The Transformation Logics

Alessandro Ronca
University of Oxford
alessandro.ronca@cs.ox.ac.uk

Abstract

We introduce a new family of temporal logics designed to finely balance the trade-off between expressivity and complexity. Their key feature is the possibility of defining operators of a new kind that we call *transformation operators*. Some of them subsume existing temporal operators, while others are entirely novel. Of particular interest are transformation operators *based on semigroups*. They enable logics to harness the richness of semigroup theory, and we show them to yield logics capable of creating *hierarchies of increasing expressivity and complexity* which are non-trivial to characterise in existing logics. The result is a genuinely novel and yet unexplored *landscape of temporal logics*, each of them with the potential of matching the trade-off between expressivity and complexity required by specific applications.

1 Introduction

We introduce the *Transformation Logics*, a new family of temporal logics designed to finely balance the trade-off between expressivity and complexity. Their key feature is the possibility of defining operators of a new kind that we call *transformation operators*. They capture patterns over sequences, and they can be thought of as a generalisation of temporal operators. The subclass of transformation operators based on *finite semigroups* is of particular interest. Such *semigroup-like* operators suffice to capture all regular languages, and remarkably they allow for creating *hierarchies of increasing expressivity and complexity* which are non-trivial to define in existing logics. The base level of such hierarchies is obtained using the operator defined by the *flip-flop monoid*. The other levels are obtained introducing operators based on *simple groups*—the building blocks of all groups. Simple groups have been systematically classified into a finite number of families, cf. [Gorenstein *et al.*, 2018]. The classification provides a compass in the landscape of groups, and a roadmap in the exploration of temporal logics, as it is made clear by the results in this paper.

Our motivation arises from the usage of temporal logics in AI. They are used in *reinforcement learning* to specify reward and dynamics functions [Bacchus *et al.*, 1996; Brafman *et al.*, 2018; Icarte *et al.*, 2018; Camacho *et al.*, 2019; De Giacomo *et al.*, 2020b; De Giacomo *et al.*, 2020a]; in *planning* for describing temporally-extended goals [Torres and Baier, 2015; Camacho *et al.*, 2017; De Giacomo and Rubin, 2018; Brafman and De Giacomo, 2019; Bonassi *et al.*, 2023]; in *stream reasoning* to express programs with the ability of referring to different points of a stream of data [Beck *et al.*, 2018; Ronca *et al.*, 2022; Walega *et al.*, 2023].

In the above applications, the required trade-off between expressivity and complexity will depend on the case at hand. When the basic expressivity of the star-free regular languages suffices, one can employ logics such as Past LTL [Manna and Pnueli, 1991] and LTLf [De Giacomo and Vardi, 2013]. In all the other cases, one needs to resort to more expressive logics. The existing extensions of the above logics have the expressivity of all regular languages, cf. ETL [Wolper, 1983] and LDLf [De Giacomo and Vardi, 2013]. This is a big leap in expressivity, which may incur an unnecessarily high computational complexity. We show next two examples where the required expressivity lies in fragments between the star-free regular languages and all regular languages. These fragments can be precisely characterised in the Transformation Logics.

Example 1. *An agent is assigned a task that can be completed multiple times. We receive an update every minute telling us whether the agent has completed the task in the minute that has just elapsed. We need to detect whether the agent has completed the task at least once on every past day.*

The example describes a periodic pattern, which is beyond the star-free regular languages. It requires to count minutes modulo $24 * 60 = 1440$ in order to establish the end of every day. This can be expressed in the Transformation Logics using a transformation operator defined by the cyclic group C_{1440}. Cyclic group operators yield an ability to capture many useful periodic patterns. At the same time, they belong to the special class of *solvable group operators*, which enjoys good properties such as a more favourable computational complexity compared to larger classes of operators. The next one is an example where solvable group operators do not suffice, and we need to resort to *symmetric group operators*.

Example 2. *A cycling race with n participants takes place, and we need to keep track of the live ranking. At each step an overtake can happen, in which case it is communicated to us in the form $(i, i+1)$ meaning that the cyclist in position i*

has overtaken the one in position $i+1$. We know the initial ranking, and we need to keep track of the live ranking.

The ranking in the example corresponds to the symmetric group S_n, which is not solvable if $n \geq 5$. The example can be specified in a Transformation Logic featuring a transformation operator defined by the group S_n.

Summary of the contribution. We introduce the Transformation Logics, providing a formal syntax and semantics. Their main characteristic is the transformation operators. The operators are very general, as we demonstrate through a series of concrete examples. We develop a systematic approach in defining operators, based on semigroup theory and algebraic automata theory, cf. [Ginzburg, 1968; Arbib, 1969; Dömösi and Nehaniv, 2005]. This way we are able to identify *prime operators* that can capture all finite operators. For them, we prove a series of expressivity and complexity results. Regarding the *expressivity*, we show there exists one operator, defined by the *flip-flop monoid*, which yields the expressivity of the *star-free regular languages*; as one keeps adding operators based on *cyclic groups* of prime order, the expressivity increases, up to capturing all languages that can be captured using *solvable group operators*; the expressivity of all regular languages is reached by adding the other prime operators, that can be defined by choosing groups from *the classification of finite simple groups*, cf. [Gorenstein *et al.*, 2018]. Regarding the *complexity*, we focus on the *evaluation problem*, and we show three sets of results. First, we show any Transformation Logic can be evaluated in *polynomial time*, whenever its operators can be evaluated in polynomial time. Second, for two notable families of operators, we show that polynomial-time evaluation is possible even when they are represented compactly. Third, we focus on the *data complexity* of evaluation showing that it corresponds to the three circuit complexity classes $\mathrm{AC}^0 \subsetneq \mathrm{ACC}^0 \subseteq \mathrm{NC}^1$ when we include (i) only the flip-flop operator, (ii) also cyclic operators, and (iii) all operators. Finally, we show how Past LTL formulas can be easily translated into the core Transformation Logic featuring the flip-flop operator.

Extended version. Proofs of all our results as well as additional details on several aspects of the paper can be found in the extended version [Ronca, 2024].

2 Preliminaries

For X a set, a *transformation* is a function $f : X \to X$. We write the identity function over any domain as id. We denote the Boolean domain $\{0,1\}$ by $\mathbb{B}$, the natural numbers by $\mathbb{N}$, and the integer numbers by $\mathbb{Z}$.

