Time-freeness and Clock-freeness and Related Concepts in P Systems [⋆]

Artiom Alhazov[1,2⋆⋆], Rudolf Freund[3], Sergiu Ivanov[4,5], Linqiang Pan[2,6], and Bosheng Song[2]

[1] Institute of Mathematics and Computer Science
Academy of Sciences of Moldova
Academiei 5, Chişinău, MD-2028, Moldova
artiom@math.md

[2] Key Laboratory of Image Information Processing
and Intelligent Control of Education Ministry of China,
School of Automation,
Huazhong University of Science and Technology,
Wuhan 430074, China

[3] Faculty of Informatics, TU Wien
Favoritenstraße 9–11, 1040 Vienna, Austria
rudi@emcc.at

[4] LACL, Université Paris Est – Créteil Val de Marne
61, av. Général de Gaulle, 94010, Créteil, France
sergiu.ivanov@u-pec.fr

[5] TIMC-IMAG/DyCTiM, Faculty of Medicine of Grenoble,
5 avenue du Grand Sablon, 38700, La Tronche, France
sergiu.ivanov@univ-grenoble-alpes.fr

[6] School of Electric and Information Engineering,
Zhengzhou University of Light Industry,
Zhengzhou 450002, China

Summary. In the majority of models of P systems, rules are applied at the ticks of a global clock and their products are introduced into the system for the following step. In timed P systems, different integer durations are statically assigned to rules; time-free P systems are P systems yielding the same languages independently of these durations. In clock-free P systems, durations are real and are assigned to individual rule applications;

[⋆] The work is supported by National Natural Science Foundation of China (61320106005 and 61033003) and the Innovation Scientists and Technicians Troop Construction Projects of Henan Province (154200510012). This paper was finished with Rudolf Freund during Artiom Alhazov's and Sergiu Ivanov's stay in Vienna in August 2017.

[⋆⋆] The work is supported by National Natural Science Foundation of China (61320106005 and 61033003) and the Innovation Scientists and Technicians Troop Construction Projects of Henan Province (154200510012).

thus, different applications of the same rule may last for a different amount of time. In this paper, we formalise timed, time-free, and clock-free P system within a framework for generalised parallel rewriting. We then explore the relationship between these variants of semantics. We show that clock-free P systems cannot efficiently solve intractable problems. Moreover, we consider un-timed systems where we collect the results using arbitrary timing functions as well as un-clocked P systems where we take the union over all possible per-instance rule durations. Finally, we also introduce and study mode-free P systems, whose results do not depend on the choice of a mode within a fixed family of modes, and compare mode-freeness with clock-freeness.

1 Introduction

Membrane systems with symbol-objects are formal computational models of distributed multiset rewriting. While standard models often assume maximal parallelism and a global-clock synchronization of rules (overview in [12]), there have been a number of attempts in the literature to relax this condition. The extreme variant are so-called asynchronous systems, where the parallelism is arbitrary instead of maximal [1, 7]. Not surprisingly, in many cases such systems are much weaker (e.g., defining $PsMAT$ instead of $PsRE$) or need much stronger ingredients to be able to perform the same goal.

A different way to relax the global synchronisation condition is lifting the assumption that all rule executions take one step. For example, in timed P systems [5], a numerical function is defined, associating to each rule the positive integer number of steps its application takes. In this context, time-freeness is an (undecidable) property that the result of all computations of a P system does not depend on the timing function.

The motivation for studying time-freeness is investigating the power and the efficiency of P systems that are robust with respect to rule execution times. Yet, the definition of the time-freeness property is not restrictive enough for some goals—the time a rule application lasts cannot be different in different situations. Indeed, since the timing function is defined on the set of rules, the following facts are immediate:

1. If a rule is simultaneously applied multiple times, then all instances finish simultaneously.
2. If a rule is simultaneously applied in different membranes with the same label, then all rules finish simultaneously.
3. If a rule is applied at different steps of a computation, then all instances last for the same amount of time.
4. If a rule is applied in different non-deterministic branches of a computation, then all instances last for the same amount of time.

A number of publications investigate the efficiency of time-free P systems in solving intractable problems, e.g. [15, 16, 17, 18]. We believe that the constructions in these publications rely on the residual synchronisation facts listed above.

In this paper we focus on a variant of timing which allows individual rule executions to last differently. This variant was introduced under the name "clock-freeness" in [14]. In clock-free P systems, rule applications may last for "arbitrary" real periods and even applications of the same rule may have different durations. We prove that clock-freeness deprives any variant of P systems operating under this semantics of the capability of solving intractable (NP-complete) problems in polynomial time.

Clock-freeness changes the way in which a P system operates quite a bit. Indeed, since durations of rule applications are real numbers, such a P system does not follow the ticks of a global clock any more, but instead "listens" to events—situations in which rule applications finish and release new potential reactants (compare this to the preliminary observations in [11]). Such a P system therefore becomes event-driven and operates in continuous time, similarly to the Gillespie algorithm [10] or to data stream-driven reactive programs, e.g. [6]. In the present work, we formally define event-driven P systems and show their relationship to clock-free and time-free P systems. Moreover, we also introduce un-timed and un-clocked P systems, where as a result we take the union of all results obtained by any timing and per-instance timing function, respectively.

We also consider yet another freeness property: mode-freeness. A P system which is mode-free with respect to a family of modes has the same behaviour under all modes from this family. We show a large family of modes under which generating P systems yield trivial languages, but accepting P systems are computationally complete. Finally, we explore the form of some clock-free and mode-free P systems and show some relatively strong connections between the two freeness properties.

This article is organised as follows. Section 2 recalls some basic notions of formal language theory and then introduces a general definition of P systems rewriting objects from a computable set O. Section 3 recalls and formally defines timing functions, time- and clock-free P systems, as well as introduces event-driven P systems. Subsection 3.4 investigates the connections between these objects. Section 4 shows one of the main results of this paper: clock-free P systems cannot solve intractable problems in polynomial time. Section 5 introduces un-timed and un-clocked P systems. Section 6 recalls the notion of an evolution mode, introduces mode-freeness with respect to a family of modes, and then shows some properties of mode-free P systems. Section 7 compares clock-freeness with mode-freeness and points out some connections between these two properties. Section 8 discusses further possibilities for defining per-instance timing and clock-freeness. Section 9 concludes the paper and also lists several open problems.

2 Preliminaries

We assume the reader to be familiar with the basics of formal language theory and P systems, but we recall some of the notions for convenience. For further introduction to the theory of formal languages and P systems, we refer the reader to [12, 13].

After recalling these basic notions, we will give a formal explanation of general rewriting in order to be able to introduce a general definition of P systems as hierarchical rewriting systems, somewhat in the spirit of [2] and [8].

2.1 Multisets

A *multiset* over V is any function $w : V \to \mathbb{N}$; $w(a)$ is the *multiplicity* of a in w. A multiset w is often represented by one of the strings containing exactly $w(a)$ copies of each symbol $a \in V$. The set of all multisets over the alphabet V is denoted by V°. By abusing string notation, the empty multiset is denoted by λ. We will also (ab)use the symbol $\in$ to denote the relation "is a member of" for multisets. Therefore, for a multiset w, $a \in w$ will stand for $w(a) > 0$.

Given two multisets $w, v \in V^\circ$, w is a submultiset of v if $w(a) \leq v(a)$, for all $a \in V$. In this case, removing w from v means constructing the multiset $v - w$ with the property $(v - w)(a) = v(a) - w(a)$.

For a multiset of tuples $w \in (A_1 \times \ldots \times A_n)^\circ$ we will use the notation $w|_{A_i}$ to refer to the multiset of projections of the elements of w on the dimension A_i, $1 \leq i \leq n$. Formally, $w|_{A_i} \in A_i^\circ$ and $w(a_i)$ for a fixed $a_i \in A_i$ is equal to the number of tuples of the form $(a_1, \ldots, a_i, \ldots, a_n)$ in w.

2.2 General Sequential and Parallel Rewriting

Consider an (infinite, computable) alphabet of objects O. An *O-rewriting rule* is a partial function $r : O \to O$. For an object $o \in O$ for which $r(o)$ is undefined, we say that r is not applicable to o. Often, the semantics of computing $r(o)$ is given by "removing the left-hand side" or r from the object o and then "adding back the right-hand side". Accordingly, we define the pair of partial functions $r_-, r_+ : O \to O$ such that their total effect is the same as that of r, i.e., $r = r_+ \circ r_-$.

Example 1. Consider the alphabet $V = \{a, b\}$ and the set $O = V^\circ$ of all finite multisets over V. The partial function $r : V^\circ \to V^\circ$ replacing an instance of a with two instances of b is an O-rewriting rule and is often written as $r : a \to bb$ or $r : a \to b^2$ (note that, in this notation, the symbol $\to$ is used to specify rule sides and not the domain or the codomain of a function); r is defined for all multisets containing at least an instance of a and is undefined for all other multisets.

For the multiset rewriting rule r, the value $r_-(w)$ can be defined by removing the left-hand side a from the multiset w (if possible) and $r_+(w)$ by adding the right-hand side bb to w. Thus, $r = r_+ \circ r_-$.

