Lecture Notes on
Monadic First- and Second-Order Logic on Strings

Dino Mandrioli, Davide Martinenghi, Angelo Morzenti,
Matteo Pradella, and Matteo Rossi

Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB), Politecnico di Milano,
Piazza Leonardo Da Vinci 32, 20133 Milano, Italy
{firstname.lastname}@polimi.it

1 Introduction

From the very beginning of formal language and automata theory, the investigation of the relations between defining a language through some kind of abstract machine and through a logic formalism has produced challenging theoretical problems and important applications in system design and verification. A well-known example of such an application is the classical Hoare's method to prove the correctness of a Pascal-like program w.r.t. a specification stated as a pair of pre- and post-conditions expressed through a first-order theory [6].

Such a verification problem is undecidable if the involved formalisms have the computational power of Turing machines but may become decidable for less powerful formalisms as in the important case of Regular Languages. Originally, Büchi, Elgot, and Trakhtenbrot [1,2,9] independently developed a *Monadic Second-Order logic* defining exactly the Regular Language family, so that the decidability properties of this class of languages could be exploited to achieve automatic verification.

Intuitively, monadic logics have some syntactic restrictions on the predicates used. In particular, only predicates that have one argument (i.e., whose arity is 1) are allowed (with the exception of the ordering relation). As usual, in the first-order case only variables can be quantified. In the second-order, instead, monadic predicates—i.e., predicates with arity 1, as mentioned above—can also be quantified, thus resulting in so-called second-order variables.

Interestingly, to capture the full class of Regular Languages by means of a monadic logic it is necessary to exploit a second-order version of the logic, which is more powerful than the simpler first-order one; it has been shown by McNaughton and Papert [7], however, that restricting the logic to first-order allows users to define precisely the *non-counting* subclass of Regular Languages—i.e., the languages which cannot "count" the number of repeated occurrences of a given subword.[1]

Such logic characterizations, however, have not been exploited in practice to achieve automatic verification due to the intractable complexity of the necessary algorithms.

---

[1] For instance, the language $\{(ab)^{2n} \mid n > 0\}$ is counting. Non-counting languages, in turn, are equivalent to other interesting subclasses of regular ones, such as, e.g., the *star-free* ones, i.e. those languages that can be defined by means of regular expressions not making use of the Kleene-* operation.

Later on, a major breakthrough in this field has been obtained thanks to the advent of *model checking*, which exploits language characterization in terms of *temporal logic* [3]. Temporal logic has the same expressive power as first-order logic but, being less succint, allows for more efficient (though still exponential) verification algorithms.

These notes present the essentials of first- and second-order monadic logics with introductory purpose and are organized as follows. In Section 2, we discuss Monadic First-Order logic and show that it is strictly less expressive than Finite-State Automata, in that it only captures a strict subset of Regular Languages—the non-counting ones. We then introduce Monadic Second-Order logic in Section 3; such a logic is, syntactically, a superset of Monadic First-Order logic and captures Regular Languages exactly. We also show how to transform an automaton into a corresponding formula and vice versa. Finally, in Section 4 we discuss the use of logical characterizations of classes of languages, such as those described in Sections 2 and 3, as the basis for automatic verification techniques.

2   Monadic First-order Logic of Order on Strings

Given an input alphabet Σ, formulae of the *monadic first-order logic* (MFO) are built out of the following elements:

- First-order variables, denoted as lowercase letters (written in boldface to avoid confusion with strings), x, y, ..., which are interpreted over the natural numbers $\mathbb{N}$.
- Monadic predicates $a(\cdot)$, $b(\cdot)$, ..., one for each symbol of Σ; intuitively, $a(x)$ evaluates to true in a string w if, and only if, the character of w at position x is a.
- The order relation $<$ between natural numbers.
- The usual propositional connectives and first-order quantifiers.

More precisely, let $\mathcal{V}$ be a finite set of first-order variables, and let Σ be an alphabet. Well-formed formulae of the MFO logic are defined according to the following syntax:

$$\varphi := a(x) \mid x < y \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists x(\varphi)$$

where $a \in \Sigma$ and $x, y \in \mathcal{V}$.

The usual predefined abbreviations are introduced to denote the remaining propositional connectives, the universal quantifier, the arithmetic relations $\geq, \leq, =, \neq, >$, and sums and subtractions between first order variables and numeric constants.

We have the following definitions of propositional connectives and first-order quantifiers:

$$\varphi_1 \wedge \varphi_2 \overset{\text{def}}{=} \neg(\neg\varphi_1 \vee \neg\varphi_2)$$

$$\varphi_1 \Rightarrow \varphi_2 \overset{\text{def}}{=} \neg\varphi_1 \vee \varphi_2$$

$$\varphi_1 \Leftrightarrow \varphi_2 \overset{\text{def}}{=} (\varphi_1 \Rightarrow \varphi_2) \wedge (\varphi_2 \Rightarrow \varphi_1)$$

$$\forall x(\varphi) \overset{\text{def}}{=} \neg\exists x(\neg\varphi)$$

the following definitions of relations:

$$x \geq y \stackrel{\text{def}}{=} \neg(x < y)$$

$$x \leq y \stackrel{\text{def}}{=} y \geq x$$

$$x = y \stackrel{\text{def}}{=} x \leq y \wedge y \leq x$$

$$x \neq y \stackrel{\text{def}}{=} \neg(x = y)$$

$$x > y \stackrel{\text{def}}{=} y < x$$

and the following definitions of constants, of the successor of a natural number, and of addition and subtraction of constant values:

$$x = 0 \stackrel{\text{def}}{=} \forall y \neg(y < x)$$

$$\text{succ}(x, y) \stackrel{\text{def}}{=} x < y \wedge \neg \exists z (x < z \wedge z < y)$$

$$y = x + k \stackrel{\text{def}}{=} \exists z_0, \ldots, z_k (z_0 = x \wedge \text{succ}(z_0, z_1) \wedge \text{succ}(z_1, z_2) \wedge \ldots \wedge \text{succ}(z_{k-1}, z_k) \wedge y = z_k)$$

$$y = x - k \stackrel{\text{def}}{=} x = y + k$$

where k is a constant in $\mathbb{N}$. Further useful abbreviations are the follwowing ones:

