-term challenge, our cP procedure can still be seen as an efficient parallel and distributed algorithm for equational deduction.

In addition to equational logic, the cP solution can also be used as a prototype to build automated deduction cP systems for other logics. For instance, by mod-

elling or introducing operators negation, conjunction, and universal quantification into cP systems, we can use them to model first-order logic.

Limitations of this work are regarding the grammar and design of the current version of cP systems. Atoms (lowercase letters) and variables (uppercase letters) cannot be converted to each other in cP systems, and there is also no straightforward way to add, remove, or modify cP-rules in a cP system at runtime. Thus, to fully execute the entire automated deduction procedure in cP systems is challenging. While in practical implementations, it is simple to perform case conversion in many modern programming languages.

8.6 Summary

Deduction in equational theory is fundamental in many research areas including automated theorem proving, formal verification, symbolic computation, and logic programming. Emphasised in several studies, many interesting and important logics are built on top of equational logic, and all the computable functions and data structures can be defined in equational logic.

In this chapter, we propose a cP system Π_d to perform equational deduction. Given a set of axioms, if each axiom contains at most m symbols, Π_d can find all the critical pairs among the axioms in $\mathcal{O}(\log m)$ steps. To reduce a term t of size m' to a normal form, Π_d can be exponentially faster than traditional rewrite systems.

Π_d can either be seen as a parallel or distributed algorithm for equational deduction, or it can be used as a prototype to design deduction cP systems for other logics. As discussed, by introducing meta cP systems, if a cP system can be encoded properly its properties can be potentially proven by other cP systems such as Π_d.

Chapter 9

Conclusion

The primary focus of this thesis is the formal verification of biologically inspired computing models, particularly cP systems. This chapter recalls the contributions of this thesis and looks to future directions.

9.1 Contributions

P systems are biologically inspired abstract computing models that are motivated by structures and functions of living cells. cP systems are a new variant of P systems which support complex symbols and generic rules. Rules in cP systems are applied following a weak priority order, this can simulate the if-then-else structures of traditional programming.

By using cP systems with a fixed constant number of generic rules, several computationally hard or real life problems were successfully solved in polynomial time. However, none of the previous studies mentioned how to validate or verify the proposed cP systems. This thesis discussed how to properly model, simulate, and verify cP systems using various formal tools. It also introduced a cP system-specific simulation and formal verification framework.

To model and simulate rule applications in cP systems, in Chapter 3, the unification problem of labelled multisets was formally defined, and an efficient unification algorithm, namely LNMU, was proposed. The well-formedness of labelled multiset unification problems was discussed and we proved that LNMU can solve well-formed labelled multiset unification problems in linear time.

In Chapter 4, multiple cP systems that solve NPC problems were proposed and verified via model checking. These include Π_{SSP} – a cP system that solves the subset sum problem in linear time, and Π_{Sudoku} – a cP system that solves the

general $m \times m$ Sudoku problem in sublinear time. Model checkers including PAT3 and ProB were used to verify the cP systems. In order to help model cP systems in PAT3 and ProB, two mapping guidelines were proposed. These guidelines can be used to transform cP system notation into modelling languages including CSP# and B. By analysing the performance and verification results, we discussed the advantages and disadvantages of the approach.

Chapter 5 verifies cP systems via another approach – interactive theorem proving. Different from model checking, which exhaustively traverses the statespace of a cP system, the theorem proving approach formalises certain specifications and proves that a cP system satisfies the specifications. An open source library was also designed and implemented, this can be used to model cP systems in the Coq proof assistant. We also provided a mapping guideline to help transform cP notation into Gallina. Multiple cP system solutions to NPC problems were verified via this approach. To verify a cP system, model checking and interactive theorem proving are two complementary formal verification approaches, which can be used together.