2.1 Formal Languages

An *alphabet* Σ is a non-empty finite set of elements called *letters*. A *string* over Σ is a finite concatenation $\sigma_1 \cdots \sigma_n$ of letters from Σ. A *language* over Σ is a set of strings over Σ. The *regular languages* are the ones languages that are defined by regular expressions, or equivalently by finite automata [Kleene, 1956]. The *star-free regular languages* are the languages that are defined by regular expressions without the Kleene star but with complementation, or equivalently by a group-free finite automaton, cf. [Ginzburg, 1968].

2.2 Propositional Logic

Syntax. A *propositional variable* is an element from a set $\mathcal{V}$ that we consider as given. Typically we denote propositional variables by lowercase Latin letters. A *propositional formula* is built out of propositional variables and the *Boolean operators* $\{\neg, \wedge, \vee\}$. It is defined inductively as a propositional variable or one of the following expressions: $\neg\alpha$, $\alpha \wedge \beta$, $\alpha \vee \beta$ where α and β are propositional formulas. Additional Boolean operators may be defined, but it is not necessary as the former operators are *universal*, they suffice to express all Boolean functions.

Semantics. An *interpretation I for a propositional formula* is a subset of the propositional variables occurring in the formula. Intuitively, the fact that a variable appears in the intepretation means that the variable stands for a proposition that is true. An *assignment* is a function $\nu : V \to \mathbb{B}$ from a set of propositional variables V to the Boolean domain $\mathbb{B} = \{0,1\}$. When $V = \{v_1, \ldots, v_n\}$, we can also write an assingment ν as the map $\langle v_1, \ldots, v_n \rangle \mapsto \langle b_1, \ldots, b_n \rangle$, with the meaning that $\nu(v_i) = b_i$. An interpretation I corresponds to the assignment ν such that $\nu(a) = 1$ iff $a \in I$. Then, the semantics of formulas is defined in terms of the following satisfiability relation. Given a formula α and an interpretation I for α, the *satisfiability relation* $I \models \alpha$ is defined following the structural definition of formulas, for variables as

- $I \models a$ iff $a \in I$,

and inductively for the other formulas as

- $I \models \neg\alpha$ iff $I \not\models \alpha$,
- $I \models \alpha \vee \beta$ iff $I \models \alpha$ or $I \models \beta$,
- $I \models \alpha \wedge \beta$ iff $I \models \alpha$ and $I \models \beta$.

An assignment ν to variables $\{a_1, \ldots, a_m\}$ can be seen as the conjunction $l_1 \wedge \cdots \wedge l_m$ where $l_i = a_i$ if $\nu(a_i) = 1$ and $l_i = \neg a_i$ otherwise. This allows us to write $I \models \nu$.

2.3 Semigroups and Groups

A *semigroup* is a non-empty set together with an *associative* binary operation that combines any two elements a and b of the set to form a third element c of the set, written $c = (a \cdot b)$. A *monoid* is a semigroup that has an *identity element* e, i.e., $(a \cdot e) = (e \cdot a) = a$ for every element a. The identity element is unique when it exists. A *group* is a monoid where every element a has an *inverse* b, i.e., $(a \cdot b) = (b \cdot a) = e$ where e is the identity element. For every element a of a group, its inverse is unique and it is denoted by a^{-1}. A *subsemigroup* (*subgroup*) of a semigroup S is a subset of S that is a semigroup (group). The *order* of a semigroup is the number of elements. For S and T semigroups, we write $ST = \{s \cdot t \mid s \in S, t \in T\}$; we also write $S^1 = S$ and $S^n = SS^{n-1}$. A semigroup S is *generated* by a semigroup T if $S = \bigcup_n T^n$. A *homomorphism* from a semigroup S to a semigroup T is a mapping $\psi : S \to T$ such that $\psi(s_1 \cdot s_2) = \psi(s_1) \cdot \psi(s_2)$ for every $s_1, s_2 \in S$. If ψ is bijective, we say that S and T are *isomorphic*. Isomorphic semigroups are considered identical. Let G be a group and let H be a subgroup of G. A *right coset* of G is $g \cdot H = \{g \cdot h \mid h \in H\}$ for $g \in G$, and a *left coset* of G is $H \cdot g = \{h \cdot g \mid h \in H\}$ for $g \in G$. Subgroup H is

normal if its left and right cosets coincide. A group is *trivial* if it is the singleton $\{e\}$. A *simple group* is a group G such that every normal subgroup of G is either trivial or G itself. For $g, h \in G$, the *commutator* of g and h is $g^{-1} \cdot h^{-1} \cdot g \cdot h$. The *derived subgroup* of G is the subgroup generated by its commutators. Setting $G^{(0)} = G$, the n-th derived subgroup $G^{(n)}$ is the derived subgroup of $G^{(n-1)}$. A group is *solvable* if there exists n such that $G^{(n)}$ is trivial.

A *flip-flop monoid* is a three-element monoid $\{s, r, e\}$ where $(r \cdot s) = s$, $(s \cdot s) = s$, $(r \cdot r) = r$, and $(s \cdot r) = r$. All flip-flop monoids are isomorphic, and hence one refers to *the* flip-flop monoid. A *cyclic group* is a group that is isomorphic to the group C_n of integers $\{0, \ldots, n-1\}$ with modular addition $i \cdot j = i + j \bmod n$. Again, one refers to any cyclic group of order n as *the* cyclic group C_n. Two relevant properties of cyclic groups are: (i) a cyclic group C_n is simple iff n is a prime number; (ii) every cyclic group is solvable.

3 The Transformation Logics

We introduce the Transformation Logics. They are a propositional formalism, atoms are variables standing for propositions that can be true or false. The truth of some variables is given as input, whereas the meaning of other variables is given by a *definition*. Definitions allow us to avoid nested expressions, and hence they aid intelligibility in this context. A definition features either a Boolean expression, the delay operator, or a transformation operator. The delay operator is akin to the before operator from Past LTL. A transformation operator has a domain of elements and a set of transformations over the domain—a *transformation* is a map from elements of a domain to elements of the same domain. At every step a transformation operator has an associated domain element to which it applies a transformation based on the truth value of the operands. Then the evaluation value is a function of the current domain element. Each transformation corresponds to a specific functionality. For example, setting a bit to one, or increasing a count. More intuition is given below in the Paragraph 'Explanation' and later in Section 3.1.

