Kleene theorems for product systems

Kamal Lodaya[1], Madhavan Mukund[2], and Ramchandra Phawade[1]

[1] The Institute of Mathematical Sciences, CIT Campus, Chennai *600113*, India
[2] Chennai Mathematical Institute, H1, SIPCOT IT Park, Siruseri *603103*, India

Abstract. We prove Kleene theorems for two subclasses of labelled product systems which are inspired from well-studied subclasses of 1-bounded Petri nets. For product T-systems we define a corresponding class of expressions. The algorithms from systems to expressions and in the reverse direction are both polynomial time. For product free choice systems with a restriction of structural cyclicity, that is, the initial global state is a feedback vertex set, going from systems to expressions is still polynomial time; in the reverse direction it is polynomial time with access to an NP oracle for finding deadlocks.

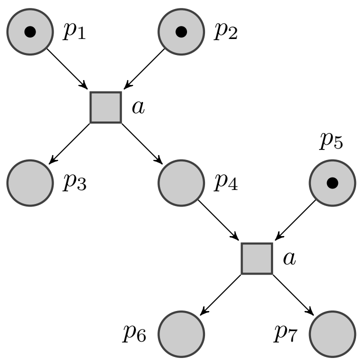
1 Introduction

The descriptional complexity of regular expressions versus automata is well known: the Kleene construction from expressions to automata can be carried out in DLOGSPACE [JR91], while for the converse translation exponential size lower bounds are known [EZ76,GH08]. In this paper we seek to extend these results to the class of 1-bounded free choice Petri net systems and their subclass, 1-bounded T-systems [CHEP71,GL73,Hack72,DE95].

In [Lod06a], we gave a syntactic characterization of Mazurkiewicz's recognizable trace languages [Maz77,Och85,DR95], seen as behaviours of labelled product automata [BeSh83,Arn94]. Our expression syntax was borrowed from Grabowski [Gra81] and Garg and Ragunath [GR92], ultimately deriving from Campbell and Haberman's path expressions [CH74] and Hoare's CSP [Hoa85] by extending regular expressions with a parallel operation (equivalent to shuffle on words), a parallel mixed with intersection (or shuffle with synchronization) and renaming a letter by another.

Zielonka's theorem [Zie87] shows that 1-bounded Petri nets can be viewed as such products. Hence we work with product systems. We are able to extend known constructions on finite automata to syntactically characterize live and structurally cyclic product free choice systems and live product T-systems. The next section gives definitions for these classes, borrowing from Petri nets. Unlike the earlier paper [Lod06a] we do not use Zielonka's theorem in our proofs. Going from automata to expressions we get polynomial time algorithms, which might appear surprising, but the benefit is derived from the condition of structural cyclicity.

In our earlier paper, we used renaming to disambiguate synchronization, but some subtleties remain. For example the system shown in the figure might seem to be described by the expression $(a_1||a_1a_2||a_2)[a/a_1, a/a_2]$, but in fact the example is *not* a labelled product system (see the definitions in the next section) and the expression is not valid, since the labelling, and the renaming operator, have to preserve the process structure: synchronizations between different sets of processes cannot get the same label.



However its behaviour is just the single word aa and the system is a labelled 1-bounded Petri net which even satisfies the "trace labelling" condition of Thiagarajan [Thi96]. Our basic results do not use the renaming operator, consequently we bear the burden of using labellings which preserve process structure.

The largest proper subclass of nets where we are aware of earlier results are on 1-bounded SR-systems [LRR03,RL03], which use an algebraic structure of series and parallel operations [LW00], with only a simple "fork-join" type of synchronization mechanism. Even T-systems are not included in SR-systems since they provide a genuine multi-way synchronization (or "rendezvous") mechanism for communication. The algebraic approach has been very successful at classifying subclasses of regular languages (see [Weil04,TT07] for surveys emphasizing this aspect). However, partially commutative monoids which are used to characterize recognizable trace languages have not so far yielded any results on free choice systems and their subclass of T-systems. Ours is the first work in this direction.

2 Labels and products

Following [Lod06a] we define a **rendezvous alphabet** to be a tuple $(A, |, 0, loc)$ where A is a finite alphabet A containing a dummy action 0, $| : A \times A \to A$ is a commutative and associative **rendezvous** operation over A with 0 as an absorbing element and $loc : A \to \wp(Loc)$ maps actions to locations such that

- $loc(0) = \emptyset$ and $loc^{-1}(\emptyset) = 0$.
- If $loc(a)$ and $loc(b)$ are not disjoint, $a|b = 0$.
- If $loc(a)$ and $loc(b)$ are disjoint and $a|b \neq 0$, $loc(a|b) = loc(a) \cup loc(b)$.
- If $loc(c) = r \cup s$ for disjoint r and s then there exist a, b such that $c = a|b$, with $loc(a) = r, loc(b) = s$.