Chapter 6 introduced the implementation of multiple translators, and a formal framework for cP system simulation and verification, namely cPV. Following the mapping guidelines proposed in previous chapters, multiple translators were implemented and integrated with PAT3 and ProB. These can be used to verify certain ground cP systems automatically. However, only cP systems with ground rules can be verified via this approach. In order to handle cP systems which may have generic evolution rules, we implemented LNMU in an optimised way, and proposed cPV, which consists of a cP system simulator and a verifier. cPV supports a DSL named cPVJ, which can be used to describe different cP system models. System properties including deadlockfreeness, confluence, termination, determinism, and goal reachability can be automatically verified in cPV. Chapter 7 evaluates cPV from both functional and performance aspects. Two benchmark datasets were built and verified, these can also be used in future studies on cP system simulation and verification.

A new research direction for cP systems is presented in Chapter 8, which is "using cP systems as a tool to verify other computing models". We proposed a cP system Π_d, which can perform automated deduction on equational theories. Using a set of complex symbols, a term in equational theories can either be mapped to a tree form or a linear form. A set of transformation rules were proposed, which can transform a term that contains m symbols between tree and linear forms

in $\mathcal{O}(\log m)$ time. Using the cP system friendly term encodings and the power of maximal parallelism, Π_d can be exponentially faster than traditional rewrite systems when performing equational deduction.

9.2 Future work

This thesis has explored several research areas including multiset unification, formal verification of cP systems, and automated deduction. In addition to the aforementioned contributions, we also address several promising and interesting future directions, these are described as follows.

The interactive theorem proving approach has great potential in verifying cP systems. Compared to model checking, using proof assistants to verify a cP system does not requires exploration of the entire statespace. By making use of proving techniques such as mathematical induction, only a small amount of memory is required. However, human intervention is required in this approach. It will be promising to find an effective software or hardward implementation which can automatically model cP systems in certain languages, specify system properties as theorems, and prove such theorems.

In cP systems, after getting all the unifiers of a max-parallel rule, how to effectively select compatible unifier groups is an interesting topic. To simulate a cP system in cPV, in each rule application, only one group of compatible unifiers needs to be randomly selected. This can be easily done in polynomial time. However, for cP system verification, all the groups of compatible unifiers need to be obtained and verified. It is not hard to prove that finding all the compatible unifier groups is a NPC problem. Like many other NPC problems, it is possible to apply certain techniques such as dynamic programming (which trades space for time) to improve the running time performance of cPV.

Inspired by metaprogramming, an interesting idea to consider is the designing of a "meta cP system" which can process other cP systems as input data (objects). By having a proper "cP system rule template", a meta cP system can run, simulate, and even verify other cP systems. It is also possible to use meta cP systems to dynamically insert or remove rules from another cP system, this will significantly increase the power of cP systems. By modifying deduction cP systems such as Π_d into meta cP systems, it is possible to use them to verify other particular cP systems.

Appendix A

Transforming ground cP systems into cP-Coq models

```
1  def cPtocPCoq(str_ruleset, system_terms, system_state, system_name):
2    atoms = set()
3    ruleset = []
4    for str_rule in str_ruleset:
5      rule = ParseRule(str_rule)
6      ruleset.append(rule)
7      for a1 in rule.LHS():
8        atoms.add(a1)
9      for a2 in rule.RHS():
10       atoms.add(a2)
11   cPCoq_file = '(*' + system_name + '*)\n'
12   cPCoq_file += 'From CP Require Export operations.\n'
13   cPCoq_file += 'From Coq Require Import Lists.List.\n'
14   cPCoq_file += 'Import ListNotations.\n\n'
15   cPCoq_file +=
16     'Definition cPsys1 := cP_sys (s ' + system_state[1:] + ') ['
17   has_sys_term = False
18   for t1 in system_terms:
19     multi = system_terms[t1]
20     for i in range(multi):
21       if has_sys_term:
22         cPCoq_file += '; Atom ' + t1
23       else:
24         has_sys_term = True
25         cPCoq_file += 'Atom ' + t1
26   cPCoq_file += '].\n\n'
27   i = 1
```

```
28    for rule in ruleset:
29      cPCoq_file += 'Definition r' + str(i) +
30        ' (sys:cPsystem_conf): cPsystem_conf :=\n'
31      cPCoq_file += 'match sys with\n'
32      cPCoq_file += '| cP_sys (s 1) terms =>\n'
33      lhs_terms = '['
34      has_lhs_term = False
35      for t2 in rule.LHS():
36        multi2 = rule.LHS()[t2]
37      for i in range(multi2):
38        if has_lhs_term:
39          lhs_terms += '; Atom ' + t2
40        else:
41          has_lhs_term = True
42          lhs_terms += 'Atom ' + t2
43      lhs_terms += ']'
44      new_sys_terms = 'sys'
45      for t3 in rule.LHS():
46        multi3 = rule.LHS()[t3]
47        for i in range(multi3):
48          new_sys_terms =
49            '(ConsumeATerm (Atom ' + t3 + ') ' + new_sys_terms + ')'
50      for t4 in rule.RHS():
51        multi4 = rule.RHS()[t4]
52        for i in range(multi4):
53          new_sys_terms =
54            '(ProduceATerm (Atom ' + t4 + ') ' + new_sys_terms + ')'
55      cPCoq_file +=
56        'if AtomBagIn ' + lhs_terms + ' terms then ChangeState
57        (s ' + rule.RState()[1:] + ') ' + new_sys_terms + '\n'
58      cPCoq_file += 'else sys\n'
59      cPCoq_file += '| _ => sys\n'
60      cPCoq_file += 'end.\n'
61    return cPCoq_file
```