– first(x) and last(x) identify, respectively, the first and last positions in the string:
  first(x) $\stackrel{\text{def}}{=} \neg \exists y (y < x)$, obviously equivalent to $x = 0$;
  last(x) $\stackrel{\text{def}}{=} \neg \exists y (y > x)$

An MFO formula is interpreted over a string $w \in \Sigma^*$, with respect to assignment $\nu : \mathcal{V} \to U$, where $U = \{0, \ldots, |w| - 1\}$, which maps $\mathcal{V}$ to a position in string w. Notice that, if $w = \varepsilon$, then function $\nu(x)$ is undefined for every variable $x \in \mathcal{V}$. The satisfaction relation (indicated, as usual, as $\models$) for MFO formulae is defined in the following way:

– $w, \nu \models a(x)$ if, and only if, $\nu(x)$ is defined and $w[\nu(x)] = a$
– $w, \nu \models x < y$ if, and only if, $\nu(x) < \nu(y)$ holds
– $w, \nu \models \neg\varphi$ if, and only if, $w, \nu \not\models \varphi$ holds
– $w, \nu \models \varphi_1 \vee \varphi_2$ if, and only if, at least one of $w, \nu \models \varphi_1$ and $w, \nu \models \varphi_2$ holds
– $w, \nu \models \exists x(\varphi)$ if, and only if, $|w| > 0$ and $w, \nu[i/x] \models \varphi$ hold for some $i \in \{0, \ldots, |w| - 1\}$

where $w[k]$ denotes the letter of w at position k, and $\nu[i/x]$ is the mapping that assigns i to x and otherwise coincides with ν. Notice that, in case $w = \varepsilon$ (and therefore $U = \emptyset$), $w, \nu \not\models a(x)$ for any a and x (i.e., it is not the case that $w, \nu \models a(x)$), and $w, \nu \not\models \exists x(\varphi)$ for any x and φ (i.e., an existential quantification is false, and conversely a universal quantification is true, if the quantified variable ranges over the empty set).

To improve readability, we will drop ν from the notation whenever there is no risk of ambiguity—i.e., we will write $w \models \varphi$ to indicate that string w satisfies formula φ.

An MFO *sentence* is a closed MFO formula. Given a sentence φ, the language $L(\varphi)$ is defined as

$$L(\varphi) = \{w \in \Sigma^* \mid w \models \varphi\}.$$

We say that a language L is *expressible* in MFO (or *definable* in MFO or *MFO-definable* for short) iff there exists a MFO sentence φ such that $L = L(\varphi)$.

2.1 Examples

The following MFO formula φ_{L_1} defines the language L_1 made of all strings that start with symbol a:

$$\varphi_{L_1} : \exists x (x = 0 \wedge a(x))$$

The following formula φ_{L_2} defines the language L_2 made of all strings in which every symbol a is necessarily immediately followed by a b (notice that these strings cannot end with a symbol a).

$$\varphi_{L_2} : \forall x (a(x) \Rightarrow \exists y (\mathrm{succ}(x, y) \wedge b(y)))$$

The following formula φ_{L_3} defines the language L_3 made of all (nonempty) strings in which the last symbol is an a.

$$\varphi_{L_3} : \exists x (\mathrm{last}(x) \wedge a(x))$$

The following formula φ_{L_4} defines the language L_4 made of all strings (containing at least 3 symbols) in which the symbol three positions from the right is an a.

$$\varphi_{L_4} : \exists x (a(x) \wedge \exists y (y = x + 2 \wedge \mathrm{last}(y)))$$

Alternatively, language L_4 is also defined by the following formula φ'_{L_4}:

$$\varphi'_{L_4} : \exists x (\mathrm{last}(x) \wedge \exists y (y = x - 2 \wedge a(y)))$$

The following formula φ_{L_ϵ} (or, alternatively, φ'_{L_ϵ}) defines the language L_ϵ made of only the empty string (assuming that the input alphabet includes at least symbol a):

$$\varphi_{L_\epsilon} : \neg \exists x (a(x) \vee \neg a(x))$$

$$\varphi'_{L_\epsilon} : \forall x (a(x) \wedge \neg a(x))$$

Formula φ_{L_ϵ} (resp., formula φ'_{L_ϵ}) states that, given a word w of language L_ϵ, there does not exist a position such that *true* holds (resp., in all its positions *false* holds). This can only occur if the set of positions is empty: by definition an existential quantification is false, and a universal quantification is true, if it ranges over an empty set, that is, if w is the empty string.

Finally, the following formula $\varphi_{L_\emptyset}$ defines the empty language (assuming that the input alphabet Σ includes at least symbol a):

$$\varphi_{L_\emptyset} : \exists x (a(x) \wedge \neg a(x))$$

Formula $\varphi_{L_\emptyset}$ is contradictory, as it states that a position exists in which symbol a both appears and does not appear. No string w (not even the empty one) is such that $w \models \varphi_{L_\emptyset}$ holds, hence the formula defines the empty language.

Every singleton language—i.e., every language consisting of one finite-length string— is trivially expressible in MFO. Consider, for instance, language $L_{abc} = \{abc\}$ that includes only string abc. It is easily defined by the following MFO formula:

$$\varphi_{L_{abc}} : \exists x \exists y \exists z (x = 0 \wedge y = succ(x) \wedge z = succ(y) \wedge last(z) \wedge a(x) \wedge b(y) \wedge c(z))$$

2.2   Expressiveness of MFO

The following statements trivially hold.