Actions in A with a single location are called local, and nonzero actions a such that for every b, $a|b = 0$ are global. Thus, a global action is either a local action that does not rendezvous at all, or a "full" synchronization $a_1|a_2| \cdots |a_k$ whose "partial" subsets such as a_3, $a_1|a_k$,... are not global since further rendezvous will be carried out. Let $G(A)$ be the global actions in A.

A **renaming** between rendezvous alphabets is a relation ρ that is Loc-respecting and $|$-stable: that is, if $a\rho b$ then $loc(a) = loc(b)$ and for all c, $(a|c)\rho(b|c)$. The expression in the Introduction does not have a Loc-respecting renaming.

Let ρ be a renaming which is an equivalence relation over alphabet A. $C = (G(A)/\rho, loc)$ is called a **distributed alphabet** where $loc([a]) \overset{\text{def}}{=} loc(a)$ is well defined. We also write ρ as the function $\rho(a) = [a]$.

Definition 1 ([Lod06a]). *Let $A_i \overset{\text{def}}{=} \{a \in A \mid i \in loc(a)\}$. A **product system of automata** N over the alphabet $(A, |, 0, loc)$ is given by automata $N_i = (P_i, p_0^i, \rightarrow_i)$ (called places, initial places and local transitions) over the alphabet A_i, for each i in Loc. We call $\Pi_{i\in Loc} p_0^i$ the initial global state. Given a renaming ρ defining a distributed alphabet C, a **labelled product system of automata** $N[\rho]$ is a product system N over C.*

A product system runs on a word w over the global actions (or $\rho(w)$ over the distributed alphabet) by associating global states from $\Pi_{i\in Loc} P_i$ to prefixes of w: the empty word is assigned the initial global state, and for every prefix va of w, if $\Pi_{i\in Loc} p_i$ is the global state reached after v, then the state $\Pi_{i\in Loc} q_i$ reached after va satisfies, for every $j \in loc(a)$, $p_j \xrightarrow{a}_i q_j$ in M_i, and for every other j, $p_j = q_j$. Thus every action transforms the places of the locations it affects, the other places remaining fixed. We call $t = \Pi_{i\in Loc} p_i \xrightarrow{a}_i q_i$ a global transition if there is some word wa such that t describes the change in global state from w to wa for some run of N on wa.

The **language** of the product system is the set of maximal words (finite or infinite), where a system keeps on running as long as possible and, in addition, each global transition which is infinitely often enabled occurs infinitely often in the run. Since a product system can be simulated by a finite automaton, its accepted language is regular and we call it a recognizable shuffle language [Moh99].

Traces. Let I be an irreflexive symmetric relation over A called independence defined by aIb if $loc(a)$ and $loc(b)$ are disjoint. Let its reflexive transitive closure on A^* be $\sim_I$, called trace congruence. For instance, if aIb then $wabx \sim_I wbax$ (a and b commute).

Notice that if a product system has a run (or an accepting run) on a word $wabx$ and aIb, then it has a run (respectively, an accepting run) on the word $wbax$ as well. Hence a recognizable shuffle language is a recognizable trace language over $(G(A), I)$ in the sense of Mazurkiewicz [Maz77]. Zielonka showed that the converse is not true [Zie87].

Let I be the independence relation over A above extended to C. Using the properties of ρ, the languages accepted by labelled product automata continue to be recognizable trace languages over (C, I). But they need no longer be recognizable shuffle languages. Every recognizable trace language is accepted by a labelled product system.

In a product system, we say the local transition $p \xrightarrow{a}_i q_1$ is **conflict-equivalent** to the local transition $p' \xrightarrow{a}_j q_1'$ if $loc(a) = loc(b)$, for every other local transition

$p \xrightarrow{b}_i q_2$, there is a local transition $p' \xrightarrow{b}_j q_2'$ and, conversely, transitions from p' are matched by transitions from p.

A product system N has a natural representation as a (labelled) 1-safe Petri net (P, T, F, λ, M_0), with places P, net transitions T, flow relation $F \subseteq (P \times T) \cup (T \times P)$, labelling $\lambda : T \to A$ and initial marking $M_0 \subseteq P$, as follows:

- $P = \bigcup_{i \in Loc} P_i$,
- $T = \{t = \Pi_{i \in Loc(a)} p_i \xrightarrow{a}_i q_i \mid t \text{ is a global transition of } N\}$,
- $F = \{(p,t),(t,q) \mid \exists t = \Pi_{i \in Loc(a)} p_i \xrightarrow{a}_i q_i : \exists i \in Loc(a) : p = p_i, q = q_i\}$,
- $\lambda(\Pi_{i \in Loc(a)} p_i \xrightarrow{a}_i q_i) = a$, and
- $M_0 = \bigcup_{i \in Loc} \{p_0^i\}$.