Syntax. A *static definition* is

$$p \coloneqq \alpha$$

where p is a propositional variable, and α is a propositional formula. A *delay definition* is

$$p \coloneqq \mathcal{D}\, q$$

where p and q are propositional variables, and $\mathcal{D}$ is called the *delay operator*. A *transformation operator* $\mathcal{T}$ is a tuple $\langle X, T, \phi, \psi \rangle$ consisting of a non-empty set X, a non-empty set T of transformations $\tau : X \to X$, a function $\phi : \mathbb{B}^m \to T$, and a function $\psi : X \to \mathbb{B}^n$. We call X the *transformation domain*; we call m and n the *input* and *output arity*, respectively. An operator is *finite* if its transformation domain is finite—in which case all its other components are necessarily finite. A *transformation definition* is

$$p_1, \ldots, p_n \coloneqq \mathcal{T}(q_1, \ldots, q_m \mid x_0)$$

where $\mathcal{T}$ is a transformation operator with input arity m and output arity n; $p_1, \ldots, p_n$ and $q_1, \ldots, q_m$ are variables; and $x_0 \in X$ is the initial domain element. A *definition* is either a static definition, a delay definition, or a transformation definition. In a definition, the expression on the left of '$\coloneqq$' is called *head*, and the expression on the right is called *body*. A *program* is a finite set of definitions. A *query* is a pair (P, q) consisting of a program P and a variable q occurring in P. Programs are required to be *nonrecursive*, i.e., to have an acyclic dependency graph. The *dependency graph* of a program has one node for each variable in the program, and it has a directed edge from a to b if there is a definition where a is in the body and b is the head. Programs are also required to define variables at most once, i.e., every variable p occurs at most once in the head of a definition; if p occurs in the head of a definition d, we say that d *defines* p. In a given program P, a variable is a *defined variable* if there is a definition in P that defines it, and it is an *input variable* otherwise.

For $\mathbf{T}$ a set of transformation operators, the *Transformation Logic* $\mathcal{L}(\mathbf{T})$ is the set of programs consisting of all static definitions, all delay definitions, and all transformation definitions with operators from $\mathbf{T}$.

Semantics. An *input* to a program P is a finite non-empty sequence of subsets of the input variables of P. An *assignment to a transformation definition* d is an expression $d \mapsto x$ with x a domain element of the operator of d. The semantics of programs is defined in terms of the following *satisfiability relation*. Given a program P, an input $I = I_1, \ldots, I_\ell$ to P, and an index $t \in [1, \ell]$, we define the satisfiability relation $(P, I, t) \models E$ where E is a propositional variable, a propositional formula, or an assignment to a transformation definition. We assume definitions are in the form given above; it allows us to refer to the symbols mentioned there, e.g., symbol p_i for the i-th head variable of a transformation definition.

1. For a an input variable,
   - $(P, I, t) \models a$ iff $a \in I_t$;
2. For α and β formulas,
   - $(P, I, t) \models \neg\alpha$ iff $(P, I, t) \not\models \alpha$;
   - $(P, I, t) \models \alpha \vee \beta$ iff $(P, I, t) \models \alpha$ or $(P, I, t) \models \beta$;
   - $(P, I, t) \models \alpha \wedge \beta$ iff $(P, I, t) \models \alpha$ and $(P, I, t) \models \beta$;
3. For p a variable defined by a static definition,
   - $(P, I, t) \models p$ iff $(P, I, t) \models \alpha$;
4. For p a variable defined by a delay definition,
   - $(P, I, t) \models p$ iff $(P, I, t-1) \models q$;
5. For $d \mapsto x$ an assignment to a transformation definition d,
   - $(P, I, 0) \models (d \mapsto x)$ iff $x = x_0$;
   - $(P, I, t) \models (d \mapsto x)$ iff
     - $(P, I, t-1) \models (d \mapsto x')$,
     - $(P, I, t) \models (\langle q_1, \ldots, q_m \rangle \mapsto \mu)$, and
     - $\tau(x') = x$ with $\tau = \phi(\mu)$;
6. For p_i a variable defined by a transformation definition d,
   - $(P, I, t) \models p_i$ iff
     - $(P, I, t) \models (d \mapsto x)$, and
     - $\psi(x) = \langle b_1, \ldots, b_i, \ldots, b_n \rangle$ with $b_i = 1$.

Explanation. The index t can be thought of as a time point, ranging over the positions of the given input I. Points 1 and 2 follow the standard semantics of propositional formulas, evaluated with respect to the assignment I_t. Points 3–5 define the semantics of definitions, relying on the auxiliary Point 6. Point 3 is the semantics of static definitions. It says that the truth value of a variable p defined by a static definitions is the truth value of the propositional formula α corresponding to the body of the definition, evaluated at the same time point. Point 4 is the semantics of delay definitions. It says that the truth value of a variable p defined by a delay definition is the truth value of the propositional variable q occurring in the body the definition, evaluated at the *previous* time point. Point 5 describes the element x that is currently associated with a transformation definition. For $t = 0$, the element is x_0, the initial element specified in the definition. For $t > 0$, the element x is determined as follows. We pick an assignment $\langle q_1, \ldots, q_m \rangle \mapsto \mu$ for the body variables of the definition. Specifically, $(P, I, t) \models (\langle q_1, \ldots, q_m \rangle \mapsto \mu)$ with $\mu = \langle b_1, \ldots, b_m \rangle$ means that $(P, I, t) \models q_i$ if $b_i = 1$ and $(P, I, t) \not\models q_i$ otherwise. The assignment determines the transformation $\tau = \phi(\mu)$, which in turn determines the next element $x = \tau(x')$ from the previous one x'. Point 6 defines the semantics of transformation definitions. It specifies when variables $p_1, \ldots, p_n$ defined by a transformation definition d are true. The semantics is defined considering a single variable p_i at a time, considered as part of the former list. There is an element x of the transformation domain of $\mathcal{T} = \langle X, T, \phi, \psi \rangle$ that is currently associated with the definition d. Namely, the condition $(P, I, t) \models (d \mapsto x)$ holds. Hence, the assignment $\langle b_1, \ldots, b_n \rangle$ to the variables $p_1, \ldots, p_n$ is given by $\psi(x)$.

3.1 Examples of Operators

The mechanism to define transformation operators allows for a great variety of operators. In this section we present several examples of transformation operators, showing that one can easily capture existing operators from the literature or design novel operators to capture known patterns on sequences.