Proposition 1. *Let L, L_1, and L_2 be any languages defined by MFO formulae φ, φ_1 and φ_2, respectively:*

- *Language $L_1 \cap L_2$ is defined by formula $\varphi_1 \wedge \varphi_2$—i.e., $L(\varphi_1 \wedge \varphi_2) = L_1 \cap L_2$.*
- *Language $L_1 \cup L_2$ is defined by formula $\varphi_1 \vee \varphi_2$—i.e., $L(\varphi_1 \vee \varphi_2) = L_1 \cup L_2$.*
- *Language $\bar{L}$ (the complement of L) is defined by formula $\neg\varphi$—i.e., $L(\neg\varphi) = \bar{L}$.*

The next theorem follows from Proposition 1.

Theorem 2. *The family of MFO-definable languages is closed under union, intersection, and complementation.*

To further investigate the expressive power of MFO,[2] we consider the MFO-definable languages over a one-letter alphabet $\Sigma = \{a\}$. In this simple case the MFO predicate $a(x)$ is always true at any position x in any interpretation, therefore it is redundant and every formula is equivalent to a formula that does not include any occurrence of predicate $a(\cdot)$—e.g., $\exists x\, (a(x) \wedge y < x)$ is equivalent to $\exists x\, (y < x)$.

We next show that, for the simple family of the languages over a one-letter alphabet, every language is MFO-definable if, and only if, it is finite[3] or co-finite (where a co-finite language is one whose complement is finite). As a consequence, for instance, the simple *regular* language $L_{even} = \{\, a^{2n} \mid n \geq 0 \,\}$ is *not* MFO-definable, which proves that the MFO logic is strictly less expressive than finite state automata and regular grammars and expressions.

Our proof that every language over a one-letter alphabet is expressible in MFO if, and only if, it is finite or co-finite is organized as follows. First, we observe that if a language is finite or co-finite, then it is expressible in MFO, as a consequence of the fact that—as exemplified by language L_{abc} in Section 2.1—singleton languages are expressible in MFO and that the family of MFO-expressible languages is closed under union and complementation. Next, we prove that if a language over a one-letter alphabet is MFO-definable, then it is finite or co-finite. This is in turn proved in three steps:

1. we introduce a new logic called QF-MFO, a quantifier-free fragment of MFO;
2. we show that every language over a one-letter alphabet $\Sigma = \{a\}$ is QF-MFO-definable if, and only if, it is finite or co-finite;
3. we show that the two logics, MFO and QF-MFO, are equally expressive, as every MFO formula φ has an equivalent QF-MFO formula.

---

[2] In the present section we follow the line of discussion adopted in the (unedited, to the best of our knowledge) lecture notes *Automata theory - An algorithmic approach* by Javier Esparza, February 13, 2019.

[3] Recall that a finite language is one whose cardinality is finite.

To define the QF-MFO logic, we first introduce a few additional abbreviations (where k is a constant in $\mathbb{N}$):

$$x < y + k \; \overset{\text{def}}{=} \; \exists z \, (z = y + k \wedge x < z)$$

$$x > y + k \; \overset{\text{def}}{=} \; \exists z \, (z = y + k \wedge z < x)$$

$$x < k \; \overset{\text{def}}{=} \; \exists z \, (z = 0 \wedge x < z + k)$$

$$x > k \; \overset{\text{def}}{=} \; \exists z \, (z = 0 \wedge x > z + k)$$

$$k < last \; \overset{\text{def}}{=} \; \forall x \, (last(x) \Rightarrow x > k)$$

$$k > last \; \overset{\text{def}}{=} \; \forall x \, (last(x) \Rightarrow x < k)$$

Definition 3 (QF-MFO). *The formulae of QF-MFO are defined by the following syntax:*

$$\varphi := x < k \; | \; x > k \; | \; x < y + k \; | \; x > y + k \; | \; k < last \; | \; k > last \; | \; \varphi \wedge \varphi \; | \; \varphi \vee \varphi$$

where $x, y \in \mathcal{V}$ and $k \in \mathbb{N}$.

In the remainder, with some (innocuous) overloading, a constant k will denote both the numerical value $k \in \mathbb{N}$ and the string a^k.

Proposition 4. *Every language L over a one-letter alphabet is QF-MFO-definable if, and only if, it is finite or co-finite.*

Proof. Only if part: Every QF-MFO sentence defines a finite or a co-finite language.

Let φ be a sentence of QF-MFO. Since QF-MFO is quantifier-free, the sentence φ is an and-or combination of formulae of type $k < last$ and $k > last$. Then, the following cases arise.

- $L(k < last) = \{k + 1, k + 2, \; \dots \}$ is a co-finite language (remember that numbers identify words, and vice versa, so that $\{k + 1, k + 2, \; \dots \}$ is the same as $\{a^{k+1}, a^{k+2}, \; \dots \}$).
- $L(k > last) = \{0, 1, \; \dots \; k\}$ is a finite language.
- $L(\varphi_1 \vee \varphi_2) = L(\varphi_1) \cup L(\varphi_2)$. If $L_1 = L(\varphi_1)$ and $L_2 = L(\varphi_2)$ are both finite, then $L(\varphi_1 \vee \varphi_2)$ is also finite; if L_1 and L_2 are both co-finite, then the language $L(\varphi_1 \vee \varphi_2) = L_1 \cup L_2 = \overline{\overline{L_1 \cup L_2}} = \overline{\overline{L_1} \cap \overline{L_2}}$ is the complement of the intersection of two finite languages, hence it is co-finite; if one of the two languages L_1 and L_2 is finite and the other is co-finite, then $L(\varphi_1 \vee \varphi_2)$ is the complement of the intersection of a finite and a co-finite language, therefore it is co-finite.
- $L(\varphi_1 \wedge \varphi_2) = L(\varphi_1) \cap L(\varphi_2)$. If $L_1 = L(\varphi_1)$ and $L_2 = L(\varphi_2)$ are both finite, then their intersection is finite; if L_1 and L_2 are both co-finite, then $L(\varphi_1 \wedge \varphi_2) = L_1 \cap L_2 = \overline{\overline{L_1 \cap L_2}} = \overline{\overline{L_1} \cup \overline{L_2}}$ is the complement of the union of two finite languages, hence it is co-finite; if one of the two languages L_1 and L_2 is finite and the other is co-finite, then $L(\varphi_1 \wedge \varphi_2)$ is the complement of the union of a finite language and a co-finite language, hence it is finite.