We now borrow some definitions from Petri nets into the framework of product systems [CHEP71,GL73,Hack72,DE95]. First on the structural side, and then on the behavioural:

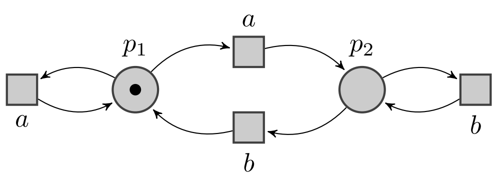
Definition 2. *A product system is **free choice**, more briefly an **FC-product**, if for every a such that $|loc(a)| > 1$, every pair of a-labelled local transitions is conflict-equivalent. We will also use **FC-dag** for FC-products which are acyclic and just **dag** for acyclic and rooted finite automata. A **product T-system (T-product)** is one where every place has at most one input transition and at most one output transition.*

Definition 3. *A global state is **live** if for any run from it and any global transition $t = \Pi_{i \in Loc(a)} p_i \xrightarrow{a}_i q_i$, the run can be extended so that transition t occurs. A product system is **live** if its initial global state is live. It is **deadlock-free** if for every i, from a place p_i in a reachable global state $\Pi_{j \in Loc} p_j$ and a local transition $p_i \xrightarrow{a}_i q_i$, there is a run in which a global transition $\Pi_{j \in Loc} p_j \xrightarrow{a}_j q_j$ occurs.*

The above definition is weaker than the usual one, an empty product system (with no transitions) is deadlock-free. The next definition is new to this paper and identifies a restriction needed for our results. It is stronger than the net-theoretic definition of a net being **cyclic** when its initial global state is a home state [BV84,DE95] (reachable from any reachable global state)—that is, the set of reachable markings is strongly connected. But it is weaker in the sense that acyclic nets are included.

Definition 4. *We say that a product system N is **structurally cyclic** if the initial global state $\Pi_{i \in Loc} p_0^i$ is a feedback vertex set (that is, removing that set of places from N makes the resulting system acyclic).*

The adjacent figure shows a product system (actually an automaton) which is live and 1-bounded. The initial state p_1 is a home state, so the system is cyclic. Removing p_1 does not eliminate all cycles in the reachable global states, so it is not structurally cyclic.



Live and 1-bounded FC-net systems have a characterization [Hack72] which shows that they can be covered by S-components (strongly connected components which are finite automata).

Theorem 5 (Hack). *A live FC-system (N, M_0) is 1-bounded iff it is covered by S-components.*

3 Expressions and languages

Let A be a finite alphabet. We will consider finite as well as infinite words on this alphabet, and languages over them.

For a word w (or any kind of expression defined below), and $a \in A$, $|w|_a$ denotes the number of occurrences of the letter a that appear in w. The alphabet of an expression or word w is $\alpha(w) = \{a \in A \mid |w|_a > 0\}$. The projection $\downarrow$ over the subalphabet $B \subseteq A$ is given as the homomorphism from A^* to B^* which retains all letters in B and deletes all letters outside B. The shuffle of two words $w\|x$ is a language, defined as usual. It is an associative and commutative operation. Now we define the synchronized shuffle over a subalphabet $X \subseteq A$.

Let $w, v \in A^*$ such that $w{\downarrow}X = v{\downarrow}X = a_1 \dots a_n$. Let $w = w_0a_1w_1 \dots a_nw_n$, with $w_i \in (A \setminus X)^*$, and $v = v_0a_1v_1 \dots a_nv_n$, with $v_i \in (A \setminus X)^*$. Then $w\|_Xv$ is defined to be the language $(w_0\|v_0)a_1(w_1\|v_1) \dots a_n(w_n\|v_n)$. If $w{\downarrow}X \neq v{\downarrow}X$, $w\|_Xv$ is undefined.

Definition 6. *Our expressions come in three syntactic sorts:* ***sums, connected expressions*** *and* ***ω-expressions****. If the operation $+$ is not used, we call them* ***T-sequences****,* ***connected T-expressions*** *and* ***ω-T-expressions****, respectively.*

Sums	$s ::= a \in A \vert s_1s_2 \vert s_1 + s_2$
Connected expressions	$c ::= 0 \vert s \vert fsync(c_1, c_2)$
ω-expressions	$e ::= c^\omega \vert par(e_1, e_2)$

For a syntactic expression x, we use $\alpha(x)$ to denote its alphabet—the set of letters of A occurring in x—and $wd(x)$ for its alphabetic width—the total number of occurrences of letters of A in x. For instance, the connected expression $fsync(aabab, abab)$ has an alphabet $\{a, b\}$ and alphabetic width 9.