Example 2. Consider, again, the alphabet $V = \{a, b\}$ and the set $O = V^*$ of all finite strings over V. In this case, an O-rewriting rule is a partial function $r : O \to O$ replacing a substring *at a particular position*. To express the effect of rewriting *any* substring of a string $s \in O$ satisfying some particular criteria, we need to consider a family of functions $(r_i : O \to O)_{i \in I}$ replacing the substring at its i-th occurrence. To express the effect of rewriting any substring in any finite string in O, we need to consider the family of functions $(r_i : O \to O)_{i \in \mathbb{N}}$.

Fix a set of O-rewriting rules R. To capture the possibility of applying multiple rules R in parallel, we define the (computable) partial function $\mathit{apply} : R^\circ \times O \to O$ which applies a multiset of rules from R to an object from O and yields a new object in O, if possible. As for the case of individual rules, to represent the idea of "removing the left-hand sides" and "adding the right hand sides", we define two other mappings apply_- and apply_+ such that $\mathit{apply}(\rho, o) = \mathit{apply}_+(\rho, \mathit{apply}_-(\rho, o))$, with $\rho \in R^\circ$ and $o \in O$.

Example 3. Consider the alphabet $V = \{a, b\}$ and $O = V^\circ$, as in Example 1, and two rewriting rules $r_1 : ab \to bb$ as well as $r_2 : bb \to a$. Take the multiset of rules $\rho = r_1 r_2$; classically, the function $\mathit{apply}(\rho, w)$ is defined for such multisets $w \in O$ which contain the submultiset $ab^3 = ab\,bb$, necessary to satisfy both the applicability requirements of rules r_1 and r_2. In this case, $\mathit{apply}_-(\rho, w)$ is the function removing the multiset ab^3 from w, $\mathit{apply}_+(\rho, w)$ is the function adding $bb\,a$ to w, and $\mathit{apply}(\rho, w)$ is the function first removing ab^3 from w and then adding $bb\,a$.

A *sequential O-rewriting framework* is the pair (O, R), where O is a set of objects and R is a set of R-rewriting rules. A *parallel O-rewriting framework* is the pair $(O, R, \mathit{apply}_-, \mathit{apply}_+)$, where (O, R) is a sequential O-rewriting framework and $\mathit{apply}_-, \mathit{apply}_+ : R^\circ \times O \to O$ are the (computable) partial functions defining the semantics of parallel application of rules from R to objects in O.

Our definition of rewriting frameworks are strongly inspired by the work [8].

2.3 P Systems

The definition of P systems we give in this paper directly generalises various models of cell-like (hierarchical) P systems in which rules are "located within" the membranes and whose membrane structure may evolve: transition P systems with membrane dissolution rules, P systems with active membranes, etc.

A comprehensive overview of different flavors of membrane systems and their expressive power is given in the handbook which appeared in 2010, see [12]. For a state of the art snapshot of the domain, we refer the reader to the P systems website [20], as well as to the bulletin of the International Membrane Computing Society [19].

Given a parallel O-rewriting framework $(O, R, \mathit{apply}_-, \mathit{apply}_+)$, a P system is the following tuple:

$$\Pi = (O, O_T, \mu, w_1, \ldots, w_n, I, R_1, \ldots, R_n, h_i, h_o),$$

where O is a (computable, infinite) set of objects, $O_T \subseteq O$ is a (computable) set of terminal objects, μ is the initial membrane structure injectively labelled by the numbers from $\{1, \dots, n\}$ and usually given by a sequence of correctly nested brackets, I is the set of allowed *ingredients* (explained below), $w_i \in O$ is the initial object in membrane i, $R_i \subseteq R \times I$ is the set of O-rewriting rules associated with membrane i and enriched with some ingredients, $1 \leq h_i \leq n$ is the label of the input membrane and $1 \leq h_o \leq n$ is the label of the output membrane.

The set of ingredients I in the above definition captures the variety of additional actions which may be associated with O-rewriting rules. We give some examples:

- *Target indications*: If $O = V^\circ$, target indications can be represented by defining $I = \{none\} \cup (V \times Tar)^\circ$, thus allowing rules to specify multisets of pairs (a, tar) of symbols $a \in V$ and target indications $tar \in Tar$.
- *Membrane dissolution*: Membrane dissolution can be represented by defining $I = \{none, \delta\}$ and by writing non-dissolving rules as $(u \to v, none)$ and dissolving rules as $(u \to v, \delta)$, with the usual dissolution semantics.
- *Membrane division, creation, etc.*: Similarly to dissolution, any modification of the membrane structure may expressed by adding the corresponding symbols to the set I.

Finally, note that membrane polarisations can be represented without ingredients by extending the set of objects to $O \times \pi$, where π is the set of polarisations (e.g., $\pi = \{-, 0, +\}$), and by having the rules read and modify the polarisations if necessary.

A *configuration* of the P system Π is the tuple $C = (\mu', w'_1, \dots, w'_n)$, where μ' is the current membrane structure and $w'_i \in O$ is the object contained in membrane i. For P systems which do not dynamically modify their membrane structure, the first component (μ') of the tuple may be omitted.

A *k-step computation of Π* is a sequence of configurations $(C_j)_{0 \leq j \leq k}$ with the following properties:

- $C_0 = (\mu, w_1, \dots, w'_{h_i}, \dots, w_n)$, where μ is the initial membrane structure of Π, w_i, $1 \leq i \leq n$, is the initial object in membrane i, and $w'_{h_i} = w_{h_i} \uplus w_{in}$, where $w_{h_i} \in O$ is the initial object in the input membrane h_i, $w_{in} \in O$ is the input object, and $\uplus$ is the operation of combining two objects (e.g., multiset union if $O = V^\circ$);
- for any configuration C_j, $0 \leq j < k$, the configuration C_{j+1} can be obtained from C_j by applying the rules to the objects of C_j according to a fixed *evolution mode* (e.g., the maximally parallel mode), and by then executing the actions required by the ingredients associated with the applied rules;
- C_k is a *halting configuration*, i.e., a configuration satisfying the halting condition of Π. One of the best known halting condition is requiring that no rule be applicable any more according to the fixed derivation mode (total halting by inapplicability).

The *result* of the computation $(C_j)_{1 \leq j \leq k}$ is derived from the object w_{h_o} found in membrane h_o in the halting configuration C_h. A typical way of deriving the result is applying the terminal projection $p_T : O \to O_T$ which allows for retrieving the "terminal part" $p_T(w_{h_0})$. Another way may be declaring that the derivation $(C_j)_{1 \leq j \leq k}$ only produces a result if $w_{h_o} \in O_T$ (otherwise Π produces no result).

P systems as we defined them are general device computing functions, yet particular cases are often considered. Π is said to work in the *generating mode* if it takes no input (the starting configuration C_0 is the same for all computations). Π is said to work in the *accepting mode* if it takes an input and accepts by a halting computation, whereas non-accepted inputs only yield non-halting computations.

A special case of accepting P systems are *deciding* P systems: for any input, all its computations must halt and are grouped into two classes—accepting and rejecting; for each input, all computations must belong to one of these groups. One usual way of discriminating between the two types of computation is by looking at the form of the object w_{h_o} in the output membrane in the halting configuration: e.g., if $O = V^\circ$, an accepting halting configuration of Π must contain the symbol *yes* in w_{h_o} and a rejecting halting configuration must contain the symbol *no*.

We will denote the language of objects generated (respectively, accepted) by the P system Π by $L_{gen}(\Pi)$ (respectively, $L_{acc}(\Pi)$). Sometimes we will use the notation $L(\Pi)$ when the context makes it clear whether Π is an acceptor or a generator.

3 Time- and Clock-freeness

In this section, we briefly recall (and generalise) the definition of timed and time-free P systems originally introduced in [5]. We then recall the original definition of clock-free P systems as introduced in [14] and give a formalisation. Finally, we show how clock-freeness can easily be captured via a simpler *event-driven semantics* (a natural continuation of [11]).

We start by defining the notion of a rule queue. Given a set of rules R, and the set of ingredients I, we will call any finite multiset of rules $\rho \in (R \times I)^\circ$ a *rule queue*. For a number set X, we will call any finite multiset $\rho \in (R \times I \times X)^\circ$ an *X-timed rule queue*. Intuitively, a rule queue is just an unordered collection of rules and ingredients, while an X-timed rule queue is a collection of rules and ingredients which have *timestamps*.

Given a P system Π, an *extended configuration* (with rule queues) is a tuple $C = (\mu, w_1, \dots, w_n, \rho_1, \dots, \rho_n)$, where $C = (\mu, w_1, \dots, w_n)$ is a configuration of Π and ρ_i is a rule queue (with or without timestamps).

3.1 Time-free P Systems

We will now recall the definitions of timed and time-free P systems from [5] and generalise them to our definition of P systems.

Given a P system Π as defined in Subsection 2.3, a *timing function* is a computable mapping $e : R_\Pi \to \mathbb{N}_+$, with $\mathbb{N}_+ = \mathbb{N} \setminus \{0\}$ and $R_\Pi = \bigcup_{1 \leq i \leq n} R_i$, which assigns *durations* to the rules of Π. The *timed P system* $\Pi(e)$ is a P system with semantics modified in the following way.