If part: Every finite or co-finite language is definable by a QF-MFO sentence.
If L is finite then $L = \{k_1, \ldots k_n\}$ and

$$\varphi_L = \varphi_{\{k_1\}} \vee \cdots \vee \varphi_{\{k_n\}} =$$

$$= (k_1 - 1 < last \wedge last < k_1 + 1) \vee \cdots \vee (k_n - 1 < last \wedge last < k_n + 1)$$

If L is co-finite, then its complement $\overline{L}$ is finite, therefore it is defined by some QF-MFO formula. Then, L is QF-MFO-definable if, for every QF-MFO sentence φ, there exists a QF-MFO sentence, call it $\overline{\varphi}$, that defines the language $\overline{L}$, the complement of L. Such a sentence $\overline{\varphi}$ is equal to $neg(\varphi)$, where function $neg(\cdot)$ is defined inductively by the following clauses.

– $neg(k < last) = last < k \vee \underbrace{(k - 1 < last \wedge last < k + 1)}_{last=k}$

– $neg(k > last) = k < last \vee \underbrace{(k - 1 < last \wedge last < k + 1)}_{k=last}$

– $neg(\varphi_1 \vee \varphi_2) = neg(\varphi_1) \wedge neg(\varphi_2)$
– $neg(\varphi_1 \wedge \varphi_2) = neg(\varphi_1) \vee neg(\varphi_2)$

$\square$

Proposition 5. *Every MFO formula φ over a one-letter alphabet is equivalent to some QF-MFO formula f—i.e., $\varphi \equiv f$.*

Proof. The proof is by induction on the structure of φ.

If $\varphi = x < y$, then $\varphi \equiv x < y + 0$.

If $\varphi = \neg \psi$, then the inductive hypothesis can be applied and then the negation can be removed using the De Morgan's laws and equivalences such as, e.g., $\neg(x < y + k) \equiv x \geq y + k$ (where $x \geq y + k$ is a natural abbreviation for $x > y + k - 1$).

If $\varphi = \varphi_1 \vee \varphi_2$, then the induction hypothesis is directly applied.

If $\varphi = \exists x \, \psi$ then, by the induction hypothesis, $\psi \equiv f$ for some QF-MFO formula f, and f can be assumed to be in disjunctive normal form—i.e., $f = D_1 \vee \cdots \vee D_n$, and $\varphi \equiv \exists x D_1 \vee \cdots \vee \exists x D_n$; then, we define a set of QF-MFO formulae f_i such that, for each $1 \leq i \leq n$, $f_i \equiv \exists x D_i$ holds. Notice that, since f is a QF-MFO formula, each f_i is such that it does not include any quantification of variable x nor, if φ is a sentence, any occurrence of the same variable.

Each f_i is built as follows. Formula f_i is a conjunction of formulae that contains all the conjuncts of D_i that do not include any occurrence of variable x, plus the conjuncts defined next. Consider every pair of conjuncts of D_i, one conjunct being of type $t_1 < x$, where $t_1 = h$ or $t_1 = y + h$ and the constraint is an *upper bound* (i.e., h is maximal, that is, it is the greatest value that appears in a constraint of the type $y + h < x$), and the other conjunct being of type $x < t_2$, where $t_2 = h$ or $t_2 = y + h$ and the constraint is a *lower bound* (i.e., h is minimal); for every such pair we add to f_i a conjunct equivalent to $t_1 + 1 < t_2$; for instance, if the two above-described conjuncts are $z - 4 < x$ and $x < y + 3$, then the added conjunct is $z < y + 6 \equiv z - 3 < y + 3$. Notice that, if only the conjunct of type $t_1 < x$ is present and the conjunct of type $x < t_2$ is missing, then the (trivially true) conjunct $x < last + 1$ must be considered—in place of the latter—as the other element of the pair; similarly, if only the conjunct of type $x < t_2$ is present and

the conjunct of type $t_1 < x$ is missing, then the (trivially true) conjunct $-1 < x$ must be considered in place of the latter.

Then, $f_i \equiv \exists x D_i$; notice that f_i does not include any occurrence of variable x nor any quantification of that variable.   □

Example 6. In the MFO formula

$$\exists x(x < y + 3 \wedge z < x + 4 \wedge z < y + 2 \wedge y < x + 1)$$

we identify the pair of constraints $z - 4 < x$ and $x < y + 3$, from which we get the additional conjunct $z - 3 < y + 3 \equiv z < y + 6$; we also identify the pair of constraints $y - 1 < x$ and $x < y + 3$, from which we get the additional conjunct $y < y + 3$. Therefore, the MFO formula $\exists x(x < y + 3 \wedge z < x + 4 \wedge z < y + 2 \wedge y < x + 1)$ is equivalent to the QF-MFO formula

$$z < y + 6 \wedge y < y + 3 \wedge z < y + 2$$

□

Example 7. We provide two examples of QF-MFO formulae equivalent to given MFO sentences.