Temporal Operators. We can define operators in the style of the temporal operators from Past LTL, cf. [Manna and Pnueli, 1991]. For instance, we can define the operator

$$\diamondsuit = \langle \mathbb{B}, T, \phi, id \rangle,$$

where T consists of the transformations $set(x) = 1$ and id, and the function ϕ is defined as $\phi(0) = id$ and $\phi(1) = set$. Then we can write a definition

$$p \coloneqq \diamondsuit(a \mid 0),$$

which defines p as true when a has happened. Here the only meaningful choice of the initial transformation element is 0, and hence it can be omitted. Thus we can write the same definition as

$$p \coloneqq \diamondsuit a,$$

with the understanding that it corresponds to the one above. Other temporal operators can be introduced in a similar way.

Threshold Counter Operators. The *threshold counter operator* with threshold value n is

$$\mathcal{T}_n = \langle \mathbb{N}, T, \phi, \psi \rangle,$$

where T consists of the transformations inc and id, with $inc(x) = x + 1$; the function ϕ is defined as $\phi(1) = inc$ and $\phi(0) = id$; and the function ψ is defined as $\psi(x) = 1$ if $x \geq n$ and $\psi(x) = 0$ otherwise. The operator allows one to check whether a given condition has occurred at least n times. Notably, we can define the operator equivalently as a finite operator by replacing the set of all natural numbers $\mathbb{N}$ with the finite set $[0, n]$ and modifying the increment transformation as $inc(x) = \min(n, x + 1)$. They are equivalent because ψ will not distinguish integers greater than n.

Example 3. *An agent must collect at least 30 units of stone, and at least 115 units of iron given that 13 have already been collected. When it has collected a sufficient number of units, it can deliver and get rewarded. The reward function is described by the query $(P, reward)$ where P consists of the following definitions:*

$$\begin{aligned}
enoughStone &\coloneqq \mathcal{T}_{30}(stone \mid 0),\\
enoughIron &\coloneqq \mathcal{T}_{115}(iron \mid 13),\\
successfulDelivery &\coloneqq enoughStone \wedge enoughIron\\
&\qquad \wedge delivery,\\
alreadyDelivered &\coloneqq \diamondsuit successfulDelivery,\\
notAlreadyDelivered &\coloneqq \neg alreadyDelivered,\\
reward &\coloneqq delivery \wedge notAlreadyDelivered.
\end{aligned}$$

The proposition reward holds true at the first time point when the agent succeeds in a delivery.

Parity Operator. The *parity operator* is

$$\mathcal{P} = \langle \mathbb{B}, T, \phi, id \rangle,$$

where T consists of the identity function id and the Boolean negation function $\neg$; and the function ϕ is defined as $\phi(0) = id$ and $\phi(1) = \neg$. The operator checks whether the input variable has been true an even number of times. It is exemplified by the program

$$\begin{aligned}
even &\coloneqq \mathcal{P}(a \mid 0),\\
odd &\coloneqq \neg even.
\end{aligned}$$

which defines whether a has happened an even or odd number of times.

Metric Temporal Operators. In the style of metric temporal logic [Koymans, 1990], we can define operators such as one that checks whether something happened in the last k steps—with time isomorphic to the naturals rather than to the reals as in the original metric temporal logic. The aforementioned operator is defined as

$$\diamondsuit_k = \langle \mathbb{Z}, T, \phi, \psi \rangle,$$

where the set T has transformations set and dec defined as $set(x) = k$ and $dec(x) = x - 1$; the function ϕ is defined as $\phi(1) = set$ and $\phi(0) = dec$; and the function ψ is defined as $\psi(x) = 1$ if $x > 0$ and $\psi(x) = 0$ otherwise. The operator can be equivalently defined as a finite operator by replacing $\mathbb{Z}$ with $[0, k]$, and defininig $dec(x) = \max(0, x - 1)$.

Operators with infinite transformation domain. While all the operators above can be defined as finite operators, one can also include operators that require an infinite transformation domain. For instance, the operator

$$\text{sameNum} = \langle \mathbb{Z}, T, \phi, \psi \rangle,$$

where T consists of $inc(x) = x + 1$, $id(x) = x$, and $dec(x) = x - 1$; the function ϕ is defined as $\phi(0,0) = \phi(1,1) = id$, $\phi(1,0) = inc$, $\phi(0,1) = dec$; and the function ψ is defined as $\psi(x) = 1$ iff $x = 0$. When the operator is used in a definition such as

$$p \coloneqq \text{sameNum}(a, b \mid 0),$$

we have that variable p is true iff a and b have been true the same number of times. The operator can be used to recognise the Dyck language of balanced parentheses, which is not regular. The existence of an equivalent finite operator would imply the existence of a finite automaton, and hence regularity of the language.

3.2 Finite Semigrouplike Operators

We present a principled way to define operators in terms of finite semigroups.

Definition 1. *Let us consider a finite semigroup $S = (X, \cdot)$, a surjective function $\phi : \mathbb{B}^m \to X$, and an injective function $\psi : X \to \mathbb{B}^n$. They define the transformation operator $\langle X, T, \phi, \psi \rangle$ where T consists of each transformation $y(x) = x \cdot y$ for $y \in X$. We call it a* semigrouplike operator.

Intuitively, ϕ is a binary decoding of the set X, and ψ is a binary encoding. Note that y is seen both as an element of X and as a function $y : X \to X$. From now on we assume that for every set X such an encoding and decoding is fixed. Any choice will be valid for our purposes. Thus, we simply say that a semigroup defines a semigrouplike operator, without mentioning the functions ϕ and ψ explicitly.

Definition 2 (Prime operators)**.** *The* flip-flop operator *is the transformation operator defined by the flip-flop monoid. An operator is a* (simple) group operator *if it is defined by a (simple) finite group. The* prime operators *are the flip-flop operator and the simple group operators.*

We will focus in particular on the flip-flop operator and on operators defined by cyclic groups. For these two operators we provide an explicit definition, which requires us to commit to a choice of the encoding/decoding functions. The definitions we provide are improved with respect to the ones following from a direct application of Definition 1. In particular, we omit one element from the transformation domain of the flip-flop operator, since it would be redundant.