Computations of Π are sequences of extended configurations with $\mathbb{N}_+$-timed rule queues (i.e., the rules in rule queues have natural timestamps). To compute the configuration C_{j+1} from a configuration C_j, consider the membrane i containing the object w_i and the rule queue ρ_i. $\Pi(e)$ shall perform the following actions:

1. Constitute the submultiset of rules ρ_{now} of the queue ρ_i which have the timestamp $j+1$; in other words, any tuple in ρ_{now} must have the form $(r, i, j+1)$. Build the new multiset ρ'_i by removing ρ_{now} from ρ_i. Take the multiset $\rho_{now}|_R$ of all O-rewriting rules in ρ_{now} and compute the object w'_i in the following way: $w'_i = apply_+(\rho_{now}|_R, w_i)$. Finally, implement the effects of all the ingredients listed in ρ_{now}.
2. Pick a multiset of rules ρ_{app} applicable to w'_i according to a fixed evolution mode and set the timestamp for every rule r added to ρ_{app} to $j+1+e(r)$. Take the multiset $\rho_{app}|_R$ of all O-rewriting rules in ρ_{app} and compute the new object w''_i in the following way: $w''_i = apply_-(\rho_{app}|R, w'_i)$. Add ρ_{app} to ρ'_i thus constituting the new rule queue ρ''_i.
3. In configuration C_{j+1}, set the contents of membrane i to w''_i and its rule queue to ρ''_i.

Thus, the queues in an extended configuration C_j contain the rules whose application started in the previous steps (excluding step j), including the rules which are scheduled to finish at step j. All queues are empty in the starting configuration and the first evolution step consists in launching some rules (in a sense, it is a "dummy" step or a "half step").

To halt, $\Pi(e)$ needs to exhaust all of the rule queues: that is, the evolution continues until there are still rules scheduled to finish in some future steps, and all queues must be empty in the halting configuration.

The result of a computation of the timed P system $\Pi(e)$ is derived from the contents of its output membrane in its halting configuration in the same way as described for non-timed P systems in Subsection 2.3.

A P system Π is called *time-free* if there exists a language of objects $L \subseteq O$ such that $L(\Pi(e')) = L(\Pi(e))$, *for any (computable)* timing functions $e : R_\Pi \to \mathbb{N}_+$ and $e' : R_\Pi \to \mathbb{N}_+$, and $L = L(\Pi(e))$ for some timing function $e : R_\Pi \to \mathbb{N}_+$. Therefore, time-freeness is the property of P systems to yield the same results independently of durations statically assigned to the rules.

3.2 Clock-free P Systems

In this subsection, we will formally define clock-free P systems following the original work [14]. The motivating intuition is as follows: real-world processes are rarely synchronised via a shared global clock. Timed and time-free P systems capture the fact that processes may have different durations and that some systems are robust to arbitrary variations in such durations; however the durations are integer numbers, which still implies the presence of a discrete global clock. Furthermore, in timed P systems, all applications of the same rule last for the same amount of time, which does not take into account the variations in the execution time of different instances of the same process. Clock-free P systems as introduced in [14] lift both of these restrictions: different applications of *the same rule* are allowed to last for different, *real*, amounts of time.

Following the same scheme as for timed and time-free P systems, we can introduce per-instance real rule timing in the following way. Consider a P system Π as defined in Subsection 2.3 with O-rewriting rules enriched with ingredients $R_\Pi \times I$ and the set $\mathcal{C}$ of all sequences of extended configurations of Π. A *per-instance (real) timing function* is a mapping $\tau : \mathcal{C} \times (R_\Pi \times I)^\circ \to (R_\Pi \times I \times \mathbb{R}_+)^\circ$, with $\mathbb{R}_+ = \{x \in \mathbb{R} \mid x > 0\}$, assigning positive real durations to the rules in a multiset of rules based on the given history of configurations.

Even before we define the effect of per-instance timing function, we give an informal example to give an intuitive impression.

Example 4. Consider the following one-membrane multiset rewriting P system:

$$\Pi_1 = (\{a,b\}^\circ, \{a,b\}^\circ, [\ ]_1, a, \{none\}, \{r_1 : a \to bb, r_2 : b \to aa\}, 1, 1)$$

and suppose it works in the maximally parallel mode. Take the initial configuration $C_0 = ([\ ]_1, aa, \lambda)$. Suppose we want to apply the rule $r_1 : a \to bb$ twice in this configuration. We will define the per-instance timing function τ to have the value $(r_1, none, 0.5)(r_1, none, \sqrt{2})$ for the singleton sequence (C_0) and the multiset of rules $(r_1, none)^2$. This will move the system into the configuration $C_1 = ([\ ]_1, \lambda, (r_1, none, 0.5)\,(r_1, none, \sqrt{2}))$.

Among the two applications of r_1, one is scheduled to finish earlier, at time 0.5. At this moment, it releases the multiset bb into the skin, which renders the rule r_2 applicable. We define the per-instance timing function τ to have the value $(r_2, none, \sin 1)(r_2, none, \cos 1)$ for the sequence $(C_0 C_1)$ and the multiset of rules $(r_2, none)^2$. This moves the system into the configuration $C_2 = ([\ ]_1, \lambda, \rho_2)$ with $\rho_2 = (r_1, none, \sqrt{2})(r_2, none, 0.5 + \sin 1)(r_2, none, 0.5 + \cos 1)$.

We will now define the semantics of per-instance real timing functions. Take a P system Π and fix a per-instance real timing function τ for it. Computations of $\Pi(\tau)$ are sequences of extended configurations with $\mathbb{R}_+$-timed rule queues (compare this with $\mathbb{N}_+$-timed rule queues for timed P systems recalled in Subsection 3.1). The queues in an extended configuration C_j contain the rules whose applications started in configurations previous to C_j (according to a fixed derivation mode), including rules scheduled to finish in this configuration. Consider a sequence $\gamma = (C_m)_{0 \le m \le j}$ of extended configurations with $\mathbb{R}_+$-timed rule queues. To compute the next configuration C_{j+1} from this sequence, $\Pi(\tau)$ proceeds in the following way:

1. Find the smallest timestamp $t_j \in \mathbb{R}$ across all rule queues in configuration C_j.
2. In every membrane i, take the submultiset ρ_{now} of the queue ρ_i in which the rules have the timestamp t_j and compute the object w_i' in the following way: $w_i' = apply_+(\rho_{now}|_R, w_i)$; also implement the effects of the ingredients listed in

ρ_{now}. Build ρ_i' by removing ρ_{now} from ρ_i. (This procedure is identical to that described in the semantics of timed P systems in Subsection 3.1, point 1.)
3. In every membrane i, pick a multiset of rules ρ_{app} applicable to w_i' according to a fixed evolution mode, compute $\rho_{app}' = \tau(\gamma, \rho_{app})$, add t_j to all timestamps in ρ_{app}', and add the result to the new rule queue ρ_i'. Take the multiset $\rho_{app}|_R$ of all O-rewriting rules in ρ_{app} and compute the new object w_i'' in the following way: $w_i'' = apply_-(\rho_{app}|R, w_i')$. Add ρ_{app} to ρ_i', thereby forming the new queue ρ_i''. (This procedure is very similar to that described in the semantics of timed P systems in Subsection 3.1, point 2.)
4. In configuration C_{j+1}, set the contents of membrane i to w_i'' and its rule queue to ρ_i''.

Like for timed P systems, the starting configuration of any computation of $\Pi(\tau)$ has all rule queues empty, and, to halt, $\Pi(\tau)$ needs to exhaust all queues.

The result of a computation of the P system $\Pi(\tau)$ equipped with the per-instance real timing function τ is derived from the contents of the output membrane in the halting configuration in the same way as described for non-timed P systems in Subsection 2.3.

A P system Π is called *clock-free* if there exists a language of objects $L \subseteq O$ such that $L(\Pi(\tau')) = L(\Pi(\tau))$, *for any (computable)* per-instance real timing functions τ and τ ', and $L = L(\Pi(\tau))$ for some per-instance real timing function τ. Therefore, clock-freeness is the property of P systems to yield the same results independently of positive real durations dynamically assigned to rule applications.

We will explicitly explain why our definition corresponds exactly to the slightly informal presentation given in [14]. In the cited paper, the author states that every rule application may have a different real duration. His proofs suppose durations may be arbitrary, but show computational completeness nevertheless. The fact that rule applications may have different real durations is captured by our per-instance real timing functions. Robustness with respect to arbitrary durations is captured by our definition of the clock-freeness property.

Example 5. Consider again the P system Π from Example 4 and the sequence of extended configurations (C_0, C_1, C_2). The corresponding evolution of the rule queue associated with the only membrane of Π is illustrated in Figure 1.

The hollow bullets on the time axis (denoted by the letter t on the figure) mark the "steps", i.e., the moments at which there is at least a rule which finishes its execution and when Π has to check whether any new rules have to be started. Clearly, the illustration does not show a halting computation of Π: new rules are started at moments $t = 0$ and $t = 0.5$, but, of course, other rules are applicable at the other moments highlighted in the figure. We do not show or treat them to avoid clutter.