Bibliography

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

[2] G. Paun, *Membrane computing: an introduction*. Springer Science & Business Media, 2002.

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

[4] Y. Liu, R. Nicolescu, and J. Sun, "Formal verification of cP systems using PAT3 and ProB," *Journal of Membrane Computing*, vol. 2, pp. 84–90, 2020.

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

[6] Y. Liu, R. Nicolescu, J. Sun, and A. Henderson, "A sublinear Sudoku solution in cP Systems and its formal verification," *Computer Science Journal of Moldova*, vol. 85, no. 1, pp. 3–28, 2021.

[7] Y. Liu, R. Nicolescu, and J. Sun, "An efficient labelled nested multiset unification algorithm," *Journal of Membrane Computing*, vol. 3, no. 3, pp. 194–204, 2021.

[8] Y. Liu, R. Nicolescu, and J. Sun, "Formal approach to cP system verification," in *The 8th Asian Conference on Membrane Computing (ACMC2019)*, p. 232, 2019.

[9] Y. Liu, R. Nicolescu, and J. Sun, "Formal verification of cP systems using Coq," *Journal of Membrane Computing*, vol. 3, no. 3, pp. 205–220, 2021.

[10] Y. Liu, R. Nicolescu, and J. Sun, "Towards automated deduction in cP systems," *Information Sciences*, vol. 587, pp. 435–449, 2022.

[11] R. Nicolescu, "Parallel and distributed algorithms in P systems," in *International Conference on Membrane Computing*, pp. 35–50, Springer, 2011.

[12] R. Nicolescu and H. Wu, "Complex objects for complex applications," *Romanian Journal of Information Science and Technology*, vol. 17, no. 1, pp. 46–62, 2014.

[13] R. Nicolescu, "Parallel thinning with complex objects and actors," in *International Conference on Membrane Computing*, pp. 330–354, Springer, 2014.

[14] R. Nicolescu, F. Ipate, and H. Wu, "Programming P systems with complex objects," in *International conference on membrane computing*, pp. 280–300, Springer, 2013.

[15] A. Henderson and R. Nicolescu, "Actor-like cP systems," in *International conference on membrane computing*, pp. 160–187, Springer, 2018.

[16] R. Nicolescu, "Structured grid algorithms modelled with complex objects," in *International Conference on Membrane Computing*, pp. 321–337, Springer, 2015.

[17] R. Nicolescu, "Revising the membrane computing model for Byzantine agreement," in *International Conference on Membrane Computing*, pp. 317–339, Springer, 2016.

[18] R. Nicolescu, "Most common words-a cP systems solution," in *International Conference on Membrane Computing*, pp. 214–229, Springer, 2017.

[19] A. Henderson, R. Nicolescu, and M. J. Dinneen, "Solving a PSPACE-complete problem with cP systems," *Journal of Membrane Computing*, vol. 2, no. 4, pp. 311–322, 2020.