Definition 3. *The* flip-flop operator *is*

$$\mathcal{F} = \langle \mathbb{B}, T, \phi, id \rangle,$$

where T consists of the transformations $set, reset, read$ defined as

$$set(x) = 1, \quad reset(x) = 0, \quad read(x) = x,$$

and the function ϕ is defined as

$$\phi(0,0) = read,\ \phi(1,0) = \phi(1,1) = set,\ \phi(0,1) = reset.$$

The flip-flop operator corresponds to the flip-flop from digital circuits—it corresponds to an *SR latch*, where input 11 is not allowed though, cf. [Roth *et al.*, 2004]. The operator allows us to specify a flip-flop with a definition as

$$storedBit \coloneqq \mathcal{F}(writeOne, writeZero).$$

Therefore the logic $\mathcal{L}(\mathcal{F})$ can be employed as a specification language for digital circuits with logic gates and flip-flops. Next we introduce the cyclic operators.

Definition 4. *The* cyclic operator of order n *is*

$$\mathcal{C}_n = \langle [0, n-1], T, \phi, \psi \rangle,$$

where the transformations are $T = \{inc_i \mid i \in [0, n-1]\}$ defined as

$$inc_i(x) = (x + i) \bmod n,$$

the function ϕ is defined as

$$\phi(b_1, \ldots, b_m) = inc_i,$$

where m is the minimum number of bits required to represent n, and i is the minimum between $n-1$ and the number whose binary representation is $b_1 \ldots b_m$; finally, the function $\psi(x)$ yields the binary representation of x.

When a cyclic operator is used in a definition as

$$p_1, \ldots, p_m \coloneqq \mathcal{C}_n(0, \ldots, 0, a \mid 0),$$

variables $p_1, \ldots, p_m$ provide the binary representation of the number of times a has been true modulo n. Note that the parity operator introduced earlier coincides with $\mathcal{C}_2$. Next we characterise the prime cyclic operators.

Proposition 1. *A cyclic operator $\mathcal{C}_n$ is a prime operator iff n is a prime number.*

The other prime operators are defined by finite simple groups as found in *the classification of finite simple groups*, cf. [Gorenstein *et al.*, 2018]. In addition to the infinite family of cyclic groups of prime order, the classification also includes the two infinite families of *alternating groups* and groups of *Lie type*, along with the 27 *sporadic* groups.

4 Expressivity Results

We show expressivity results for Transformation Logics featuring finite operators, with a focus on semigroup-like operators. We start by defining the notion of expressivity for a Transformation Logic, in terms of formal languages.

Definition 5. *Consider a program P with input variables $V = \{p_1, \ldots, p_n\}$. Every letter $\langle b_1, \ldots, b_n \rangle$ of the alphabet $\mathbb{B}^n$ defines an assignment ν to the variables in V, as $\nu(p_i) = b_i$. Then, every word w over $\mathbb{B}^n$ defines an input I_w to P. A query (P, q)* accepts *a word w over $\mathbb{B}^n$ iff $(P, I_w, |w|) \models q$; and it* recognises *a language L over $\mathbb{B}^n$ if it accepts exactly the words in L.*

Definition 6. *The* expressivity *of a logic $\mathcal{L}$ is the set of languages recognised by its queries. A logic $\mathcal{L}_1$ is* less expressive *than a logic $\mathcal{L}_2$, written $\mathcal{L}_1 \sqsubset \mathcal{L}_2$, if the expressivity of $\mathcal{L}_1$ is properly included in the expressivity of $\mathcal{L}_2$.*

First, we establish the expressivity when all finite operators are included.

Theorem 1. *For* $\mathbf{A}$ *the set of all finite operators, the expressivity of* $\mathcal{L}(\mathbf{A})$ *is the regular languages.*

The result follows from the fact that finite operators capture finite automata; conversely, programs with finite operators can be mapped to a composition of finite automata. In particular to a so-called *cascade composition* of automata. This characterisation allows us to employ *algebraic automata theory*, cf. [Ginzburg, 1968; Arbib, 1969; Dömösi and Nehaniv, 2005], in proving that prime operators suffice to capture regular languages.

Theorem 2. *For* $\mathbf{P}$ *the set of all prime operators, the expressivity of the logic* $\mathcal{L}(\mathbf{P})$ *is the regular languages.*

The result is obtained by showing that every transformation definition with a finite transformation operator can be captured by a program with prime operators only. The prime operators to use are suggested by the *prime decomposition theorem* for finite automata [Krohn and Rhodes, 1965]. For star-free regular languages, we obtain the following specialised result.

Theorem 3. *For* $\mathcal{F}$ *the flip-flop operator, the expressivity of the logic* $\mathcal{L}(\mathcal{F})$ *is the star-free regular languages.*

Beyond star-free, we have nameless fragments of the regular languages. Here we start their exploration. First, to go beyond star-free it suffices to introduce group operators.

Theorem 4. *For any non-empty set* $\mathbf{G}$ *of group operators, the logic* $\mathcal{L}(\mathcal{F}, \mathbf{G})$ *is strictly more expressive than* $\mathcal{L}(\mathcal{F})$.

Next we focus on cyclic operators. We show them to yield a core of the Transformation Logics with favourable properties. First, cyclic operators along with the flip-flop operator form a canonical and universal set of operators for the logic $\mathcal{L}(\mathcal{F}, \mathbf{S})$, where $\mathbf{S}$ is the set of all solvable group operators. They are canonical and universal for $\mathcal{L}(\mathcal{F}, \mathbf{S})$ in the same way the operators $\{\wedge, \neg\}$ are canonical and universal for propositional logic. We state universality and then canonicity.

Theorem 5 (Universality)**.** *For* $\mathbf{C}$ *the set of all cyclic operators of prime order, and* $\mathbf{S}$ *any set of solvable operators, the expressivity of* $\mathcal{L}(\mathcal{F}, \mathbf{C})$ *includes the expressivity of* $\mathcal{L}(\mathcal{F}, \mathbf{S})$.

Theorem 6 (Canonicity)**.** *Given two sets* $\mathbf{C}_1$ *and* $\mathbf{C}_2$ *of cyclic operators of prime order, the logics* $\mathcal{L}(\mathcal{F}, \mathbf{C}_1)$ *and* $\mathcal{L}(\mathcal{F}, \mathbf{C}_2)$ *have the same expressivity if and only if* $\mathbf{C}_1 = \mathbf{C}_2$.