Finally, we define an important class of per-instance real timing functions. We will call such a function τ *a Markovian real timing function* if its value does not depend on the first argument. Formally, τ is Markovian if, for a fixed multiset of

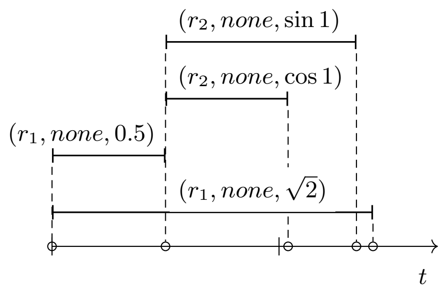

Fig. 1. A graphical illustration of the two-step computation of Π_1 described in Example 4. The hollow bullets mark the "steps".

rules ρ and for any two sequences of configurations γ_1 and γ_2, the following holds $\tau(\gamma_1, \rho) = \tau(\gamma_2, \rho)$. We will call P systems which are clock-free with respect to the class of Markovian timing functions *Markovian clock-free.*

3.3 Event-driven P Systems

Consider again the semantics of P systems with per-instance real timing functions, and especially the illustration in Figure 1. The evolution of such P systems is quite clearly centred around the concept of an *event*: the moment at which some rule executions finish and release the results into the membrane. The computations driven by per-instance real timing functions are essentially sequences of such events. This intuitively implies that what only matters is the order in which rules finish, and not so much the actual individual timings. This observation was stated in a preliminary form in [11].

In this section we first introduce event-driven P systems and then show the equivalence between this variant and clock-free P systems.

Following the same scheme as for per-instance real timing functions, we can define finishing functions in the following way. Consider a P system as defined in Subsection 2.3 with O-rewriting rules enriched with ingredients $R_\Pi \times I$ and the set $\mathcal{C}$ of all sequences of extended configurations of Π with simple rule queues (no timestamps). A *finishing function* is a mapping $\phi : \mathcal{C} \times (R_\Pi \times I)^\circ \to (R_\Pi \times I)^\circ$ indicating, based on the history of configurations, which rules from a given rule queue must finish their execution. Note that ϕ may also return an empty multiset.

Take a P system Π and fix a finishing function ϕ for it. We define the semantics of $\Pi(\phi)$ in the following way. Computations of $\Pi(\phi)$ are sequences of configurations with simple rule queues (no timestamps). Again, the rule queues of an extended configuration C_j contain the rules whose applications started before C_j according to the corresponding fixed derivation mode and $\Pi(\phi)$. Given a sequence $\gamma = (C_m)_{0 \leq m \leq j}$ of extended configurations with simple queues, $\Pi(\phi)$ proceeds in the following way to obtain the configuration C_{j+1}. In membrane i containing the object w_i and the rule queue ρ_i, Π does the following:

1. Apply the finishing function to find the submultiset of rules ρ_{now} which must finish: $\rho_{now} = \phi(\gamma, \rho_i)$. Take the multiset $\rho_{now}|_R$ of all rewriting rules in ρ_{now} and compute the new object $w'_i = \mathit{apply}_+(\rho_{now}|_R, w_i)$; also implement the effect of the ingredients listed in ρ_{now}. Build ρ'_i by removing ρ_{now} from ρ_i.
2. Pick a multiset of rules ρ_{app} applicable to w'_i according to a fixed evolution mode and add ρ_{app} to ρ'_i, thereby constituting the new rule queue ρ''_i. Compute the new object w''_i in the following way: $w''_i = \mathit{apply}_-(\rho_{app}, w'_i)$.
3. In configuration C_{j+1}, set the contents of membrane i to w''_i and its rule queue to ρ''_i.

As before (Subsections 3.1 and 3.2), all computations start with empty rule queues and the system needs to exhaust all rule queues in order to halt. The result is retrieved as for P systems operating under conventional semantics (Subsection 2.3).

Recall that the finishing function ϕ is allowed to return an empty multiset. In this paper, we choose to only consider functions which, for a given sequence of configurations γ of a fixed P system Π, return a *non-empty multiset* for at least one rule queue (non-denying functions). This ensures that every configuration in a computation of Π corresponds to a rule finishing event.

Example 6. Consider again the P system from Example 4:

$$\Pi_1 = (\{a,b\}^\circ, \{a,b\}^\circ, [\ ]_1, a, \{\mathit{none}\}, \{r_1 : a \to bb, r_2 : b \to aa\}, 1, 1)$$

and the first three configurations of its computation (C_0, C_1, C_2) illustrated in Figure 1. We can reproduce the effects of these three steps using a finishing function. The initial configuration will be, as before, $K_0 = ([\ ]_1, aa, \lambda)$. In this configuration the maximally parallel mode forces Π to apply r_1 twice and to move into the following configuration $K_1 = ([\ ]_1, \lambda, (r_1, \mathit{none})^2)$. We define the finishing function ϕ to return (r_1, none) for history (K_0, K_1) and the queue (r_1, none). This will release the products of r_1 into the skin membrane and render r_2 applicable. The maximally parallel derivation mode enforces the two applications of r_2, moving Π_1 into the configuration $K_2 = ([\ ]_1, \lambda, (r_1, \mathit{none})\,(r_2, \mathit{none})^2)$.

We now show side by side the configurations C_0, C_1, and C_2 of $\Pi(\tau)$ working under the per-instance real timing function τ from Example 4 and the configurations K_0, K_1, and K_2 from the previous example (we denote $t_1 = 0.5 + \sin 1$ and $t_2 = 0.5 + \cos 1$):

Note that, with the finishing function from Example 6, we are able to reproduce the contents of rule queues in (C_0, C_1, C_2), without using time stamps. We will later formally show that per-instance real timing functions are equivalent to finishing functions, which makes them into a useful instrument for reasoning about computations with per-instance real timing.

For a P system Π to be independent of the finishing strategy ϕ means that there exists a language $L \subseteq O$ of objects of Π such that $L = \Pi(\phi)$ for *any computable* finishing function ϕ.

	C_i	K_i
0	$([\ ]_1, aa, \lambda)$	$([\ ]_1, aa, \lambda)$
1	$([\ ]_1, \lambda,\ (r_1, none, 0.5)\,(r_1, none, \sqrt{2}))$	$([\ ]_1, \lambda,\ (r_1, none)^2)$
2	$([\ ]_1, \lambda,\ (r_1, none, \sqrt{2})\,(r_2, none, t_1)\,(r_2, none, t_2))$	$([\ ]_1, \lambda,\ (r_1, none)\,(r_2, none)^2)$

Table 1. A comparison between the forms of configurations in Examples 4 and 6. We denote $t_1 = 0.5 + \sin 1$ and $t_2 = 0.5 + \cos 1$.

Since a finishing function essentially defines the *sequencing* of the releases of "processed" rule products, P systems which are independent of this sequencing can be seen as "waiting" for events to happen and "handling" them. Thus, we will refer to such systems using the term *event-driven P systems*.

Finally, in analogy with Markovian per-instance timing functions, we define Markovian finishing strategies. We will call a strategy ϕ *a Markovian finishing strategy* if its value does not depend on the first argument. Formally, ϕ is Markovian if, for a fixed multiset of rules ρ and for any two sequences of configurations γ_1 and γ_2, the following holds: $\tau(\gamma_1, \rho) = \tau(\gamma_2, \rho)$. We will call P systems which are event-driven with respect the class of Markovian finishing functions *Markovian event-driven*.

3.4 Timing Types and Finishing Strategies

Because timed P systems, P systems with per-instance real timing, and P systems with finishing strategies stem from the same idea—introduce rule durations to P systems—it is not surprising that these models have a lot in common. In this subsection, we outline the main connections.

First of all, we would like to bring the reader's attention upon the form of the configurations shown in Table 1: in many of them, the multisets contained in the membranes are empty, the "semantic focus" being on rule queues. This is an effect which may be surprising at first, but which actually underlines the important difference of P systems with rule queues as compared to usual P systems: in the former case, configurations mark the *intervals* the start of some rule applications and the end of some other rule applications (also seen in Figure 1), while configurations for P systems operating under conventional semantics (Subsection 2.3) capture the *moments* between the end of some rule applications and the start of some other rule applications.

We will now show a series of intuitively clear inclusions of families of P systems with rule queues. We start with a general statement about timing functions and per-instance real timing functions.

Proposition 1. *Given a P system Π and any timing function e (Subsection 3.1) there exists a per-instance real timing function τ_e (Subsection 3.2) such that $L(\Pi(e)) = L(\Pi(\tau_e))$.*

Proof. Consider the timing function τ_e always assigning the duration $e(r)$ to any application of the rule r in any evolution of Π. It follows from the definitions of semantics in Subsections 3.1 and 3.2 that, for any computation γ_e of $\Pi(e)$ there exists a computation γ_τ of $\Pi(\tau_e)$ producing the same output object, and conversely. Moreover, for a given step j, the configurations $C_j \in \gamma_e$ and $K_j \in \gamma_\tau$ are identical (modulo the inclusion of $\mathbb{N}$ into $\mathbb{R}$).

The converse proposition is not true: there exist per-instance timing functions which do *not* have a corresponding timing function.