[20] R. Nicolescu, M. J. Dinneen, J. Cooper, A. Henderson, and Y. Liu, "Logarithmic SAT Solution with Membrane Computing," *Axioms*, vol. 11, no. 2, p. 66, 2022.

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

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

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

[24] G. Păun, "Introduction to membrane computing," in *Applications of Membrane Computing*, pp. 1–42, Springer, 2006.

[25] G. Păun and G. Rozenberg, "A guide to membrane computing," *Theoretical Computer Science*, vol. 287, no. 1, pp. 73–100, 2002.

[26] E. M. Clarke, "Model checking," in *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pp. 54–56, Springer, 1997.

[27] M. Müller-Olm, D. Schmidt, and B. Steffen, "Model-checking," in *International Static Analysis Symposium*, pp. 330–354, Springer, 1999.

[28] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pp. 46–57, ieee, 1977.

[29] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Workshop on logic of programs*, pp. 52–71, Springer, 1981.

[30] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker BLAST," *International Journal on Software Tools for Technology Transfer*, vol. 9, no. 5, pp. 505–525, 2007.

[31] D. Beyer and M. E. Keremoglu, "CPAchecker: A tool for configurable software verification," in *International Conference on Computer Aided Verification*, pp. 184–190, Springer, 2011.

[32] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "NuSMV: a new symbolic model checker," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 410–425, 2000.

[33] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "Nusmv 2: An opensource tool for symbolic model checking," in *International conference on computer aided verification*, pp. 359–364, Springer, 2002.

[34] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM: Probabilistic symbolic model checker," in *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pp. 200–204, Springer, 2002.

[35] G. J. Holzmann, "The model checker SPIN," *IEEE Transactions on software engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[36] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi, "UPPAAL-a tool suite for automatic verification of real-time systems," in *International hybrid systems workshop*, pp. 232–243, Springer, 1995.

[37] Y. Liu, J. Sun, and J. S. Dong, "Pat 3: An extensible architecture for building multi-domain model checkers," in *2011 IEEE 22nd international symposium on software reliability engineering*, pp. 190–199, IEEE, 2011.

[38] M. Leuschel and M. Butler, "ProB: A model checker for B," in *International symposium of formal methods europe*, pp. 855–874, Springer, 2003.

[39] M. Leuschel and M. Butler, "ProB: an automated analysis toolset for the B method," *International Journal on Software Tools for Technology Transfer*, vol. 10, no. 2, pp. 185–203, 2008.

[40] J. Sun, Y. Liu, and J. S. Dong, "Model checking CSP revisited: introducing a process analysis toolkit," in *International symposium on leveraging applications of formal methods, verification and validation*, pp. 307–322, Springer, 2008.

[41] J.-R. Abrial and A. Hoare, *The B-book: assigning programs to meanings*, vol. 1. Cambridge university press Cambridge, 1996.

[42] R. S. Boyer and J. S. Moore, "A theorem prover for a computational logic," in *International Conference on Automated Deduction*, pp. 1–15, Springer, 1990.

[43] U. Norell, *Towards a practical programming language based on dependent type theory*, vol. 32. Citeseer, 2007.

[44] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang, "Secure distributed programming with value-dependent types," *ACM SIGPLAN Notices*, vol. 46, no. 9, pp. 266–278, 2011.

[45] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, *et al.*, "Dependent types and multi-monadic effects in F," in *Proceedings of the 43rd annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 256–270, 2016.

[46] M. J. Gordon, "HOL: A proof generating system for higher-order logic," in *VLSI specification, verification and synthesis*, pp. 73–128, Springer, 1988.

[47] L. Paulson, *Isabelle: A generic theorem prover*, vol. 828. Springer Science & Business Media, 1994.

[48] A. Trybulec and H. A. Blair, "Computer assisted reasoning with Mizar," in *IJCAI*, vol. 85, pp. 26–28, Citeseer, 1985.

[49] R. Matuszewski and P. Rudnicki, "Mizar: the first 30 years," *Mechanized mathematics and its applications*, vol. 4, no. 1, pp. 3–24, 2005.