The semantics of each of these expressions is a language over A. For sums s it is a nonempty language of nonempty finite words, for connected expressions c it is a language of nonempty finite words, for ω-expressions e it is a language of infinite words. The languages are closed under an independence relation which we will define from the expressions themselves, so that we have languages of Mazurkiewicz traces [DR95].

3.1 Sums

To begin with, the language associated with a letter $a \in A$ is $\{a\}$. Formally, $Lang(a) = \{a\}$, $Lang(s_1s_2)$ is the concatenation and $Lang(s_1 + s_2)$ the union, as usual. The alphabet $\alpha(s)$ and projection $s{\downarrow}B$ of a sum are well-defined. We also use the sets $Init(s) \subseteq A$ for the initial actions of a sum, and also Antimirov derivatives $Der_a(s)$ [Ant96]. (Briefly, the Brzozowski a-derivative of $ab + ac$ is the expression $b + c$ [BMc63], the Antimirov a-derivative of $ab + ac$ is the set of expressions $\{b, c\}$.) For the derivatives we will need an extra syntactic entity ϵ standing for the empty word.

3.2 Connected expressions

Now we come to the semantics of connected expressions. Let Loc be the set of all maximal sums occurring in the given expression. Each letter a is located at the sum in which it occurs. We will inductively maintain the following **clustering** property: for initial actions $a, b \in Init(s)$ of a sum s, $loc(a) = loc(b)$. We use $Der_a^{\{l\}}(s)$ instead of $Der_a(s)$ to emphasize that the derivatives are taken at the set of locations $\{l\}$.

Define two occurrences of letters in a connected expression, say a and b, to be independent, if $loc(a)$ and $loc(b)$ are disjoint. (Thus our clustering property implies that initial letters in a sum are dependent.) We define trace equivalence for words over an alphabet with an independence relation, as before. Our semantics for the connected expressions yields unions of trace equivalence classes. Clearly this is so for a sum s since there are no independent letters.

For the connected expression 0, we have $Lang(0) = \alpha(0) = \emptyset$ and $0{\downarrow}B = 0$.

Let c_1, c_2 be connected expressions and L_1, L_2 be disjoint sets of locations such that loc_1 maps c_1 to L_1 and loc_2 maps c_2 to L_2. Their free choice synchronization $fsync(c_1, c_2)$ is over the locations $L_1 \cup L_2$, where for every letter $a \in X = \alpha(c_1) \cap \alpha(c_2)$: $loc(a) = loc_1(a) \cup loc_2(a)$. For the other letters in A, $loc(a)$ is inherited from loc_1 or loc_2, as appropriate. So inductively we have the loc function mapping occurrences of letters in an expression $\alpha(e)$ to nonempty subsets of Loc.

Now we inductively define the derivatives and the semantics of the operation $fsync(c_1, c_2)$. For a letter $a \notin X$ (say a occurs in c_1) we have $Der_a^{L_1 \cup L_2}(fsync(c_1, c_2)) = \{fsync(c', c_2) \mid c' \in Der_a^{L_1}(c_1)\}$, and symmetrically for the case when a appears in c_2. Otherwise suppose that for every sum s in c_1 and c_2, $Init(s) \cap X \neq \phi$ implies $Init(s) \subseteq X$. We say that this synchronization on the common letters X is **clustered** and define $Der_a^{L_1 \cup L_2}(fsync(c_1, c_2)) = \{fsync(c_1', c_2') \mid c_1' \in Der_a^{L_1}(c_1), c_2' \in Der_a^{L_2}(c_2)\}$. It is possible that a synchronization is clustered but has no derivatives, for example in $fsync(ab, ba)$.

We can keep taking derivatives in this fashion for all letters, but only finitely many times since the derivatives become shorter. The number of such derivatives can blow up exponentially in the number of + operators in the expression.

If every synchronization we encounter is clustered we say that the $fsync$ operation itself is clustered and we define its language as:

$$Lang(fsync(c_1, c_2)) = \bigcup \{w_1 \|_X w_2 \mid w_1 \in Lang(c_1),\ w_2 \in Lang(c_2)\}.$$

But it might be that at some derivative of c_1 or c_2, we have a sum s with a choice between an action $a \in Init(s) \cap X$ and an action $b \in Init(s) \setminus X$. (This cannot happen if s is a T-sequence which has $|Init(s)| = 1$.) Then we say that the $fsync$ is not clustered and declare by fiat that $Lang(fsync(c_1, c_2)) = \emptyset$.