Proposition 2. *There exists a multiset-rewriting P system Π and a per-instance timing function τ such that $L(\Pi(\tau)) \neq L(\Pi(e))$ for any timing function e.*

Proof (Sketch). Consider the one-membrane multiset rewriting P system Π with the following rules:

$$\begin{array}{ll} r_1 : a \to c & r_3 : cb \to d \\ r_2 : c \to f & r_4 : cd \to \heartsuit \end{array}$$

Fix the starting multiset in the only membrane of Π to aab. We can construct a per-instance real timing function for Π which will yield the evolution shown in Figure 2. Note that the two applications of r_1 take a *different* amount of time,

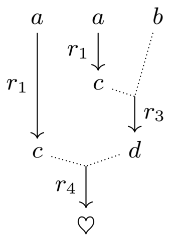

Fig. 2. A computation impossible without per-instance timing (because two applications of $r_1 : a \to c$ take different time).

which lets the first c arrive (on the right) to produce a d together with b so that, when the second c arrives (on the left), it can produce $\heartsuit$ together with d. On the other hand, if all applications of r_1 lasted for the same amount of time, both c's would appear at the same time, and one of them would have to evolve by rule r_3 turning into f and guaranteeing that rule r_4 cannot be applied.

The fact that in timed P systems in the sense of Subsection 3.1 different applications of the same rule last for the same amount of time implies the statement of the proposition.

According to the previous two propositions, per-instance timing allows richer behaviour than simple timing (in the sense of Subsection 3.1). This immediately implies the following statement.

Theorem 1. *A P system Π which is clock-free is also time-free.*

The relationship between per-instance timing functions and finishing strategies is even stronger. In the following, we say that two rule queues are *equivalent modulo timestamps* removing all timestamps from both yields two equal rule queues. We also consider the natural extension of this equivalence to configurations with rule queues.

Proposition 3. *Given an O-rewriting P system Π and any per-instance real timing function τ such that its value $\tau(\alpha, o)$ for the object $o \in O$ does not depend on the timestamps in the sequence of configurations α, there exists a finishing strategy ϕ such that $L(\Pi(\tau)) = L(\Pi(\phi))$.*

Proof. Take a computation γ of $\Pi(\tau)$ and suppose that we have already defined ϕ sufficiently to build a prefix $\bar{\gamma}'_j$ of length j of a computation γ' of $\Pi(\phi)$ in which all configurations are equivalent modulo timestamps to the corresponding configurations in the prefix $\bar{\gamma}_j$ of γ. Extend the definition of ϕ to require the same rules to finish in configuration K_j of γ' as those which are scheduled to finish in C_j in γ. Since we require τ to be independent of the timestamps in $\bar{\gamma}_j$, extending ϕ in this way is always possible. This observation, together with the fact that the starting configurations of γ and γ' are vacuously equivalent modulo timestamps (since their rule queues are empty), implies that we can define ϕ such that all configurations in γ' are equivalent modulo timestamps to the corresponding configurations in γ. This means that the results in the halting configurations of γ and γ' are equal, which proves the proposition.

Corollary 1. *Given a P system Π and any Markovian per-instance real timing function τ, there exists a finishing strategy ϕ such that $L(\Pi(\tau)) = L(\Pi(\phi))$.*

This corollary directly implies the following statement about event-driven P systems and Markovian clock-free P systems.

Theorem 2. *Any P system Π which is event-driven is Markovian clock-free.*

The converse of Proposition 3 also holds.

Proposition 4. *Given a P system Π and any finishing strategy ϕ, there exists a per-instance real timing function τ such that $L(\Pi(\phi)) = L(\Pi(\tau))$.*

Proof. Take a computation γ of $\Pi(\phi)$; γ is a sequence of configurations with simple rule queues (without timestamps). Construct a new sequence of configurations γ' with $\mathbb{R}_+$-timed rule queues in which all rules in all rule queues of configuration C_j get the timestamp j. Now consider the per-instance real timing function τ which assigns exactly these timestamps to rule applications in γ'. The fact that we can always carry out this transformation implies the statement of the proposition.

According to the previous proposition, per-instance timing strategies may ensure richer behaviour than finishing strategies, which implies the following statement.

Theorem 3. *A P system Π which is clock-free is event-driven (in the sense of Subsection 3.3).*

Figure 3 summarises the relations between the various kinds of freeness properties of P systems with rule queues we have considered in this paper. This figure

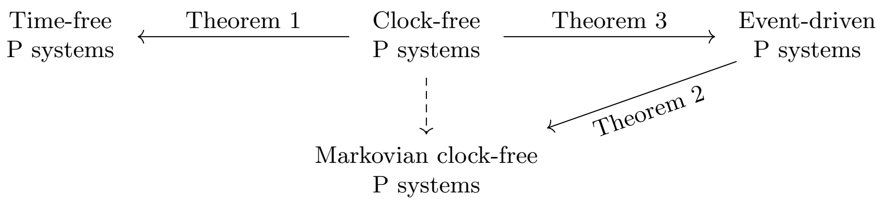

Fig. 3. Inclusions between the different kinds of freeness properties considered in this section.

also takes into consideration that any clock-free P system is trivially Markovian clock-free (because Markovian per-instance timing functions form a proper subclass of per-instance timing functions). This relation is represented as a dashed arrow.

4 Clock-freeness and Efficiency: P versus NP

One of the famous features of some variants of P systems is the capability of solving intractable (NP-complete) problems in polynomial time. The classical approach is generating an exponential number of computing units in polynomial time, which allows fast exploration of the space of candidate solutions (see [12] for some classic examples). Recently, efficient *time-free* solutions to intractable problems have been provided, e.g. in [15, 16, 17, 18]. Since there is no upper bound on the values the timing function may assign, the authors of the cited papers measure the time complexity of their constructions in terms of rule starting steps—the number of moments in the evolution of the P system at which rule executions start—rather than in terms of the total running time.

On the other hand, we tend to see the number of *rule finishing steps* as a better measure for time complexity of time- and clock-free P systems. We take as a motivating example a computation in which rules only start in the first configuration and then finish at different times. This computation has only one starting step, but it may have multiple finishing steps, the number of which is more closely related to the number of events that occurred.

We now show that, assuming that in order to solve an NP-complete problem an exponential number of computing units is necessary, efficient (in terms of the number of finishing steps) *clock-free* solutions are impossible to construct (we assume $\mathsf{P} \neq \mathsf{NP}$). The intuition is as follows: in time-free P systems, we may not assume any particular duration for a given rule application, but we are sure that all applications of the same rule take the same amount of time. The fact that this property is no longer guaranteed under clock-freeness turns out to be essential for (in)efficiency.

Theorem 4. *Consider an* NP*-complete problem $\mathcal{P}$ and take a P system Π solving it. Then there exists a per-instance (real) timing function under which all computations of Π contain an exponential (in the size of the input) number of rule finishing steps (assuming that in order to solve an* NP*-complete problem an exponential number of computing units is needed).*

Proof. In P systems as defined in Subsection 2.3, the atomic "computing units" are single rule applications. Therefore, according to our assumption, Π must run an exponential number of rule applications. Since Π operates under per-instance timing, we can ensure that no two rule applications end at the same time, which implies that Π has exponentially many rule finishing steps.

5 Un-timed and Un-clocked P Systems

In contrast to time-free P systems, where all timing functions have to generate the same results, in an un-timed P systems we collect all the results obtained by using any timing function, i.e., for a given P system Π we define

$$L_{un-timed}(\Pi) = \bigcup_{t \text{ timing function}} L(\Pi, t).$$

Moreover, in the same way, in an un-timed P system we collect all the results obtained by using any per-instance timing function, i.e., for a given P system Π we define

$$L_{un-clocked}(\Pi) = \bigcup_{\tau \text{ per-instance timing function}} L(\Pi, \tau).$$

In an un-clocked P system, we simply may assume each rule application to last an arbitrary amount of time. In the following we give a small example which yields different results when considered as an un-clocked or as an un-timed system and is neither time- nor clock-free:

Example 7. We consider the one-membrane multiset rewriting P system Π with the following non-cooperative rules with inhibitors:

$$r_1 : a \to aa|_{\neg c} \text{ and } \mathrm{r_2 : b \to c}$$

Starting from the axiom ab, in parallel we have to apply both $r_1 : a \to aa|_{\neg c}$ and $b \to c$ in parallel. Considering Π as an un-timed system, r_1 can be applied again and again in parallel to all symbols a being generated until the application of the rule r_2 has finished which immediately stops the derivation by the appearance of the inhibitor c. In sum, we obtain

$$L_{un-timed}(\Pi) = \bigcup_{t \text{ timing function}} L(\Pi, t) = \bigcup_{n \in \mathbb{N}} \{a^{2^n} c\}.$$

Of course, this system is neither time- nor clock-free. The infinite set is generated not due to the choice between applicable rule multisets, but due to the non-deterministic choice of the timing function.