[50] B. Barras, S. Boutin, C. Cornes, J. Courant, J.-C. Filliatre, E. Gimenez, H. Herbelin, G. Huet, C. Munoz, C. Murthy, *et al.*, *The Coq proof assistant reference manual: Version 6.1*. PhD thesis, Inria, 1997.

[51] G. Huet, G. Kahn, and C. Paulin-Mohring, "The Coq proof assistant a tutorial," *Rapport Technique*, vol. 178, 1997.

[52] D. Kästner, J. Barrho, U. Wünsche, M. Schlickling, B. Schommer, M. Schmidt, C. Ferdinand, X. Leroy, and S. Blazy, "CompCert: Practical experience on integrating and qualifying a formally verified optimizing compiler," in *ERTS2 2018-9th European Congress Embedded Real-Time Software and Systems*, pp. 1–9, 2018.

[53] S. Conchon and J.-C. Filliâtre, "A persistent union-find data structure," in *Proceedings of the 2007 workshop on Workshop on ML*, pp. 37–46, 2007.

[54] G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. L. Roux, A. Mahboubi, R. O'Connor, S. Ould Biha, *et al.*, "A machine-checked proof of the odd order theorem," in *International conference on interactive theorem proving*, pp. 163–179, Springer, 2013.

[55] G. Gonthier *et al.*, "Formal proof-the four-color theorem," *Notices of the AMS*, vol. 55, no. 11, pp. 1382–1393, 2008.

[56] D. Díaz-Pernil, I. Pérez-Hurtado, M. J. Pérez-Jiménez, and A. Riscos-Núnez, "A P-Lingua programming environment for membrane computing," in *International workshop on membrane computing*, pp. 187–203, Springer, 2008.

[57] M. García-Quismondo, R. Gutiérrez-Escudero, I. Pérez-Hurtado, M. J. Pérez-Jiménez, and A. Riscos-Núñez, "An overview of P-Lingua 2.0," in *International Workshop on Membrane Computing*, pp. 264–288, Springer, 2009.

[58] S. Konur, L. Mierlă, F. Ipate, and M. Gheorghe, "kPWorkbench: A software suit for membrane systems," *SoftwareX*, vol. 11, p. 100407, 2020.

[59] J. Goguen, "Theorem proving and algebra," *arXiv preprint arXiv:2101.02690*, 2021.

[60] D. Knuth and P. Bendix, "Simple word problems in universal algebras," in *Automation of Reasoning*, pp. 342–376, Springer, 1983.

[61] W. Baader and F. Snyder, "Unification theory," *Handbook of automated reasoning*, vol. 1, pp. 447–533, 2001.

[62] H. Yasuura, "On parallel computational complexity of unification," in *Unknown Host Publication Title*, pp. 235–243, Ohmsha Ltd, 1984.

[63] D. Kapur and P. Narendran, "NP-completeness of the set unification and matching problems," in *International Conference on Automated Deduction*, pp. 489–495, Springer, 1986.

[64] H. Mannila and E. Ukkonen, "On the complexity of unification sequences," in *International Conference on Logic Programming*, pp. 122–133, Springer, 1986.

[65] D. Benanav, D. Kapur, and P. Narendran, "Complexity of matching problems," *Journal of symbolic computation*, vol. 3, no. 1-2, pp. 203–216, 1987.

[66] A. Itai and J. A. Makowsky, "Unification as a complexity measure for logic programming," *The Journal of Logic Programming*, vol. 4, no. 2, pp. 105–117, 1987.

[67] D. Kapur and P. Narendran, "Complexity of unification problems with associative-commutative operators," *Journal of Automated Reasoning*, vol. 9, no. 2, pp. 261–288, 1992.

[68] J. A. Robinson, "A machine-oriented logic based on the resolution principle," *Journal of the ACM (JACM)*, vol. 12, no. 1, pp. 23–41, 1965.

[69] J. A. Robinson, "Computational logic: The unification computation," *Machine intelligence*, vol. 6, pp. 63–72, 1971.