The canonicity result implies the existence of infinite expressivity hierarchies such as the following one, if we note that every $\mathcal{C}_p$ is a prime operator when p is a prime number.

Corollary 1. $\mathcal{L}(\mathcal{C}_2) \sqsubset \mathcal{L}(\mathcal{C}_2, \mathcal{C}_3) \sqsubset \mathcal{L}(\mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_5) \sqsubset \cdots$.

It is worth noting that this form of canonicity does not hold for prime operators in general.

Theorem 7. *There are two sets* $\mathbf{P}_1 \subset \mathbf{P}_2$ *of prime operators such that* $\mathcal{L}(\mathbf{P}_1)$ *and* $\mathcal{L}(\mathbf{P}_2)$ *have the same expressivity.*

Combining our expressivity results together, we obtain the following hierarchy theorem.

Theorem 8. *For* $\mathbf{S}$ *all solvable group operators, and* $\mathbf{P}$ *all prime operators, the following infinite hierarchy of expressivity holds:*

$$\mathcal{L}(\mathcal{F}) \sqsubset \mathcal{L}(\mathcal{F}, \mathcal{C}_2) \sqsubset \mathcal{L}(\mathcal{F}, \mathcal{C}_2, \mathcal{C}_3) \sqsubset \cdots \sqsubset \mathcal{L}(\mathcal{F}, \mathbf{S}) \sqsubset \mathcal{L}(\mathbf{P}).$$

The theorem provides our current picture of the expressivity of the Transformation Logics. At the bottom of the hierarchy we have $\mathcal{L}(\mathcal{F})$ with the expressivity of the star-free regular languages. At the top of the hierarchy we have $\mathcal{L}(\mathbf{P})$ with the expressivity of all regular languages. Between them we have infinitely-many logics capturing distinct fragments of the regular languages.

5 Complexity Results

We study the complexity of the evaluation problem for the Transformation Logics.

Definition 7. *The* evaluation problem *of a Transformation Logic* $\mathcal{L}$ *is the problem to decide, given a query* (P, q) *with* $P \in \mathcal{L}$*, and an input* $I = I_1, \ldots, I_\ell$ *for* P*, whether it holds that* $(P, I, \ell) \models q$.

To study the complexity we need to assume a representation for the operators, which determines the *size of an operator*. The choice of a representation for operators in general is beyond the scope of this paper. We discuss two concrete cases below.

Definition 8. *A family* $\mathbf{T}$ *of operators is* polytime *if, for every operator* $\langle X, T, \phi, \psi\rangle \in \mathbf{T}$ *with* $\phi : \mathbb{B}^m \to T$ *and* $\psi : X \to \mathbb{B}^n$*, it holds that, for every* $x \in X$ *and every* $\mu \in \mathbb{B}^m$*, the value* $\psi(\tau(x))$ *with* $\tau = \phi(\mu)$ *can be computed in time polynomial in the size of the operator.*

Theorem 9. *For any (possibly infinite) set* $\mathbf{T}$ *of polytime operators, the evaluation problem of* $\mathcal{L}(\mathbf{T})$ *is in* PTIME*. Furthermore, there exists a set* $\mathbf{H}$ *of polytime operators such that the evaluation problem of* $\mathcal{L}(\mathbf{H})$ *is* PTIME*-complete.*

We argue the upper bound applies to every finite set of finite operators, including prime operators. On the contrary, infinite sets of finite operators may not be polytime.

Lemma 1. *Every finite set of finite operators is polytime. There exists a set of finite operators that is not polytime.*

Theorem 10. *For any finite set* $\mathbf{T}$ *of finite operators, the evaluation problem of* $\mathcal{L}(\mathbf{T})$ *is in* PTIME*.*

Next we focus on threshold counter operators $\mathcal{T}_n$ and cyclic operators $\mathcal{C}_n$. We denote the complete families as $\{\mathcal{T}_n\}$ and $\{\mathcal{C}_n\}$. Notably, such operators can be represented compactly.

Definition 9. *A representation for the family of threshold counter operators* $\{\mathcal{T}_n\}$ *is said to be* compact *if the size of the representation of* $\mathcal{T}_n$ *is* $O(\log n)$*. Similarly for the family of cyclic operators* $\{\mathcal{C}_n\}$.

A compact representation can be obtained by encoding the symbols $\mathcal{T}$ and $\mathcal{C}$ with a constant number of bits, and the index n in binary with a logarithmic number of bits.

Proposition 2. *The threshold-counter operators* $\{\mathcal{T}_n\}$ *and the cyclic operators* $\{\mathcal{C}_n\}$ *admit a compact representation.*

Lemma 2. *The families of operators $\{\mathcal{T}_n\}$ and $\{\mathcal{C}_n\}$ are polytime, even when represented compactly.*

Overall we obtain that polynomial time evaluation is possible for every Transformation Logic that includes any finite number of finite operators, along with the threshold-counter and cyclic operators.

Theorem 11. *For any finite set $\mathbf{A}$ of finite operators, the evaluation problem of $\mathcal{L}(\mathbf{A}, \{\mathcal{T}_n\}, \{\mathcal{C}_n\})$ is in* PTIME *even under compact representation of the operators $\{\mathcal{T}_n\}$ and $\{\mathcal{C}_n\}$.*

5.1 Constant-Depth and Data Complexity

We study the complexity of evaluation of programs having a constant depth. It implies *data complexity* results. Specifically, all membership complexity results for constant-depth programs imply also membership in data complexity, when the program is fixed and the size of the input to the program is arbitrary. Data complexity measures how evaluation of a fixed program scales with the size of the input [Vardi, 1982]. We first define the depth of a program, and corresponding classes of constant-depth programs.

Definition 10. *Consider a program P. The* depth *of an input variable of P is zero. The* depth *of a variable defined by a static definition $d \in P$ is the maximum depth of a variable in the body of d plus the depth of the parse-tree of the body of d. The* depth *of a variable defined by a delay or transformation definition $d \in P$ is the maximum depth of a variable in the body of d plus one. The* depth of P *is the maximum depth of a variable in P.*

Definition 11. *For any set of transformation operators $\mathbf{T}$, and any depth k, the Transformation Logic $\mathcal{L}(\mathbf{T} \mid k)$ is the subset of $\mathcal{L}(\mathbf{T})$ with programs of depth at most k.*

Our results are phrased in terms of three circuit complexity classes, reported below with the known inclusions.

$$\mathrm{AC}^0 \subsetneq \mathrm{ACC}^0 \subseteq \mathrm{NC}^1$$

The first result is for the flip-flop operator, and it builds on a result for the complexity of group-free semigroups [Chandra *et al.*, 1985].