Considering Π as an un-clocked system, again $r_1 : a \to aa|_{\neg c}$ and $b \to c$ have to be applied in parallel in the first step, and r_1 can be applied until the application of the rule r_2 has finished which immediately stops the derivation by the appearance of the inhibitor c. Yet in contrast to the un-timed version, the applications of the rule r_1 to the symbols a appearing in the meantime may end at arbitrary moments of time. To the two symbols a appearing when the first application of rule r_1 has finished, r_1 has to be applied simultaneously to both symbols a, yet from that moment on the different instances of rule r_1 may finish in an unsynchronized way. Hence, as we sum up all possible results, we may restrict ourselves to consider only the events when just one rule application ends. The only symbols to which a rule, i.e., r_1, now can be applied are the two symbols a having evolved as a result of this one rule application, and to these two symbols two copies of r_1 have to be applied simultaneously. In fact, this only means that the number of symbols a has increased by one with each finishing of a rule r_1. Therefore, in sum we obtain

$$L_{un-clocked}(\Pi) = \bigcup_{\tau \text{ per-instance timing function}} L(\Pi, \tau) = \{a^n c \mid n \in \mathbb{N}_+ \setminus \{1, 3\}\}.$$

A similar result can be obtained by the one-membrane multiset rewriting P system Π' with the following non-cooperative rules with promoters:

$$r_1 : a \to aa|_b, \quad r_2 : b \to b, \quad \text{and} \quad \mathrm{r_3 : b \to c}$$

Starting from the axiom ab, we now may assume that the execution of the rules r_1 and r_2 takes exactly the same time, because the promoter b is needed to allow the copies of rules r_1 to be applied. Again, the derivation halts as soon as the promoter b is eliminated by applying r_3. For the un-timed mode, the application of rules r_1 still is synchronized, too, and we therefore obtain

$$L_{un-timed}(\Pi') = \bigcup_{t \text{ timing function}} L(\Pi', t) = \bigcup_{n \in \mathbb{N}} \{a^{2^n} c\}.$$

In the un-clocked mode, the finishing of rules may be arbitrary, yet still the promoter b is needed to continue with applying rule r_1, i.e., only when the application of the rule $r_2 : b \to b$ has finished, rule r_1 can be applied. In sum, we again obtain

$$L_{un-clocked}(\Pi') = \bigcup_{\tau \text{ per-instance timing function}} L(\Pi', \tau) = \{a^n c \mid n \in \mathbb{N}_+ \setminus \{1, 3\}\}.$$

6 Mode-freeness

Considering time-free, clock-free, and event-driven P systems motivates further discussion about robustness with respect to variations of other parameters. In this section we will consider *mode-freeness*: robustness with respect to the choice of the evolution mode.

6.1 Evolution Modes

Take a (computable) set of objects O and consider a parallel O-rewriting framework $(O, R, apply_-, apply_+)$. Following [9], we denote by $Appl(R, o)$ the set of multisets of rules applicable to the object $o \in O$ in parallel. Given an n-membrane O-rewriting P system Π and a configuration C of it, we denote by $Appl(\Pi, C)$ the set of tuples of the form $(\rho_1, \dots, \rho_n)$, in which ρ_i is a multiset of rules applicable to the object w_i in membrane i in configuration C.

Example 8. Consider the multiset ab and the set of multiset rewriting rules $R = \{r_1 : a \to b, r_2 : b \to c\}$. Then $Appl(R, ab) = \{r_1, r_2, r_1 r_2\}$. Take a multiset-rewriting P system Π with the set of ingredients $I = \{none\}$ and two membranes with equal sets of rules $R_1 = R_2 = \{(r_1, none), (r_2, none)\}$. Consider the configuration $C = (\mu, ab, ab)$ of Π, then $Appl(\Pi, C) = A \times A$, where $A = \{(r_1, none), (r_2, none), (r_1, none)(r_2, none)\}$.

Given a P system Π and a configuration C, an *evolution mode* (derivation mode) is a strategy ϑ for filtering the set $Appl(\Pi, C)$. According to [9], we denote by $Appl(\Pi, C, \vartheta) \subseteq Appl(\Pi, C)$ the set of tuples of multisets of rules of Π applicable in configuration C according to the derivation mode ϑ. When $Appl(\Pi, C, \vartheta)$ contains more than one element, Π chooses between the allowed tuples non-deterministically in order to continue the computation. We will denote the language generated (respectively, accepted) by Π operating under the derivation mode ϑ by $L_{gen}(\Pi, \vartheta)$ (respectively, $L_{acc}(\Pi, \vartheta)$).

We will now recall some typical examples of derivation modes considered in [9]. All of these examples are formulated for a P system Π and a configuration C of it.

Example 9. The *asynchronous* derivation mode *asyn* is the mode allowing any combination of rules to be applied: $Appl(\Pi, C, asyn) = Appl(\Pi, C)$.

Example 10. The *sequential* derivation mode *sequ* is the mode only allowing one rule to be applied at any time. $Appl(\Pi, C, sequ)$ therefore contains tuples of singleton multisets of rules.

Example 11. The *maximally parallel* derivation mode *max* only includes tuples of non-extendable multisets of rules.

Formally, for a tuple $(\rho_1, \ldots, \rho_n) \in Appl(\Pi, C, max)$, the set $Appl(\Pi, C)$ contains *no* tuple $(\rho'_1, \ldots, \rho'_n)$ such that at least one ρ_i is a submultiset of ρ'_i, for $1 \leq i \leq n$.

A very interesting derivation mode (considered in a detailed way in [3]) is the following one.

Example 12. The *set-maximally parallel* mode *smax* only allows tuples of multisets containing at most one instance of any rule. Formally, for any tuple $(\rho_1, \ldots, \rho_n) \in Appl(\Pi, C, smax)$, it is true that $\rho_i(r) \leq 1$ for any rule r in membrane i, $1 \leq i \leq n$.

Finally, we show several more derivation modes which we use later.

Example 13. The $max_{\geq k}$ mode only allows tuples of multisets which contain *at least* k rules. That is, for any tuple $(\rho_1, \ldots, \rho_n) \in Appl(\Pi, C, max_{\geq k})$, it must hold that $|\rho_i| \geq k$, for all $1 \leq i \leq k$.

Example 14. Suppose that the sets of rules R_i associated with membranes i of Π, $1 \leq i \leq n$, are equipped with total orders $\leq_i$ and consider the mode *det* ("the determinator") which only allows tuples of singleton multisets of rules, which are also minimal with respect to the corresponding order. Formally, for any tuple $(\rho_1, \ldots, \rho_n) \in Appl(\Pi, C, det)$, it is true that $|\rho_i| \leq 1$ and, for any other tuple of singleton multisets $(\rho'_1, \ldots, \rho'_n) \in Appl(\Pi, C)$, it holds that $r_i \leq_i r'_i$, where $\rho_i = r_i$, $\rho'_i = r'_i$, and $1 \leq i \leq n$.

Note that, according to this definition, $Appl(\Pi, C, det)$ is either empty (if $Appl(\Pi, C)$ is empty) or a singleton set, which justifies the informal name "the determinator".

Finally, an extreme example of an evolution mode.

Example 15. The *empty* evolution mode $\varnothing$ is the evolution mode disallowing any rule applications: $Appl(\Pi, C, \varnothing) = \emptyset$.

6.2 Freeness with Respect to a Family of Modes

Consider an O-rewriting P system Π and the family of evolution modes Θ. We say that Π is *Θ-mode-free* if there exists a language $L \subseteq O$ such that $L = L(\Pi, \vartheta)$, *for all* $\vartheta \in \Theta$. We use the notation $pL_{gen}(O, \Theta)$ to refer to the family of languages over O generated by Θ-mode-free O-rewriting P systems. We replace the subscript *gen* by *acc* to refer to the family of languages accepted by Θ-mode-free O-rewriting P systems.

It turns out that the idea mode-freeness has already been indirectly invoked in the literature. Indeed, the constructions from the paper [3] that literally hold for the modes max and $smax$ are $\{max, smax\}$-mode free.

We start our discussion of more general kinds of mode-freeness by remarking that mode-freeness with respect to the family of all modes (denoted by Θ_U) is a very restrictive condition filtering out non-trivial behaviour.

Proposition 5. *Consider the family of all derivation modes Θ_U. Then the following statements hold:*

- *$pL_{gen}(O, \Theta_U)$ only contains $\emptyset$ and singleton languages,*
- *$pL_{acc}(O, \Theta_U) = 2^O$, where 2^O is the set of all subsets of O, but all computation is done by the procedure extracting the result from the output object.*

Proof. Consider the Θ_U-mode free O-rewriting P system Π. Since Π should yield the same language under *any* mode, it is sufficient to investigate its behaviour under the empty mode $\varnothing$. Under this mode, Π never evolves.

For generation, this means that the result is computed from the initial object placed in the output membrane, which, depending on the procedure for extracting the result, may yield a singleton language or the empty language (e.g., in the case in which the initial object in the output membrane has no corresponding terminal projection).

Suppose now that Π is an acceptor. We will consider the following cases.

- If Π accepts by halting, it accepts any object because it halts immediately: $L_{acc}(\Pi) = O$.
- Suppose Π accepts by placing an object of a specific form into the output membrane.
  - If the output membrane of Π is different from its input membrane and the initial object placed into the output membrane does not satisfy the acceptance criterion, then Π rejects all inputs: $L_{acc}(\Pi) = \emptyset$.
  - If the output membrane of Π is the same as its input membrane, then Π will accept those inputs objects which satisfy the acceptance criterion for output objects. Therefore Π can be made to accept any subset of O by varying its acceptance criterion.

These observations conclude the proof.

To avoid trivial results, we assume in what follows that the procedure for extracting the result out of the output object is reasonably simple.

As we have just seen, Θ_U-mode-freeness is a very strong restriction. We will now consider an important subfamily of Θ_U: non-denying modes. Given a P system Π, a mode ϑ is *non-denying* if, for any configuration C of Π, $Appl(\Pi, C) \neq \emptyset$ implies that $Appl(\Pi, C, \vartheta) \neq \emptyset$. The mode is called denying otherwise. We will use the notation $\Theta_{\neg deny}$ to refer to the subfamily of non-denying modes.