A short proof shows that $fsync$ is an associative operation (see, for example, Hoare's CSP [Hoa85]).

The alphabet of $fsync(c_1, c_2)$ is empty if its language is empty and the union of the alphabets of c_1 and c_2 otherwise. In the former case, $fsync(c_1, c_2){\downarrow}B$ is 0, otherwise it is $fsync(c_1{\downarrow}B, c_2{\downarrow}B)$, which is recursively defined.

We call a connected expression **clustered** if every $fsync$ operation in it is clustered. Connected T-expressions are clustered and "deterministic" (there is at most one derivative given an expression and a letter). We will call an FC-product **clustered** if at every reachable global state, if actions a and b are enabled, either they have the same set of locations or they have disjoint locations.

3.3 Omega-power and shuffle

Consider now the expression c^ω. Assume associated with the connected expression c is a function loc over the locations Loc. The independence relation is the one computed for the expression c. $Lang(c^\omega) = [(Lang(c))^\omega]$, the trace closure under our independence relation, where $K^\omega = \{w_1 w_2 \cdots \mid \forall i, w_i \in K\}$. Each equivalence class is a set of infinite words.

Finally the semantics of the par operator is defined to be shuffle of languages.

4 From expressions to product automata

In this section we construct product automata for our syntactic entities. The first result is well known (see, for example, [BeSe86,JR91]).

Lemma 7. *A sum s over A is the language accepted by an acyclic rooted finite automaton, which we call a* ***dag****. (In case s is a T-sequence, the automaton consists of a single directed path.) The size of the automaton is $O(wd(s))$ (for example, using derivatives as states), and it can be computed in linear time and* DLOGSPACE.

4.1 Connected expressions

Now we come to connected expressions, for which we will construct a product of automata. Before that we look for deadlocks.

Lemma 8. *The emptiness of the language of a connected expression c can be checked in* NP*, and of a connected T-expression in time $O(wd(c)^2)$.*

Proof. The complexity bound for a connected T-expression holds because we track at most $wd(c)$ tokens (represented by pointers in the expression) through a word of length at most $wd(c)$ to determine whether we reach the end of each T-sequence. This does not work for connected expressions: for example, $fsync(ab+ac, ad+ae+af)$ has six runs beginning with a in the resultant product. Now we use nondeterminism to guess the word letter-by-letter and move tokens. On any letter, if there is a derivative in one component of an $fsync$ but none in another, we have a deadlock. □

Lemma 9. *Let c be a connected expression. Then there exists a deadlock-free clustered connected FC-dag (N, M_0) accepting $Lang(c)$ which is covered by a set Loc of dags. The size of the constructed system is $O(wd(c))$ and it can be computed using a polynomial time algorithm with access to an* NP *oracle. For connected T-expressions, the time bound is $O(wd(c)^2)$.*

Proof. We use the preceding lemma to check for a deadlock using an NP oracle query. If the answer is yes, we return the empty product system, covered by the empty set of dags!

If the oracle says there is no deadlock, we use Lemma 7 as the base case of an induction producing a deadlock-free clustered FC-dag that is covered by dags. We let the set Loc stand for these dags. Thus, for a dag constructed above, we pick a fresh location $l \in Loc$ and we locate every letter a labelling a transition in the dag by $loc(a) = \{l\}$. As an aside, note that the generated independence relation is empty, every transition in the path is dependent on the other transitions and on itself.

Inductively, consider connected expressions c_1 and c_2, and assume we have corresponding deadlock-free clustered FC-dags, covered by the dags in L_1 and L_2, with the independence relation of the FC-dags matching those of the expressions.

For the expression $fsync(c_1, c_2)$, we construct using the derivatives the FC-dag that is the synchronization of these two FC-dags, which are assumed to be covered by disjoint dags. The resulting automaton will be acyclic and covered by the dags $L_1 \cup L_2$. Because of the clustering property of the expressions, the resulting FC-dag will be clustered. Its size is $O(wd(c_1) + wd(c_2))$.

The trace equivalence generated from the locations is such that the language K of the constructed automaton is trace-closed. We can now verify that K is obtained by performing the $fsync$ operation on the languages of the two component automata. □

4.2 Omega-power and shuffle

For the expression c^ω we map in linear time the ω-power operation to the construction of an FC-product.

Lemma 10. *Let $e = c^\omega$ be an expression over alphabet A with $Lang(c)$ a nonempty language of nonempty words. Then there exists a live and structurally cyclic FC-product accepting $Lang(e)$. The size of the constructed system is $O(wd(c))$ and it can be computed using a polynomial time algorithm with access to an* NP *oracle. For ω-T-expressions, the time bound is $O(wd(c)^2)$.*

Proof. For the expression c^ω, consider the deadlock-free clustered connected FC-dag N for c, covered by the dags in Loc and accepting the language K, obtained from the previous lemma. Recall that the trace equivalence generated from the independence relation of N saturates K, that is, $K = [K]$.