[70] M. V. Zilli, "Complexity of the unification algorithm for first-order expressions," *Calcolo*, vol. 12, no. 4, pp. 361–371, 1975.

[71] M. Paterson and M. Wegman, "Linear unification," in *Proceedings of the eighth annual ACM symposium on Theory of computing*, pp. 181–186, 1976.

[72] A. Martelli and U. Montanari, "An efficient unification algorithm," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, no. 2, pp. 258–282, 1982.

[73] A. Dovier, A. Policriti, and G. Rossi, "A uniform axiomatic view of lists, multisets, and sets, and the relevant unification algorithms," *Fundamenta Informaticae*, vol. 36, no. 2, 3, pp. 201–234, 1998.

[74] E. Dantsin and A. Voronkov, "A nondeterministic polynomial-time unification algorithm for bags, sets and trees," in *International Conference on Foundations of Software Science and Computation Structure*, pp. 180–196, Springer, 1999.

[75] J. Jaffar, "Efficient unification over infinite terms," *New Generation Computing*, vol. 2, no. 3, pp. 207–219, 1984.

[76] C. Dwork, P. C. Kanellakis, and J. C. Mitchell, "On the sequential nature of unification," *The Journal of Logic Programming*, vol. 1, no. 1, pp. 35–50, 1984.

[77] D. E. Rydeheard and R. M. Burstall, "A categorical unification algorithm," in *Category Theory and Computer Programming*, pp. 493–505, Springer, 1986.

[78] G. Huet, *Résolution d'équations dans des langages d'ordre 1, 2, ... ω*. PhD thesis, PhD thesis, Université Paris VII, 1976.

[79] M. d. J. Pérez Jiménez and F. Sancho Caparrini, "Verifying a P system generating squares," *Romanian Journal of Information Science and Technology (ROMJIST), 5 (1-2), 181-191.*, 2002.

[80] P. Kefalas, G. Eleftherakis, M. Holcombe, and M. Gheorghe, "Simulation and verification of P systems through communicating X-machines," *BioSystems*, vol. 70, no. 2, pp. 135–148, 2003.

[81] O. Andrei, G. Ciobanu, and D. Lucanu, "Executable specifications of P systems," in *International Workshop on Membrane Computing*, pp. 126–145, Springer, 2004.

[82] F. Ipate, M. Gheorghe, and R. Lefticaru, "Test generation from P systems using model checking," *The Journal of Logic and Algebraic Programming*, vol. 79, no. 6, pp. 350–362, 2010.

[83] F. Ipate, R. Lefticaru, and C. Tudose, "Formal verification of P systems using SPIN," *International Journal of Foundations of Computer Science*, vol. 22, no. 01, pp. 133–142, 2011.

[84] F. Ipate and A. Turcanu, "Modeling, verification and testing of P systems using Rodin and ProB," *Proceedings of the Ninth Brainstorming Week on Membrane Computing, 209-219. Sevilla, ETS de Ingeniería Informática, 31 de enero-4 de febrero, 2011*, 2011.

[85] R. Lefticaru, C. Tudose, and F. Ipate, "Towards automated verification of P systems using SPIN," *International Journal of Natural Computing Research (IJNCR)*, vol. 2, no. 3, pp. 1–12, 2011.

[86] B. Aman and G. Ciobanu, "Modelling and verification of weighted spiking neural systems," *Theoretical Computer Science*, vol. 623, pp. 92–102, 2016.

[87] M. Gheorghe, F. Ipate, R. Lefticaru, M. J. Pérez-Jiménez, A. Ţurcanu, L. Valencia Cabrera, M. García-Quismondo, and L. Mierlă, "3-Col problem modelling using simple kernel P systems," *International Journal of Computer Mathematics*, vol. 90, no. 4, pp. 816–830, 2013.

[88] R. Lefticaru, M. Gheorghe, S. Konur, I. M. Niculescu, and H. N. Adorna, "Spiking Neural P Systems Simulation and Verification," in *18th International Conference on High Performance Computing and Simulation (HPCS)*, (Barcelona, Spain), IEEE, 2021.

[89] A. Bundy, "A survey of automated deduction," in *Artificial intelligence today*, pp. 153–174, Springer, 1999.