Example 16. The derivation modes *asyn*, *sequ*, *max*, *smax*, and *det* are non-denying. The derivations modes $max_{\leq k}$ and $\varnothing$ are denying modes.

Mode-freeness with respect to non-denying modes turns out to be a much more interesting property than Θ_U-mode-freeness. In the generative case, $\Theta_{\neg deny}$-mode-freeness yields P systems generating singleton languages and the empty language. (Note that Θ_U-mode-free P systems can achieve the same behaviour only by playing on the procedure for extracting the result out of the output object.)

Proposition 6. *$pL_{gen}(O, \Theta_{\neg deny})$ only contains $\emptyset$ and singleton languages.*

Proof. Consider a $\Theta_{\neg deny}$-mode-free P system Π. Since Π should generate the same language under any non-denying mode, we can investigate its behaviour under "the determinator" mode *det*. Under this mode, Π evolves sequentially and deterministically. This means that it can either generate a singleton language, or the empty language in case it never halts or generates an output object without a terminal projection.

The situation changes drastically in the accepting case: indeed, deterministic acceptor P systems simulating deterministic register machines exist (e.g., [4, Theorem 2]). We nevertheless sketch a simple construction here.

Theorem 5. *Given an alphabet V and the set of recursively enumerable multiset languages $RE \subset V^\circ$, the following holds: $pL_{acc}(V^\circ, \Theta_{\neg deny}) = RE$.*

Proof (Sketch). Consider the one-membrane multiset-rewriting P system Π simulating a deterministic register machine M. Π uses cooperation and inhibitors. For every instruction $p : (A(r), q)$ incrementing register r and going from state p to state q, Π has a rule $p \to qr$. For every instruction $p : (S(r), q, z)$ checking register r for zero in state p, decrementing r and moving into state q, or moving into state z if the decrement is not possible, Π includes the rule $pr \to q$ to ensure the decrement and the rule $p \to z|_{\neg r}$ for the zero check.

Two properties follow from this construction sketch:

- Π correctly simulates the computations of M,
- exactly one rule is applicable in any evolution step of Π, which means that Π is sequential and deterministic.

The second property implies that, if we take $L = L_{acc}(\Pi, det)$, then $L = L_{acc}(\Pi, \vartheta)$ for any non-denying derivation mode $\vartheta \in \Theta_{\neg deny}$, i.e., Π is $\Theta_{\neg deny}$-mode-free. The fact that we can perform this construction for any register machine M implies the statement of the theorem.

The two previous statements highlight a huge gap between the generating and the accepting cases under mode-freeness with respect to non-denying modes: mode-free generation only produces trivial languages, while mode-free acceptance is computationally complete.

The construction in Theorem 5 allows us to derive a sufficient criterion for mode-freeness with respect to non-denying modes.

Theorem 6. *If a P system Π is deterministic under the evolution mode asyn in any reachable configuration, then it is $\Theta_{\neg deny}$-mode-free.*

Proof. For Π to be deterministic under *asyn* means that, for any reachable configuration C, the set $Appl(\Pi, C, asyn)$ is a singleton set. This only happens when at most one rule is applicable to C in the whole system. The effect of all non-denying modes on Π will therefore be the same: apply the only applicable rule (if there exists one), or halt if no more rules are applicable anywhere. This observation implies that Π is $\Theta_{\neg deny}$-mode-free.

The converse statement is not necessarily true in general: for example, a multiset-rewriting P system whose only behaviour consists in erasing all the symbols in the input one by one will be able to behave similarly under any non-denying mode. We do expect the converse statement to be true for "reasonable" P systems, however.

Conjecture 1. Any computationally universal $\Theta_{\neg deny}$-mode-free P system is deterministic under *asyn* in any of its reachable configurations.

We recall that being computationally universal means being capable to "run any program". More concretely, a P system is computationally universal if it can simulate a universal register machine. We refer the reader to [12] for comprehensive explanations.

7 Clock-freeness versus Mode-freeness

In this section we start a discussion about the relationship between clock- and mode-freeness. Despite their different origins, the two freeness properties exhibit a number of similarities. Consider, for example, the sketch of the $\Theta_{\neg deny}$-mode-free P system simulating an arbitrary register machine from Theorem 5. This system is trivially clock-free because at most one rule can be applied at any time. Furthermore, we can reformulate the criterion from Theorem 6 for the clock-free case in the following way.

Theorem 7. *If a P system Π is deterministic under the evolution mode asyn in any reachable configuration, then it is clock-free.*

Proof. By the same arguments as in the proof of Theorem 6, we conclude that a P system Π with the required properties may only apply at most one rule at any step, which trivially implies clock-freeness.

In case Conjecture 1 is true, being $\Theta_{\neg deny}$-mode-free is equivalent to being deterministic under the mode *asyn* for computationally universal P systems. This allows us to formulate the following derived hypothesis.

Conjecture 2 (assuming Conjecture 1). Any computationally universal $\Theta_{\neg deny}$-mode-free P system is also clock-free.

The two statements we have formulated in this section reveal parts of a strong relationship between clock- and mode-freeness. In particular, the previous conjecture warrants wondering whether any clock-free P system is also $\Theta_{\neg deny}$-mode-free. The following example and the associated propositions allow us to answer this question in the negative.

Example 17. Consider a register machine M and construct a one-membrane multiset rewriting P system Π_M in the following way.

- For every instruction $p : (A(r), q, q')$ which increments register r and non-deterministically moves from state p to either state q or q', add the rules $p \to qr$ and $p \to q'r$ to Π_M.
- For every instruction $p : (S(r), q, z)$ decrementing r and moving into state q, or moving into state z if the decrement is not possible, add the following rules to Π_M:

$$p \to p'p_r$$

$$\begin{array}{ll} p' \to p'' & p_r r \to d_r \\ p''p_r \to z & p''d_r \to q \end{array}$$

Π_M operates under the maximally parallel mode, under normal semantics (no timing, finishing strategies, etc.), and simulates the register machine M. Simulation of the increment instruction is straightforward. To simulate the decrement, Π_M splits the state symbol p into p' and p_r. p_r tries to decrement the register r by the rule $p_r r \to d_r$ while p' waits for one step turning into p''. Then, if p'' finds a d_r (meaning that the register was successfully decremented), the rule $p''d_r \to q$ is applied; otherwise the rule $p''p_r \to z$ is applied ensuring the correct choice between states q and z.

Proposition 7. *Π_M operating under the [set-]maximally parallel mode is clock-free (and event-driven).*

Proof. The only moment at which Π_M applies more than one rule is during the simulation of the decrement, when the symbols p' and p_r are produced and the configuration contains an instance of r. In this case, p' and p_r are immediately consumed by the applications of the corresponding rules (because Π_M operates in the [set-]maximally parallel mode). Note that no rule in Π_M is applicable before *both* p'' *and* d_r are produced, which makes the behaviour of Π_M independent of the timings on the individual applications of rules in this branch of the simulation. These observations imply that Π_M is clock-free.

Proposition 8. *Π_M is not $\Theta_{\neg deny}$-mode-free.*

Proof. As seen in the proof of the previous proposition, Π_M operating under the modes *max* and *smax* simulates the register machine M. However, if we fix a total order on the rules of Π_M such that $p' \to p''$ is less than $p_r r \to d_r$, and have Π_M operate under "the determinator" mode *det*, then Π_M will never have the chance to apply the rule $p_r r \to d_r$, meaning that Π_M will not simulate M any more.

This shows that the language accepted/generated by Π_M is different under two different non-denying modes (*max* and *det*) which implies the statement of the theorem.

Corollary 2. *There exists a clock-free P system which is not $\Theta_{\neg deny}$-mode-free.*

8 A Note on the Semantics of Clock-freeness

We would now like to use the instruments we have constituted throughout the paper to point out some issues with the informal introduction of clock-freeness in the original work [14]. These issues appear under derivation modes different from the maximally parallel one—*extendable modes*—or with rules which have *non-monotonous* rule applicability semantics. We now define these terms formally.

Given a P system Π and a configuration C of Π, we will call a mode ϑ *non-extendable* if all multisets in $Appl(\Pi, C, \vartheta)$ are non-extendable, i.e., for any $\rho \in Appl(\Pi, C, \vartheta)$, there exists *no* $\rho' \in Appl(\Pi, C)$ such that ρ is a submultiset of ρ'. If ϑ is not non-extendable, it is called extendable.

Example 18. The maximally parallel mode is by definition a non-extendable mode, but any other mode ϑ such that $Appl(\Pi, C, \vartheta) \subseteq Appl(\Pi, C, max)$ is non-extendable as well.

Given a parallel rewriting framework $\mathcal{F} = (O, R, apply_-, apply_+)$ and a partial order relation $\subseteq$ on O, we say that the rule applicability semantics of $\mathcal{F}$ is *monotonous* if, for two objects $o_1, o_2 \in O$, $o_1 \subseteq o_2$ implies that $Appl(R, o_1) \subseteq Appl(R, o_2)$, where $Appl(R_1, o)$ denotes the set of multisets of rules from R applicable to o_1.