For each dag $l \in Loc$, we fuse the initially marked places of l with its sink places (which are different since K does not have the empty word). Call the new product system N'. The product satisfies the following properties:

(1) Each node of N was covered by some dag $l \in Loc$. So N' is an FC-product.
(2) It is structurally cyclic since by construction the initial global state is a feedback vertex set.
(3) Fusing the sink and source places makes each dag of N strongly connected in N', in fact a strongly connected component of N', since N was connected and deadlock-free. By Theorem 5, N' is live.

We now show that the language of N' is $Lang(e) = [Lang(c)^\omega]$.

By construction $K^\omega \subseteq Lang(N')$. Since N' has the same locations as N, it generates the same trace equivalence and hence we have that $[K^\omega] \subseteq [Lang(N')] = Lang(N')$.

To prove the converse inclusion, $Lang(N') \subseteq [K^\omega]$, suppose not and we have w accepted by N' but not in $[K^\omega]$. We can remove prefixes of w which are in $[K]$, so let us assume $w = uav$, u is a proper prefix of K and ua is not a prefix of a word in $[K^\omega]$. Since N was deadlock-free, there is some extension ub that is a prefix of K such that b is enabled after executing u. If a and b are dependent and they are both enabled, in a clustered FC-dag they have the same locations, and ua would be a prefix of K as well. Hence a and b are independent and we can commute them. We apply this argument repeatedly to increase the length of the prefix; but since K is a finite language, after some point we will find that $w = uav$ for some $u \in [K]$ after which a is enabled, hence a is enabled at the initial global state of N. We can remove this prefix and again continue the argument. This shows that w is in $[K^\omega]$, a contradiction. $\square$

For the expression $par(e_1, e_2)$, all occurrences of letters in e_1 are independent of those in e_2, so that the net corresponding to them is obtained by taking the disjoint union of the two subnets, and its language is the shuffle of the two sublanguages. Clearly the size of the constructed system is $O(wd(e_1)) + O(wd(e_2))$. So we conclude:

Theorem 11. *For every ω-expression e, there is a live and structurally cyclic FC-product of size $O(wd(e))$ accepting $Lang(e)$.*

We put this together with our earlier result on connected expressions. The FC-products constructed are not necessarily live.

Corollary 12. *For every expression e which is a shuffle of connected expressions and ω-expressions, there is a structurally cyclic FC-product accepting $Lang(e)$. Further, the emptiness of the language of such expressions can be checked in* NP. *For ω-T-expressions, the time complexity is $O(wd(e)^2)$.*

5 FC-products to expressions

In this section we discuss how to build language equivalent expressions for a given FC-product. We follow the same strategy as in the previous section, working through dags and FC-dags before tackling the general case.

Lemma 13. *Let $N = (P, \rightarrow, p_0)$ be a dag. Then there exists an equivalent sum s for its language. The alphabetic width of this expression is quadratic in N and it can be computed in time quadratic in N.*

Proof. First, we delete all nodes unreachable from p_0 and then apply Kleene's theorem. Each transition appears in a path and the length of each path is linear in N which gives a quadratic upper bound. $\square$

Next, we construct expressions for FC-dags. We do not check whether the expression has deadlocks.

Lemma 14. *Let $N[\rho]$ be a connected FC-dag. There is a connected clustered expression c for $Lang(N[\rho])$ of alphabetic width $O(|N|^2)$ which can be computed in $O(|N|^3)$ time.*

Proof. Using Lemma 13, we obtain in quadratic time equivalent sum expressions s_i of size quadratic in the alphabetic width, for each component of the product. The renaming ρ satisfies the property that transitions labelled the same have the same locations. Hence we can consider the expression formed by taking $fsync$ expressions of the s_i, taken in some order. Since the FC-dag was a free choice net, each synchronization will be clustered. Its alphabetic width is quadratic in the size of N. $\square$

Finally we have a cubic time algorithm from live structurally cyclic FC-products to ω-expressions.

Theorem 15. *Let $N[\rho]$ be a live, structurally cyclic product FC-automaton. Then we can compute in cubic time an ω-expression of alphabetic width $O(|N|^2)$ for the accepted language.*

Proof. Consider $N[\rho]$ a given live, structurally cyclic FC-product. We first divide it up into strongly connected components and deal with them separately. This can be done in time $O(|N|)$.