- The MFO formula $\exists x \, \exists y \, \exists z \, (x < y \wedge y < z)$ defines the language $\{a^k \mid k \geq 3\}$. By repeated application of the inductive step, moving inside-out, we obtain $f_1 \equiv \underbrace{\exists z \, (x < y \wedge y < z)}_{\exists z \, D_1}$ and the pair of constraints on the quantified variable z, $y < z$ and $z < last + 1$, from which we derive constraint $y < last$, so that $f_1 \equiv x < y \wedge y < last$ holds; in the next inductive step we have $f_1 \equiv \underbrace{\exists y(x < y \wedge y < last)}_{\exists y \, D_1}$ and the pair of constraints on the quantified variable y, $x < y$ and $y < last$, from which we derive $x + 1 < last$ and $f_1 \equiv x + 1 < last$; in the final inductive step we have $f_1 \equiv \underbrace{\exists x \, (x + 1 < last)}_{\exists x \, D_1}$ and the pair of constraints on the quantified variable x, $-1 < x$ and $x < last - 1$, from which we obtain $0 < last - 1$ and $f_1 \equiv last > 1$, hence $last > 1$ is the QF-MFO formula equivalent to the original MFO formula $\exists x \, \exists y \, \exists z \, (x < y \wedge y < z)$.
- The MFO formula $\exists x \, (\neg \exists y \, (x < y) \wedge x < 4)$ defines the language $\{a^k \mid k \leq 4\}$. Again moving inside-out, we have $f_1 \equiv \underbrace{\exists y \, (x < y)}_{\exists y \, D_1}$ and the pair of constraints on the quantified variable y, $x < y$ and $y < last + 1$, from which we derive $x + 1 < last + 1 \equiv x < last$ and $f_1 \equiv x < last$; at the next inductive step we apply negation and obtain $f_1 \equiv \underbrace{\exists x \, (x \geq last \wedge x < 4)}_{\exists x \, D_1}$ and the pair of constraints on the quantified variable x, $last - 1 < x$ and $x < 4$, from which we obtain $f_1 \equiv last < 4$. Hence, $last < 4$ is the QF-MFO formula equivalent to the original MFO formula $\exists x \, (\neg \exists y \, (x < y) \wedge x < 4)$.

□

The following theorem easily follows from Proposition 4 and Proposition 5.

Theorem 8. *A language over a one-letter alphabet is expressible in MFO if, and only if, it is finite or co-finite.*

Every MFO formula defines a regular language. In fact, MFO is a restriction of the Monadic Second-Order (MSO) logic introduced in Section 3 and, as shown there, for every MSO sentence φ there is a Finite-State Automaton (FSA) that accepts exactly the language defined by φ—hence, *a fortiori* this also holds for every MFO formula. We have therefore the following result (whose proof will be given in Section 3.1).

Statement 1 *For every MFO sentence φ there is a FSA $\mathcal{A}$ such that $L(\varphi) = L(\mathcal{A})$.*

However, the MFO logic is strictly less expressive than Finite State Automata (also abbreviated as FSA), as not all regular languages are expressible in MFO. Indeed, as a consequence of Theorem 8 the regular language L_{even} defined above, which includes exactly the strings over alphabet $\Sigma = \{a\}$ having even length and therefore is neither finite nor co-finite, is *not* expressible in MFO, as stated by the next corollary.

Corollary 9. *There is no MFO sentence φ that defines language L_{even} (i.e., such that $L(\varphi) = L_{even}$).*

From Statement 1 and Corollary 9 the following result is immediate, by observing that it is easy to define a FSA $\mathcal{A}$ such that $L(\mathcal{A}) = L_{\text{even}}$.

Theorem 10. *MFO is strictly less expressive than FSA.*

On the other hand, the set of languages that can be defined through MFO formulae is not closed under the so-called "Kleene star" operation, as stated by the following theorem.

Theorem 11. *The set of languages that can be defined by MFO sentences is not closed under the * operation.*

Proof. To prove the claim it is enough to remark that the following MFO formula $\varphi_{L_{aa}}$ defines language $L_{aa} = \{aa\}$ (i.e., the language containing only string aa):

$$\varphi_{L_{aa}} : \exists x \exists y (x = 0 \wedge y = x + 1 \wedge a(x) \wedge a(y) \wedge \text{last}(y))$$

and that $L_{\text{even}} = L_{aa}^*$. $\qquad\square$

From Theorem 2 and Theorem 11, it can be shown [7] that MFO can express only the so-called "star-free" languages—that is, those that can be obtained through union, intersection, complementation and concatenation of finite languages.

For example, language L_3 of Section 2.1 can be obtained from finite languages $L_3' = \emptyset$ and $L_3'' = \{a\}$—containing, respectively, no string (hence, whose cardinality is 0) and only string a (hence, whose cardinality is 1)—in the following way:

$$L_3 = (\neg L_3') \cdot L_3''$$

As a further example, the language $L_{\exists a}$, which is made of all strings that contain at least an a, can be defined in the following way:

$$L_{\exists a} = (\neg L_3') \cdot L_3'' \cdot (\neg L_3')$$

and the language $L_{\exists!a}$ made of all strings that contain *exactly* one a can be defined as follows:

$$L_{\exists!a} = (\neg L_{\exists a}) \cdot L_3'' \cdot (\neg L_{\exists a})$$

3   Monadic Second-order Logic of Order on Strings

Formulae of the *monadic second-order logic of order* (MSO), as defined by Büchi and others [8], are built out of the elements of the MFO logic defined in Section 2 plus, in addition:

- Second-order variables, denoted as uppercase boldface letters, $X, Y, \dots$, which are interpreted over *sets* of natural numbers.

More precisely, let Σ be an input alphabet, $\mathcal{V}_1$ be a set of first-order variables, and $\mathcal{V}_2$ be a set of second-order (or set) variables. Well-formed formulae of MSO logic are defined according to the following syntax:

$$\varphi := a(x) \mid X(x) \mid x < y \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists x(\varphi) \mid \exists X(\varphi)$$

where $a \in \Sigma, x, y \in \mathcal{V}_1$, and $X \in \mathcal{V}_2$.