Example 19. The semantics of cooperative multiset rewriting is monotonous: for a fixed set of multiset rewriting rules R and two multisets w_1 and w_2 such that w_1 is a submultiset of w_2, at least as many rules are applicable to w_2 as to w_1.

The semantics of cooperative multiset rewriting rules with *inhibitors* is non-monotonous: consider the singleton set of rules $R = \{r : a \to b|_c\}$ and the multisets $w_1 = a$ and $w_2 = ac$; w_1 is a submultiset of w_2, but r is only applicable to w_1 and not to w_2.

Now consider again the informal definition of clock-free semantics from [14] and take a P system Π with non-monotonous rule applicability semantics. Whenever some rules can be applied, Π has to start their application by "removing their left-hand sides" using $apply_-$. However, since the applicability semantics in Π is non-monotonous, this may immediately render more rules applicable. Letting Π continue "removing the left-hand sides" would mean that Π may run parts of the computation which should follow each other *at the same moment*.

Suppose now that Π works under a mode ϑ which is extendable. This means that Π does *not* have to start all of the rules which are potentially applicable immediately. Since Π does not have a global clock, we do not know when Π should consider applying these left-over rules, and if it starts applying them immediately, it would violate the derivation mode ϑ.

The definitions in Subsection 3.2 address both of these issues by declaring that the only time Π should start new rule applications is when some (other) rule applications release new products. Another way of handling these problems would be restricting per-instance real timing functions to only take values in a closed interval $[c_0; +\infty) \subseteq \mathbb{R}_+$, for some fixed positive constant $c_0 \in R_+$. Under this restriction, we know that any rule takes at least c_0 units of time to finish, which means that Π could reconsider applying new rules either when some rule products become available, or c_0 units of time after the last pack of rule applications started.

Changing the way in which per-instance timing is defined should give rise to formulations of different event-driven semantics. Indeed, with the restriction described in the previous paragraph, the types of events to which Π may react would be extended with the ticks of a "local timer" going off in c_0 units of time after each start of some rule applications.

9 Conclusion and Discussion

In this paper we recalled timed, time-free, and clock-free P systems and provided a common framework for the three notions. This framework allows discussing different kinds of timing functions and freeness properties for P systems operating on arbitrary object types allowing for parallel rule application. We also discussed mode-freeness and showed that, even though mode-freeness and clock-freeness express robustness with respect to variations in quite different parameters, mode-free and clock-free P systems exhibit a number of similarities.

Both mode-freeness and clock-freeness as well as the other concepts like in un-timed and un-clocked P systems seem to offer plenty of possibilities for future research, among which we would like to state the following ones, in no particular order.

1. *Complete Figure 3:* Further investigate the relationship between the freeness properties shown in Figure 3. Find new inclusions, show the (non-)strictness of the known inclusions, consider yet different variations of the timing functions and finishing strategies.
2. *Prove or disprove Conjecture 1:* This conjecture states that that any $\Theta_{\neg deny}$-mode-free P system is *asyn*-deterministic in any reachable configuration C. As shown in Conjecture 2, this could reveal a strong connection between mode-free P systems and clock-free ones.
3. *Other families of modes:* We have only considered two infinite families of evolution modes in detail—the family of all modes Θ_U and the family of non-denying modes $\Theta_{\neg deny}$. We showed that the properties of P systems being mode-free with respect to these families are rather unusual (huge gap between the power

of generators and acceptors). Are there other families of modes exhibiting similar properties?

4. *Halting conditions:* In this paper we essentially glossed over halting conditions. Investigating mode-freeness and clock-freeness with respect to different halting conditions may prove interesting. What would freeness with respect to some families of halting conditions mean?
5. *Mode-freeness without inhibitors:* Theorem 5 shows a computationally complete family of $\Theta_{\neg deny}$-mode-free P systems. These P systems rely on cooperativity *and* on inhibitors (as usual, priorities would work just as well). What are the languages accepted by multiset-rewriting $\Theta_{\neg deny}$-mode-free P systems without inhibitors?
6. *Different clock-freeness:* As pointed out in Section 8, the intuitive idea of allowing individual rule applications to last for a different amount of time gives rise to multiple possible semantics. Subsection 3.2 describes one of them; exploring other possibilities may prove interesting for applications in modelling.
7. *Un-timed and un-clocked P systems:* Which variants of P systems still remain computationally complete when being considered as un-timed or un-clocked systems?

References

1. Artiom Alhazov and Rudolf Freund. Asynchronous and maximally parallel deterministic controlled non-cooperative P systems characterize NFIN and confin. In Erzsébet Csuhaj-Varjú, Marian Gheorghe, Grzegorz Rozenberg, Arto Salomaa, and György Vaszil, editors, *Membrane Computing - 13th International Conference, CMC 2012, Budapest, Hungary, August 28-31, 2012, Revised Selected Papers*, volume 7762 of *Lecture Notes in Computer Science*, pages 101–111. Springer, 2012.
2. Artiom Alhazov, Rudolf Freund, Sergiu Ivanov, and Marion Oswald. Observations on P systems with states. In Marian Gheorghe, Ion Petre, Mario J. Pérez-Jiménez, Grzegorz Rozenberg, and Arto Salomaa, editors, *Multidisciplinary Creativity. Hommage to Gheorghe Păun on His 65th Birthday.* Spandugino, 2015.
3. Artiom Alhazov, Rudolf Freund, and Sergey Verlan. P systems working in maximal variants of the set derivation mode. In Alberto Leporati, Grzegorz Rozenberg, Arto Salomaa, and Claudio Zandron, editors, *Membrane Computing - 17th International Conference, CMC 2016, Milan, Italy, July 25-29, 2016, Revised Selected Papers*, volume 10105 of *Lecture Notes in Computer Science*, pages 83–102. Springer, 2016.
4. Cristian S. Calude and Gheorghe Păun. Bio-steps beyond Turing. *BioSystems*, 77(1-3):175–194, November 2004.
5. Matteo Cavaliere and Dragoş Sburlan. Time–independent p systems. In Giancarlo Mauri, Gheorghe Păun, Mario J. Pérez-Jiménez, Grzegorz Rozenberg, and Arto Salomaa, editors, *Membrane Computing: 5th International Workshop, WMC 2004, Milan, Italy, June 14-16, 2004, Revised Selected and Invited Papers*, pages 239–258. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.
6. Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, 1997.

7. Rudolf Freund. Asynchronous P systems and P systems working in the sequential mode. In Giancarlo Mauri, Gheorghe Paun, Mario J. Pérez-Jiménez, Grzegorz Rozenberg, and Arto Salomaa, editors, *Membrane Computing, 5th International Workshop, WMC 2004, Milan, Italy, June 14-16, 2004, Revised Selected and Invited Papers*, volume 3365 of *Lecture Notes in Computer Science*, pages 36–62. Springer, 2004.
8. Rudolf Freund, Marian Kogler, and Marion Oswald. A general framework for regulated rewriting based on the applicability of rules. In Jozef Kelemen and Alica Kelemenová, editors, *Computation, Cooperation, and Life - Essays Dedicated to Gheorghe Paun on the Occasion of His 60th Birthday*, volume 6610 of *Lecture Notes in Computer Science*, pages 35–53. Springer, 2011.
9. Rudolf Freund and Sergey Verlan. A formal framework for static (tissue) P systems. In George Eleftherakis, Petros Kefalas, Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa, editors, *Membrane Computing*, volume 4860 of *Lecture Notes in Computer Science*, pages 271–284. Springer Berlin Heidelberg, 2007.
10. Daniel T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry*, 81(25):2340–2361, 1977.
11. Sergiu Ivanov. A formal framework for clock-free networks of cells. *International Journal of Computer Mathematics*, 90(4):776–788, 2013.
12. Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa. *The Oxford Handbook of Membrane Computing*. Oxford University Press, Inc., New York, NY, USA, 2010.
13. Grzegorz Rozenberg and Arto Salomaa, editors. *Handbook of Formal Languages, 3 volumes*. Springer, New York, NY, USA, 1997.
14. Dragoş Sburlan. Clock-free P systems. In *Pre-proceedings of the Fifth Workshop on Membrane Computing (WMC5), Milano, Italy, June 2004*, pages 372–383, Milano, Italy, June 2004.
15. Bosheng Song, Mario J. Pérez-Jiménez, and Linqiang Pan. An efficient time-free solution to SAT problem by P systems with proteins on membranes. *J. Comput. Syst. Sci.*, 82(6):1090–1099, 2016.
16. Bosheng Song, Tao Song, and Linqiang Pan. Time-free solution to SAT problem by P systems with active membranes and standard cell division rules. *Natural Computing*, 14(4):673–681, 2015.
17. Bosheng Song, Tao Song, and Linqiang Pan. A time-free uniform solution to subset sum problem by tissue P systems with cell division. *Mathematical Structures in Computer Science*, 27(1):17–32, 2017.
18. Tao Song, Luis F. Macías-Ramos, Linqiang Pan, and Mario J. Pérez-Jiménez. Time-free solution to SAT problem using P systems with active membranes. *Theor. Comput. Sci.*, 529:61–68, 2014.
19. Bulletin of the International Membrane Computing Society (IMCS). `http://membranecomputing.net/IMCSBulletin/index.php`.
20. The P Systems Website. `http://ppage.psystems.eu/`.