[90] G. Huet and D. Oppen, "Equations and rewrite rules: a survey," *Formal Language Theory*, pp. 349–405, 1980.

[91] J. Hsiang, H. Kirchner, P. Lescanne, and M. Rusinowitch, "The term rewriting approach to automated theorem proving," *The Journal of Logic Programming*, vol. 14, no. 1-2, pp. 71–99, 1992.

[92] L. Bachmair and H. Ganzinger, "Rewrite-based equational theorem proving with selection and simplification," *Journal of Logic and Computation*, vol. 4, no. 3, pp. 217–247, 1994.

[93] M. P. Bonacina, "A taxonomy of theorem-proving strategies," in *Artificial Intelligence Today*, pp. 43–84, Springer, 1999.

[94] G. Huet, "Confluent reductions: Abstract properties and applications to term rewriting systems," *Journal of the ACM (JACM)*, vol. 27, no. 4, pp. 797–821, 1980.

[95] G. Peterson and M. Stickel, "Complete sets of reductions for some equational theories," *Journal of the ACM (JACM)*, vol. 28, no. 2, pp. 233–264, 1981.

[96] L. Bachmair, N. Dershowitz, and D. A. Plaisted, "Completion without failure," in *Rewriting Techniques*, pp. 1–30, Elsevier, 1989.

[97] J. Hsiang, "Refutational theorem proving using term-rewriting systems," *Artificial Intelligence*, vol. 25, no. 3, pp. 255–300, 1985.

[98] J. Hsiang, "Rewrite method for theorem proving in first order theory with equality," *Journal of Symbolic Computation*, vol. 3, no. 1-2, pp. 133–151, 1987.

[99] M. Kurihara and H. Kondo, "Completion for multiple reduction orderings," *Journal of Automated Reasoning*, vol. 23, no. 1, pp. 25–42, 1999.

[100] I. Wehrman, A. Stump, and E. Westbrook, "Slothrop: Knuth-Bendix completion with a modern termination checker," in *International Conference on Rewriting Techniques and Applications*, pp. 287–296, Springer, 2006.

[101] H. Sato, S. Winkler, M. Kurihara, and A. Middeldorp, "Multi-completion with termination tools (system description)," in *International Joint Conference on Automated Reasoning*, pp. 306–312, Springer, 2008.

[102] S. Winkler, H. Sato, A. Middeldorp, and M. Kurihara, "Multi-completion with termination tools," *Journal of Automated Reasoning*, vol. 50, no. 3, pp. 317–354, 2013.

[103] Y. Liu, R. Nicolescu, and J. Sun, "Multiset unification and cP system simulation," in *The International Conference on Membrane Computing 2020*, (Vienna, Austria), TU Wien, 2020.

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

[105] Y.-B. Kim, *Distributed algorithms in membrane systems*. PhD thesis, ResearchSpace@ Auckland, 2012.

[106] M. Maliţa, "Membrane computing in Prolog," in *Pre-Proceedings of The Workshop on Multiset Processing (WMP-CdeA 2000)*, p. 8, 2000.

[107] D. Balbontín Noval, M. J. Pérez-Jiménez, and F. Sancho Caparrini, "A MzScheme implementation of transition P systems," in *Workshop on Membrane Computing*, pp. 58–73, Springer, 2002.

[108] A. V. Baranda, F. Arroyo, J. Castellanos, and R. Gonzalo, "Towards an electronic implementation of membrane computing: a formal description of non-deterministic evolution in transition P systems," in *International Workshop on DNA-Based Computers*, pp. 350–359, Springer, 2001.

[109] F. Arroyo, C. Luengo, A. V. Baranda, and L. d. Mingo, "A software simulation of transition P systems in Haskell," in *Workshop on Membrane Computing*, pp. 19–32, Springer, 2002.

[110] A. Syropoulos, E. G. Mamatas, P. C. Allilomes, and K. T. Sotiriades, "A distributed simulation of transition P systems," in *International Workshop on Membrane Computing*, pp. 357–368, Springer, 2003.