Naturally, all abbreviations introduced in Section 2 are still valid. We also introduce the following additional abbreviations:

$$x \in X \stackrel{\text{def}}{=} X(x)$$
$$X \subseteq Y \stackrel{\text{def}}{=} \forall x(x \in X \Rightarrow x \in Y)$$
$$X = Y \stackrel{\text{def}}{=} (X \subseteq Y) \wedge (Y \subseteq X)$$
$$X \neq Y \stackrel{\text{def}}{=} \neg(X = Y)$$

where x, y, X are as before, and $Y \in \mathcal{V}_2$.

An MSO formula is interpreted over a string $w \in \Sigma^*$, with respect to assignments $\nu_1 : \mathcal{V}_1 \to \{0, \dots, |w|-1\}$ and $\nu_2 : \mathcal{V}_2 \to \mathcal{P}(\{0, \dots, |w|-1\})$. Notice that, like assignment ν for MFO formulae, ν_1 maps each first-order variable of $\mathcal{V}_1$ to a position in string w. Assignment ν_2, instead, maps each second-order variable of $\mathcal{V}_2$ to a *set* of positions in string w.

Then, the satisfaction relation $\models$ for MSO formulae is defined in the following way:

- $w, \nu_1, \nu_2 \models a(x)$ if, and only if, $\nu_1(x)$ is defined and $w[\nu_1(x)] = a$
- $w, \nu_1, \nu_2 \models X(x)$ if, and only if, $\nu_1(x) \in \nu_2(X)$ holds
- $w, \nu_1, \nu_2 \models x < y$ if, and only if, $\nu_1(x) < \nu_1(y)$ holds
- $w, \nu_1, \nu_2 \models \neg\varphi$ if, and only if, $w, \nu_1, \nu_2 \not\models \varphi$ holds
- $w, \nu_1, \nu_2 \models \varphi_1 \vee \varphi_2$ if, and only if, at least one of $w, \nu_1, \nu_2 \models \varphi_1$ and $w, \nu_1, \nu_2 \models \varphi_2$ holds
- $w, \nu_1, \nu_2 \models \exists x(\varphi)$ if, and only if, $|w| > 0$ and some $i \in \{0, \dots, |w|-1\}$ satisfies $w, \nu_1[i/x], \nu_2 \models \varphi$
- $w, \nu_1, \nu_2 \models \exists X(\varphi)$ if, and only if, $|w| > 0$ and some $S \subseteq \{0 \dots |w|-1\}$ satisfies $w, \nu_1, \nu_2[S/X] \models \varphi$

where $w[k]$ and $\nu[i/x]$ are as in Section 2, and $\nu_2[S/X]$ assigns S to X and otherwise coincides with ν_2.

To improve readability, we will drop ν_1, ν_2 from the notation whenever there is no risk of ambiguity, and write $w \models \varphi$ to indicate that string w satisfies MSO formula φ.

The definitions of MSO *sentence* and of language $L(\varphi)$ defined by sentence φ are as for the MFO logic.

Example 12. The following MSO formula φ_{even} defines language L_{even} introduced in Section 2.2.

$$\varphi_{\text{even}} : \exists \boldsymbol{P} \left(\forall \boldsymbol{x} \left(\begin{array}{l} (\boldsymbol{x} = 0 \Rightarrow \neg \boldsymbol{P}(\boldsymbol{x})) \\ \wedge \\ \forall \boldsymbol{y}(\boldsymbol{y} = \boldsymbol{x} + 1 \Rightarrow (\neg \boldsymbol{P}(\boldsymbol{x}) \Leftrightarrow \boldsymbol{P}(\boldsymbol{y}))) \\ \wedge \\ a(\boldsymbol{x}) \\ \wedge \\ (\text{last}(\boldsymbol{x}) \Rightarrow \boldsymbol{P}(\boldsymbol{x})) \end{array} \right) \right)$$

Formula φ_{even} introduces a second-order variable $\boldsymbol{P}$ that identifies exactly all even positions in string w. More precisely, the first position of w (which is conventionally 0), is not even, and indeed the first conjunct in formula φ_{even} states that $\boldsymbol{P}(x)$ does not hold when x is 0. In addition, the second conjunct in φ_{even} states that the next position after $\boldsymbol{x}$ (i.e., position $\boldsymbol{y}$ such that $\boldsymbol{y} = \boldsymbol{x} + 1$ holds), if it exists, is even (i.e., $\boldsymbol{P}(\boldsymbol{y})$ holds there) if, and only if, position $\boldsymbol{x}$ is odd; hence, since the first position is odd, the second position is even, the third is odd, the fourth is even, and so on. The third conjunct states that, in every position $\boldsymbol{x}$, $a(\boldsymbol{x})$ holds (i.e., a appears in every position). Finally the last conjunct states that the last position in the string must be even. $\qquad\square$

3.1   Expressiveness of MSO

Since every MFO formula is also an MSO formula, from the fact that MSO formula φ_{even} introduced in Example 12 defines language L_{even}, which, by Proposition 9, cannot be defined by an MFO formula, we have the following straightforward result.

Theorem 13. *The MSO logic is strictly more expressive than the MFO logic.*

Indeed, the original seminal result by Büchi and others is that, unlike the MFO logic, MSO indeed has the same expressive power as FSAs, as captured by the following theorem.

Theorem 14. *A language L is regular if, and only if, there exists a sentence φ in the MSO logic such that $L = L(\varphi)$.*

Before proving Theorem 14, we remark that, since for every FSA there is an equivalent MSO formula—and vice versa—MSO enjoys all closure properties of FSAs, as captured by the following corollary.

Corollary 15. *The set of languages that can be defined by MSO formulae is closed under union, intersection, complementation, and Kleene star.*

The proof of Theorem 14 is constructive, i.e., it provides an algorithmic procedure that, for a given FSA $\mathcal{A}$, builds an equivalent MSO sentence $\varphi_{\mathcal{A}}$, and vice versa. Next, we offer an intuitive explanation of the construction, referring the reader to, e.g., [8] for a complete and detailed proof.