Now we adopt a small trick. Make a copy P_0' of the places P_0 in the initial global state and change the system so that the edges coming into P_0 are replaced by edges into the corresponding places of P_0'. Since P_0 is a feedback vertex set, the resulting net system $N'[\rho]$ is a connected deadlock-free FC-dag of size $O(|N|)$.

By Lemma 14 we can compute in $O(|N|^3)$ time a connected expression c of alphabetic width $O(|N|^2)$ for this FC-dag. We claim the expression c^ω describes the language of the original net system (N, M_0). The proof follows the same arguments as in Lemma 10.

For each SCC, use the argument above, and then use the *par* operator to obtain the shuffle of the languages. This preserves both the time complexity and the expression's alphabetic width. $\square$

We can extend the result above to deal with product systems which are not necessarily live, but structural cyclicity is crucially used. The constructed expression is not checked for deadlocks.

Corollary 16. *Let $N[\rho]$ be a structurally cyclic product FC-system. Then we can compute in polynomial time a shuffle of connected and ω-expressions, of alphabetic width polynomial in $|N|$, for the accepted language.*

Proof. If a dag synchronizes with a strongly connected product, the initial global state of the resulting system will not be live. We unfold the given product into a dag. For example, if path abc synchronizes with circuit $debfcgh$ on b and c, we replace the circuit by the path $debfcghde$ where the d and e transitions can occur twice, but the synchronizations occur once.

When repeating this process, some synchronizations might occur more than once. For example, if the erstwhile circuit $debfcgh$ synchronized with circuit $xyzd$ on d, the second circuit is now replaced by path $xyzdxyzdxyz$ with two occurrences of the synchronization d.

Using this idea, we can modify the algorithm in the proof of Theorem 15 to first cover the reachable parts of the given product system with dags and strongly connected components, then convert the non-live part into equivalent dags, finally obtaining an FC-product which is divided into connected dags which are not live (which might blow up this part of the net to a size $O(|N|^2)$), and live strongly connected components with a feedback vertex set as initial state, which are not modified. Now we use the two preceding theorems to provide connected expressions and ω-expressions. The final expression is a shuffle of these. $\square$

Finally, the algorithms of this section can be seen to produce T-sequences, connected T-expressions and ω-T-expressions in case we are given a product T-system which is path-like, a T-dag and live, respectively, since T-systems are structurally cyclic. Thus we have efficient Kleene characterizations for product T-systems as well.

6 Conclusion

In this paper we have shown Kleene theorems for the class of acyclic product T-systems and live product T-systems, and also over the corresponding subclasses of FC-products with a restriction to structural cyclicity. Using in one direction a Berry-Sethi type algorithm [BeSe86] we have obtained a polynomial time algorithm with access to an NP oracle. In the other direction we have made use of the condition of structural cyclicity to obtain a polynomial time algorithm.

There are several avenues for further research. Perhaps the complexity can be improved from $\mathrm{FP}^{\mathrm{NP}}$. We would like to extend our work to deal with cyclic free choice product systems; from a conceptual viewpoint we are interested in seeing if the polynomial time reachability algorithms for live, cyclic and 1-bounded free choice nets can fall out of this kind of algebraic structure. It is not clear if the same idea extends to all 1-bounded free choice nets.

References

[Ant96] V. Antimirov. Partial derivatives of regular expressions and finite automaton constructions, *Theoret. Comput. Sci.* 155, 1996, 291–319.

[Arn94] A. Arnold. *Finite transition systems*, Prentiece-Hall, 1994.

[BeSe86] G. Berry and R. Sethi. From regular expressions to deterministic automata, *Theoret. Comp. Sci.* 48:3, 1986, 117–126.

[BeSh83] E. Best and M. Shields. Some equivalence results for free choice and simple nets and on the periodicity of live free choice nets, *Proc. CAAP*, L'Aquila (G. Ausiello and M. Protasi, eds.), *LNCS* 159, 1983, 141–154.

[BV84] E. Best and K. Voss. Free choice systems have home states, *Acta Inform.* 21, 1984, 89–100.

[BMc63] J.A. Brzozowski and E.J. McCluskey. Signal flow graph techniques for sequential circuit state diagrams, *IEEE Trans. Electr. Comput.* EC-12, 1963, 67–76.

[CH74] R.H. Campbell and A.N. Habermann. The specification of process synchronization by path expressions, in *Proc. Operating Systems conference* (E. Gelenbe and C. Kaiser, eds.), *LNCS* 16, 1974, 89–102.

[CHEP71] F. Commoner, A.W. Holt, S. Even and A. Pnueli. Marked directed graphs, *J. Comp. Syst. Sci.* 5:5, 1971, 511–523.