Theorem 12. *For any k, evaluation of $\mathcal{L}(\mathcal{F} \mid k)$ is in AC^0.*

Cyclic group operators, and solvable group operators in general, increase the complexity of the evaluation problem.

Theorem 13. *For any depth k, and any finite set $\mathbf{S}$ of solvable group operators, evaluation of $\mathcal{L}(\mathcal{F}, \mathbf{S} \mid k)$ is in ACC^0. Furhermore, there is a solvable group operator $\mathcal{G}$ such that, for every depth $k \geq 1$, evaluation of $\mathcal{L}(\mathcal{G} \mid k)$ is not in AC^0.*

The upper bound makes use of a result for the complexity of solvable groups [Barrington, 1989]. The lower bound relies on the fact that the cyclic (hence solvable) group C_2 captures the parity function, known not to be in AC^0 [Furst *et al.*, 1984].

Non-solvable groups introduce an exact correspondence with the larger cicuit complexity class NC^1. Our result builds on a result for the complexity of non-solvable groups [Barrington, 1989].

Theorem 14. *For any depth k, and any finite set $\mathbf{G}$ of groups containing a non-solvable group, the evaluation problem of $\mathcal{L}(\mathcal{F}, \mathbf{G} \mid k)$ is complete for NC^1 under AC^0 reductions.*

6 Relationship with Past LTL

We show the Transformation Logic $\mathcal{L}(\mathcal{F})$ captures Past LTL. First, the *before* and *since* operators correspond to the delay and flip-flop operators, respectively.

Lemma 3. *Consider a Past LTL formula $\varphi = \ominus p$. Let P be the singleton program consisting of the definition $q \coloneqq \mathcal{D}\, p$. For every interpretation I of φ and every time point t, it holds that $(I, t) \models \varphi$ iff $(P, I, t) \models q$.*

Lemma 4. *Consider a Past LTL formula $\varphi = a \,\mathcal{S}\, b$. Let P be the program consisting of the two definitions $c \coloneqq \neg a$ and $p \coloneqq \mathcal{F}(b, c \mid 0)$. For every interpretation I of φ and every time point t, it holds that $(I, t) \models \varphi$ iff $(P, I, t) \models p$.*

Given the lemmas above, we can build a program for any given Past LTL formula by induction on its parse-tree, introducing one static definition for each occurence of a Boolean operator, one delay definition for each occurrence of the before operator, and one transformation definition for each occurrence of the since operator.

Theorem 15. *Every Past LTL formula φ can be translated into a program of $\mathcal{L}(\mathcal{F})$. Such a program has size linear in the size of φ, and it can computed from φ in logarithmic space.*

7 Related Work

The Transformation Logics are a valuable addition to the rich set of temporal and dynamic logics adopted in AI. Such logics include *propositional* temporal logics of the past such as Past LTL, cf. [Manna and Pnueli, 1991]; temporal logics of the future interpreted both on finite and infinite traces [Pnueli, 1977; Wolper, 1983; De Giacomo and Vardi, 2013]; dynamic logics such as PDL, cf. [Harel *et al.*, 2000], and linear dynamic logic on finite traces [De Giacomo and Vardi, 2013]; logics with a dense interpretation of time such as metric temporal logic [Koymans, 1990]. They also include *first-order* variants such as Past FOTL [Chomicki, 1995]. Finally, they include *rule-based* logics of several kinds: propositional modal temporal logics such as Templog [Abadi and Manna, 1989; Baudinet, 1995]; logics with first-order variables where time occurs explicitly as an argument of a special sort such as $\mathrm{Datalog}_{1\mathrm{S}}$ [Chomicki and Imielinski, 1988], and Temporal Datalog [Ronca *et al.*, 2018; Ronca *et al.*, 2022]; logics with metric temporal operators such as DatalogMTL [Brandt *et al.*, 2018; Walega *et al.*, 2019; Walega *et al.*, 2020]; and propositional modal logics specialised for reasoning on streams [Beck *et al.*, 2018], and weighted variants thereof [Eiter and Kiesel, 2020] which share an algebraic flavour with our work.

8 Conclusions

The Transformation Logics provide a general framework for logics over sequences. The flexibility given by the transformation operators allows for defining logics with the expressivity of many different fragments of the regular languages, under the guidance of group theory. While the star-free regular languages are well-known as they are the expressivity of many well-established formalisms, the fragments beyond them are less known. The Transformation Logics provide a way to explore them.

Acknowledgments

Alessandro Ronca is supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (Grant agreement No. 852769, ARiAT). For the purpose of Open Access, the author has applied a CC BY public copyright licence to any Author Accepted Manuscript (AAM) version arising from this submission.

References

[Abadi and Manna, 1989] Martín Abadi and Zohar Manna. Temporal logic programming. *J. Symb. Comput.*, 8, 1989.

[Arbib, 1969] Michael Arbib. *Theories of Abstract Automata*. Automatic Computation. Prentice-Hall, 1969.

[Bacchus *et al.*, 1996] Fahiem Bacchus, Craig Boutilier, and Adam J. Grove. Rewarding behaviors. In *AAAI*, 1996.

[Barrington, 1989] David A. Mix Barrington. Bounded-width polynomial-size branching programs recognize exactly those languages in NC1. *J. Comput. Syst. Sci.*, 38, 1989.

[Baudinet, 1995] Marianne Baudinet. On the expressiveness of temporal logic programming. *Inf. Comput.*, 117, 1995.

[Beck *et al.*, 2018] Harald Beck, Minh Dao-Tran, and Thomas Eiter. LARS: A logic-based framework for analytic reasoning over streams. *Artif. Intell.*, 261, 2018.

[Bonassi *et al.*, 2023] Luigi Bonassi, Giuseppe De Giacomo, Marco Favorito, Francesco Fuggitti, Alfonso Emilio Gerevini, and Enrico Scala. Planning for temporally extended goals in pure-past linear temporal logic. In *ICAPS*, 2023.

[Brafman and De Giacomo, 2019] Ronen I. Brafman and Giuseppe De Giacomo. Planning for LTLf/LDLf goals in non-markovian fully observable nondeterministic domains. In *IJCAI*, 2019.