From FSA to MSO logic

The key idea of the construction consists in using, for each state q of FSA $\mathcal{A}$, a second-order variable X_q, whose value is the set of positions of all the characters that $\mathcal{A}$ may read in a transition starting from state q.

Without loss of generality, we assume that $\mathcal{A}$'s set of states Q is $\{0, 1, \ldots, m\}$, for some m, where 0 denotes the initial state. Then, we encode the definition of the FSA $\mathcal{A}$ recognizing L (i.e, such that $L = L(\mathcal{A})$) as the conjunction of several clauses, each one capturing a part of the definition of $\mathcal{A}$:

- We introduce a formula capturing the transition relation δ of $\mathcal{A}$, which includes a disjunct for each transition $\delta(q_i, a) = q_j$ of the automaton:
  $\forall x, y \left(y = x + 1 \Rightarrow \bigvee_{\delta(q_i, a) = q_j} \left(x \in X_i \wedge a(x) \wedge y \in X_j \right) \right)$.
- The fact that the machine starts in state 0 is captured by formula
  $\forall x (x = 0 \Rightarrow x \in X_0)$.
- Since the automaton cannot be in two different states i, j at the same time, for each pair of distinct second-order variables X_i and X_j we introduce formula
  $\neg \exists y (y \in X_i \wedge y \in X_j)$.
- Acceptance by the automaton—i.e. $\delta(q_i, a) \in F$—is formalized by formula
  $\forall x \left(\text{last}(x) \Rightarrow \bigvee_{\delta(q_i, a) \in F} \left(x \in X_i \wedge a(x) \right) \right)$.

Finally, MSO formula $\varphi_{\mathcal{A}}$ corresponding to automaton $\mathcal{A}$ is the following sentence

$$\varphi_{\mathcal{A}} : \exists X_0, X_1, \ldots X_m(\varphi)$$

where φ is the conjunction of all the above clauses.

It is not difficult to show that the set of strings satisfying formula $\varphi_{\mathcal{A}}$ is exactly L.

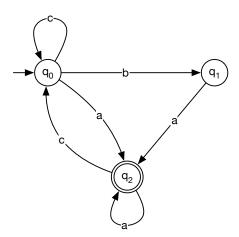*Example 16.* Consider the FSA $\mathcal{A}_{\text{ex}}$ shown in Figure 1. The corresponding MSO for-

Fig. 1: Finite-State Automaton $\mathcal{A}_{\text{ex}}$.

mula $\varphi_{\mathcal{A}_{\text{ex}}}$ built according to the rules described above is the following:

$$\varphi_{\mathcal{A}_{\text{ex}}} : \exists X_0, X_1, X_2 \left(\begin{array}{l} \forall x, y \left(y = x + 1 \Rightarrow \left(\begin{array}{ll} x \in X_0 \wedge c(x) \wedge y \in X_0 & \vee \\ x \in X_0 \wedge b(x) \wedge y \in X_1 & \vee \\ x \in X_0 \wedge a(x) \wedge y \in X_2 & \vee \\ x \in X_1 \wedge a(x) \wedge y \in X_2 & \vee \\ x \in X_2 \wedge c(x) \wedge y \in X_0 & \vee \\ x \in X_2 \wedge a(x) \wedge y \in X_2 & \end{array} \right) \right) \\ \wedge \\ \forall x (x = 0 \Rightarrow x \in X_0) \\ \wedge \\ \neg \exists y (y \in X_0 \wedge y \in X_1) \wedge \\ \neg \exists y (y \in X_0 \wedge y \in X_2) \wedge \\ \neg \exists y (y \in X_1 \wedge y \in X_2) \\ \wedge \\ \forall x \left(\text{last}(x) \Rightarrow \left(\begin{array}{l} (X_0(x) \wedge a(x)) \\ \vee \\ (X_1(x) \wedge a(x)) \\ \vee \\ (X_2(x) \wedge a(x)) \end{array} \right) \right) \end{array} \right)$$

where the first clause captures the transition relation of the automaton; the second clause formalizes its initial state; the next three conjuncts state the mutual exclusion of states; and the last clause captures the acceptance condition. □

Remark 17. The formalization of FSA $\mathcal{A}$ through MSO formula $\varphi_{\mathcal{A}}$, and in particular the clause capturing the acceptance condition, assumes that accepted strings contain at least one symbol (i.e., they are not empty). To formalize in MSO an FSA $\mathcal{A}_\epsilon$ that also accepts the empty string, it is enough to include in the corresponding formula $\varphi_{\mathcal{A}_\epsilon}$ also a disjunct covering the case of the empty string, in the following way:

$$\varphi_{\mathcal{A}_\epsilon} : \exists X_0, X_1, \ldots X_m(\varphi) \vee \varphi_{L_\epsilon}$$

where φ_{L_ϵ} is the formula introduced in Section 2.1 to capture the empty string.

From MSO logic to FSA

The construction in the opposite direction has been proposed in various versions in the literature. Here we summarize its main steps along the lines of [8]. First, the MSO sentence is translated into a standard form that uses only second-order variables (no first-order variables are allowed), the $\subseteq$ predicate, and variables W_a, for each $a \in \Sigma$, denoting the set of all the positions of the word containing the character a. Moreover, we use Succ, which has the same meaning as succ, has second-order variable arguments that are singletons. This simpler (yet equivalent) logic is defined by the following syntax:

$$\varphi := X \subseteq W_a \mid X \subseteq Y \mid \text{Succ}(X, Y) \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists X(\varphi).$$

As before, we also use the standard abbreviations for, e.g., $\wedge$, $\forall$, =. To translate first-order variables to second-order variables we need to state that a (second-order) variable is a singleton. Hence we introduce the abbreviation:

$$\mathrm{Sing}(X) \stackrel{\mathrm{def}}{=} \exists Y(Y \subseteq X \wedge Y \neq X \wedge \neg \exists Z(Z \subseteq X \wedge Z \neq Y \wedge Z \neq X))$$