[DE95] J. Desel and J. Esparza. *Free choice Petri nets*, Cambridge, 1995.

[DR95] V. Diekert and G. Rozenberg, eds. *The book of traces*, World Scientific, 1995.

[EZ76] A. Ehrenfeucht and P. Zeiger. Complexity measures for regular expressions, *J. Comp. Syst. Sci.* 12, 1976, 134–146.

[GR92] V.K. Garg and M.T. Ragunath. Concurrent regular expressions and their relationship to Petri nets, *Theoret. Comp. Sci.* 96:2, 1992, 285–304.

[GL73] H.J. Genrich and K. Lautenbach. Synchronisationsgraphen, *Acta Inform.* 2, 1973, 143–161.

[Gra81] J. Grabowski. On partial languages, *Fund. Inform.* IV:2, 1981, 427–498.

[GH08] H. Gruber and M. Holzer. Finite automata, digraph connectivity and regular expression size, *Proc. 35th ICALP*, Reykjavik (L. Aceto, I. Damgård, L.A. Goldberg, M.M. Halldórsson, A. Ingólfsdóttir and I. Walukiewicz, eds.), *LNCS* 5126, 2008, 39–50.

[Hack72] M.H.T. Hack. Analysis of production schemata by Petri nets, *Project MAC Report* TR-94, MIT, 1972.

[Hoa85] C.A.R. Hoare. *Communicating sequential processes*, Prentice-Hall, 1985.

[JR91] T. Jiang and B. Ravikumar. A note on the space complexity of some decision problems for finite automata, *Inf. Proc. Lett.* 40:1, 25–31, 1991.

[Kle56] S.C. Kleene. Representation of events in nerve nets and finite automata, in *Automata studies* (C.E. Shannon and J. McCarthy, eds.), Princeton, 1956, 3–41.

[Lod06a] K. Lodaya. Product automata and process algebra, *Proc. 4th SEFM*, Pune (P.K. Pandya and D.v.Hung, eds.), IEEE, 2006, 128–136.

[Lod06b] K. Lodaya. A regular viewpoint on processes and algebra, *Acta Cybernetica* 17:4, 2006, 751–763.

[LRR03] K. Lodaya, D. Ranganayukulu and K. Rangarajan. Hierarchical structure of 1-safe nets, *Proc. 8th Asian*, Mumbai (V.A. Saraswat, ed.), *LNCS* 2896, 2003, 173–187.

[LW00] K. Lodaya and P. Weil. Series-parallel languages and the bounded-width property, *Theoret. Comp. Sci.* 237:1-2, 2000, 347–380.

[Maz77] A. Mazurkiewicz. Concurrent program schemes and their interpretations, *DAIMI Report* PB-78, Aarhus University, 1977.

[Mil89] R. Milner. *Communication and concurrency*, Prentice-Hall, 1989.

[Moh99] S.K. Mohalik. *Local presentations for finite state distributed systems*, PhD thesis, University of Madras, 1999.

[MR02] S. Mohalik and R. Ramanujam. Distributed automata in an assumption-commitment framework, *Sādhanā* 27,part 2, April 2002, 209–250.

[MS97] M. Mukund and M.A. Sohoni. Keeping track of the latest gossip in a distributed system, *Distr. Comp.* 10:3, 1997, 117–127.

[Och85] E. Ochmański. Regular behaviour of concurrent systems, *Bull. EATCS* 27, 1985, 56–67.

[Pet62] C.-A. Petri. Fundamentals of a theory of asynchronous information flow, *Proc. IFIP*, Munich (C.M. Popplewell, ed.), North-Holland, 1962, 386–390.

[RL03] D. Ranganayukulu and K. Lodaya. Infinite series-parallel posets of 1-safe nets, *Proc. Algorithms and Artificial Systems*, Chennai (P. Thangavel, ed.), Allied, 2003, 107–124.

[TT07] P. Tesson and D. Thérien. Logic meets algebra: the case of regular languages, *Log. Meth. Comput. Sci.* 3:1, 2007.

[TV84] P.S. Thiagarajan and K. Voss. A fresh look at free choice nets, *Inf. Contr.* 61:2, 1984, 85–113.

[Thi96] P.S. Thiagarajan. Regular trace event structures, *BRICS Report* RS-96-32, Dept of Computer Science, Aarhus University, 1996.

[Weil04] P. Weil. Algebraic recognizability of languages, *Proc. MFCS*, Prague (J. Fiala, V. Koubek and J. Kratochvíl, eds.), *LNCS* 3153, 2004, 149–175.

[Zie87] W. Zielonka. Notes on finite asynchronous automata, *RAIRO Inform. Th. Appl.* 21:2, 1987, 99–135.