[Brafman *et al.*, 2018] Ronen I. Brafman, Giuseppe De Giacomo, and Fabio Patrizi. LTLf/LDLf non-markovian rewards. In *AAAI*, 2018.

[Brandt *et al.*, 2018] Sebastian Brandt, Elem Güzel Kalayci, Vladislav Ryzhikov, Guohui Xiao, and Michael Zakharyaschev. Querying log data with metric temporal logic. *J. Artif. Intell. Res.*, 62, 2018.

[Camacho *et al.*, 2017] Alberto Camacho, Eleni Triantafillou, Christian J. Muise, Jorge A. Baier, and Sheila A. McIlraith. Non-deterministic planning with temporally extended goals: LTL over finite and infinite traces. In *AAAI*, 2017.

[Camacho *et al.*, 2019] Alberto Camacho, Rodrigo Toro Icarte, Toryn Q. Klassen, Richard Anthony Valenzano, and Sheila A. McIlraith. LTL and beyond: Formal languages for reward function specification in reinforcement learning. In *IJCAI*, 2019.

[Chandra *et al.*, 1985] Ashok K. Chandra, Steven Fortune, and Richard J. Lipton. Unbounded fan-in circuits and associative functions. *J. Comput. Syst. Sci.*, 30, 1985.

[Chomicki and Imielinski, 1988] Jan Chomicki and Tomasz Imielinski. Temporal deductive databases and infinite objects. In *PODS*, 1988.

[Chomicki, 1995] Jan Chomicki. Efficient checking of temporal integrity constraints using bounded history encoding. *ACM Trans. Database Syst.*, 20, 1995.

[De Giacomo and Rubin, 2018] Giuseppe De Giacomo and Sasha Rubin. Automata-theoretic foundations of FOND planning for LTLf and LDLf goals. In *IJCAI*, 2018.

[De Giacomo and Vardi, 2013] Giuseppe De Giacomo and Moshe Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *IJCAI*, 2013.

[De Giacomo *et al.*, 2020a] Giuseppe De Giacomo, Marco Favorito, Luca Iocchi, Fabio Patrizi, and Alessandro Ronca. Temporal logic monitoring rewards via transducers. In *KR*, 2020.

[De Giacomo *et al.*, 2020b] Giuseppe De Giacomo, Luca Iocchi, Marco Favorito, and Fabio Patrizi. Restraining bolts for reinforcement learning agents. In *AAAI*, 2020.

[Dömösi and Nehaniv, 2005] Pál Dömösi and Chrystopher L. Nehaniv. *Algebraic theory of automata networks: An introduction*. SIAM, 2005.

[Eiter and Kiesel, 2020] Thomas Eiter and Rafael Kiesel. Weighted LARS for quantitative stream reasoning. In *ECAI*, 2020.

[Furst *et al.*, 1984] Merrick L. Furst, James B. Saxe, and Michael Sipser. Parity, circuits, and the polynomial-time hierarchy. *Math. Syst. Theory*, 17, 1984.

[Ginzburg, 1968] Abraham Ginzburg. *Algebraic Theory of Automata*. Academic Press, 1968.

[Gorenstein *et al.*, 2018] Daniel Gorenstein, Richard Lyons, and Ronald Solomon. *The Classification of the Finite Simple Groups, Number 8*. American Mathematical Soc., 2018.

[Harel *et al.*, 2000] David Harel, Jerzy Tiuryn, and Dexter Kozen. *Dynamic Logic*. MIT Press, 2000.

[Icarte *et al.*, 2018] Rodrigo Toro Icarte, Toryn Q. Klassen, Richard Anthony Valenzano, and Sheila A. McIlraith. Teaching multiple tasks to an RL agent using LTL. In *AAMAS*, 2018.

[Kleene, 1956] Stephen C. Kleene. Representation of events in nerve nets and finite automata. *Automata studies*, 34, 1956.

[Koymans, 1990] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real Time Syst.*, 2, 1990.

[Krohn and Rhodes, 1965] Kenneth Krohn and John Rhodes. Algebraic theory of machines. I. Prime decomposition theorem for finite semigroups and machines. *Trans. Am. Math. Soc.*, 116, 1965.

[Manna and Pnueli, 1991] Zohar Manna and Amir Pnueli. Completing the temporal picture. *Theor. Comput. Sci.*, 83, 1991.

[Pnueli, 1977] Amir Pnueli. The temporal logic of programs. In *FOCS*, 1977.

[Ronca *et al.*, 2018] Alessandro Ronca, Mark Kaminski, Bernardo Cuenca Grau, Boris Motik, and Ian Horrocks. Stream reasoning in Temporal Datalog. In *AAAI*, 2018.

[Ronca *et al.*, 2022] Alessandro Ronca, Mark Kaminski, Bernardo Cuenca Grau, and Ian Horrocks. The delay and window size problems in rule-based stream reasoning. *Artif. Intell.*, 306, 2022.

[Ronca, 2024] Alessandro Ronca. The Transformation Logics. *arXiv*, 2304.09639, 2024.

[Roth *et al.*, 2004] Charles Roth, Larry Kinney, and Eugene John. *Fundamentals of logic design*. Thomson Brooks/Cole, 2004.

[Torres and Baier, 2015] Jorge Torres and Jorge A. Baier. Polynomial-time reformulations of LTL temporally extended goals into final-state goals. In *IJCAI*, 2015.

[Vardi, 1982] Moshe Y. Vardi. The complexity of relational query languages (extended abstract). In *STOC*, 1982.

[Walega *et al.*, 2019] Przemyslaw Andrzej Walega, Bernardo Cuenca Grau, Mark Kaminski, and Egor V. Kostylev. DatalogMTL: Computational complexity and expressive power. In *IJCAI*, 2019.

[Walega *et al.*, 2020] Przemyslaw Andrzej Walega, Bernardo Cuenca Grau, Mark Kaminski, and Egor V. Kostylev. DatalogMTL over the integer timeline. In *KR*, 2020.

[Walega *et al.*, 2023] Przemyslaw Andrzej Walega, Mark Kaminski, Dingmin Wang, and Bernardo Cuenca Grau. Stream reasoning with DatalogMTL. *J. Web Semant.*, 76, 2023.

[Wolper, 1983] Pierre Wolper. Temporal logic can be more expressive. *Inf. Control.*, 56, 1983.