[111] I. A. Nepomuceno-Chamorro, "A Java simulator for membrane computing," *J. Univers. Comput. Sci.*, vol. 10, no. 5, pp. 620–629, 2004.

[112] A. Cordón-Franco, M. A. Gutiérrez-Naranjo, M. J. Pérez-Jiménez, and F. Sancho-Caparrini, "A Prolog simulator for deterministic P systems with active membranes," *New Generation Computing*, vol. 22, no. 4, pp. 349–363, 2004.

[113] M. Á. Gutiérrez-Naranjo, M. d. J. Pérez-Jiménez, and A. Riscos-Núñez, "A simulator for confluent P systems," *Proceedings of the Third Brainstorming Week on Membrane Computing, 169-184. Sevilla, ETS de Ingeniería Informática, 31 de Enero-4 de Febrero, 2005,*, 2005.

[114] L. Bianco and A. Castellini, "Psim: a computational platform for Metabolic P systems," in *International Workshop on Membrane Computing*, pp. 1–20, Springer, 2007.

[115] A. Castellini and V. Manca, "MetaPlab: A computational framework for metabolic P systems," in *International Workshop on Membrane Computing*, pp. 157–168, Springer, 2008.

[116] M. A. Martínez-del Amor, I. Pérez-Hurtado, M. J. Pérez-Jiménez, and A. Riscos-Núñez, "A P-Lingua based simulator for tissue P systems," *The Journal of Logic and Algebraic Programming*, vol. 79, no. 6, pp. 374–382, 2010.

[117] C. Buiu, O. Arsene, C. Cipu, and M. Patrascu, "A software tool for modeling and simulation of numerical P systems," *BioSystems*, vol. 103, no. 3, pp. 442–447, 2011.

[118] O. Arsene, C. Buiu, and N. Popescu, "SNUPS-a simulator for numerical membrane computing," *International Journal of Innovative Computing, Information and Control*, vol. 7, no. 6, pp. 3509–3522, 2011.

[119] I. Pérez-Hurtado, L. Valencia-Cabrera, J. M. Chacón, A. Riscos-Núnez, and M. J. Pérez-Jiménez, "A P-Lingua based simulator for tissue P systems with cell separation," *Romanian Journal of Information Science and Technology*, vol. 17, no. 1, pp. 89–102, 2014.

[120] P. Guo, C. Quan, and L. Ye, "UPSimulator: A general P system simulator," *Knowledge-Based Systems*, vol. 170, pp. 20–25, 2019.

[121] Z. Dang, O. H. Ibarra, C. Li, and G. Xie, "On the decidability of model-checking for P systems," *J. Autom. Lang. Comb.*, vol. 11, no. 3, pp. 279–298, 2006.

[122] M. Gheorghe, R. Ceterchi, F. Ipate, S. Konur, and R. Lefticaru, "Kernel P systems: from modelling to verification and testing," *Theoretical Computer Science*, vol. 724, pp. 45–60, 2018.

[123] M. Gheorghe, S. Konur, and F. Ipate, "Kernel P systems and stochastic P systems for modelling and formal verification of genetic logic gates," in *Advances in unconventional computing*, pp. 661–675, Springer, 2017.

[124] M. Hermann, C. Kirchner, and H. Kirchner, "Implementations of term rewriting systems," *The Computer Journal*, vol. 34, no. 1, pp. 20–33, 1991.

[125] J. Bergstra and J. Tucker, "A characterisation of computable data types by means of a finite equational specification method," in *International Colloquium on Automata, Languages, and Programming*, pp. 76–90, Springer, 1980.

[126] J. Siekmann, "Unification theory," *Journal of Symbolic computation*, vol. 7, no. 3-4, pp. 207–274, 1989.

[127] L. Bachmair, *Canonical equational proofs*. Springer, 1991.

[128] U. Martin, "How to choose the weights in the Knuth Bendix ordering," in *International Conference on Rewriting Techniques and Applications*, pp. 42–53, Springer, 1987.

[129] K. Korovin and A. Voronkov, "Orienting rewrite rules with the Knuth–Bendix order," *Information and Computation*, vol. 183, no. 2, pp. 165–186, 2003.