Then, in the transformation below, $\mathrm{Succ}(X, Y)$ is always conjoined with $\mathrm{Sing}(X) \wedge \mathrm{Sing}(Y)$ and the resulting formula is therefore false whenever X or Y are not singletons. The following step entails the inductive construction of the equivalent automaton. This is built by associating a single automaton to each elementary subformula and by composing them according to the structure of the global formula. This inductive approach requires to use open formulas, i.e., formulas where free variables occur. For technical reasons, with such formulas we are going to consider words on the alphabet $\Sigma \times \{0, 1\}^k$, where k is the number of free variables; in the subsequent steps of the transformation from MSO logic to FSA, the alphabet will revert to Σ. Hence, if $X_1, X_2, \ldots X_k$ are the free variables used in the formula, a value of 1 in the, say, j-th component means that the considered position belongs to X_j (that is, the second-order variable X_j represents a first-order variable whose value is the considered position); 0 means the opposite. For instance, if $w = (b, 1, 0)(a, 0, 0)(a, 0, 1)$, then $w \models X_2 \subseteq W_a$, $w \models X_1 \subseteq W_b$, with X_1 and X_2 singletons representing (first-order variables and hence) positions in string w respectively equal to 0 and 2.

Formula transformation

1. First order variables are translated in the following way: $\exists x(\varphi(x))$ becomes $\exists X(\mathrm{Sing}(X) \wedge \varphi'(X))$, where φ' is the translation of φ, and X is a fresh new variable not occurring elsewhere.
2. Subformulas having the form $a(x)$, $\mathrm{succ}(x, y)$ are translated into $X \subseteq W_a$, $\mathrm{Succ}(X, Y)$, respectively.
3. The other parts are unchanged.

Inductive construction of the automaton We assume for simplicity that $\Sigma = \{a, b\}$, and that $k = 2$, i.e. two variables are used in the formula. Moreover, in the transition labels of the automata we use the shortcut symbol $\circ$ to mean all possible values.
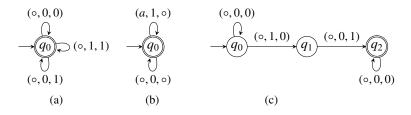

Fig. 2: Automata for the construction from MSO logic to FSA.

- The formula $X_1 \subseteq X_2$ is translated into an automaton that checks that there are 1's for the X_1 component only in positions where there are also 1's for the X_2 component (Figure 2 (a)).
- The formula $X_1 \subseteq W_a$ is analogous: the automaton checks that positions marked by 1 in the X_1 component must have symbol a (Figure 2 (b)).
- The formula $\mathrm{Succ}(X_1, X_2)$ considers two singletons, and checks that the 1 for component X_1 is immediately followed by a 1 for component X_2 (Figure 2 (c)).
- Formulas inductively built with $\neg$ and $\vee$ are covered by the closure of regular languages w.r.t. complement and union, respectively.
- For a formula of type $\exists X(\varphi)$, we use the closure under alphabet projection; for instance, we may start with an automaton with input alphabet $\Sigma \times \{0, 1\}^2$, for the formula $\varphi(X_1, X_2)$ and we may need to define an automaton for the formula $\exists X_1(\varphi(X_1, X_2))$. But in this case the alphabet is $\Sigma \times \{0, 1\}$, where the last component represents the only free remaining variable, i.e. X_2.

  The automaton $\mathcal{A}_{\exists}$ is built by starting from the one for $\varphi(X_1, X_2)$, and changing the transition labels from $(a, 0, 0)$ and $(a, 1, 0)$ to $(a, 0)$; $(a, 0, 1)$ and $(a, 1, 1)$ to $(a, 1)$, and analogously for those with b. The idea is that this last automaton nondeterministically "guesses" the quantified component (i.e. X_1) when reading its input, and the resulting word $w \in (\Sigma \times \{0, 1\}^2)^*$ is such that $w \models \varphi(X_1, X_2)$. Thus, $\mathcal{A}_{\exists}$ recognizes $\exists X_1(\varphi(X_1, X_2))$.

We refer the reader to the available literature for a full proof of equivalence between the logic formula and the constructed automaton. Here we illustrate the rationale of the above construction through the following example.

Example 18. Consider the language $L = \{a, b\}^* aa \{a, b\}^*$: it consists of the strings satisfying the formula:
$\varphi_L = \exists x \exists y (\mathrm{succ}(x, y) \wedge a(x) \wedge a(y))$.

As seen before, first we translate this formula into a version using only second-order variables: $\varphi'_L = \exists X, Y (\mathrm{Sing}(X) \wedge \mathrm{Sing}(Y) \wedge \mathrm{Succ}(X, Y) \wedge X \subseteq W_a \wedge Y \subseteq W_a)$.

The automata for $\mathrm{Sing}(X)$ and $\mathrm{Sing}(Y)$ are depicted in Figure 3; they could also be obtained by expanding the definition of Sing and then projecting the quantified variables.

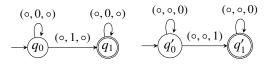Fig. 3: Automata for $\mathrm{Sing}(X)$ and $\mathrm{Sing}(Y)$.

By intersecting the automata for $\mathrm{Sing}(X)$, $\mathrm{Sing}(Y)$, and $\mathrm{Succ}(X, Y)$, by means of the customary construction of the cartesian product automaton (the details of the construction are not shown), we obtain an automaton that is identical to the one we defined

for translating formula $\mathrm{Succ}(X_1, X_2)$, where here X takes the role of X_1 and Y of X_2. Intersecting it with those for $X \subseteq W_a$ and $Y \subseteq W_a$ produces the automaton of Figure 4.
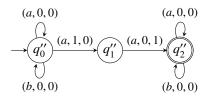

Fig. 4: Automaton for the conjunction of $\mathrm{Sing}(X)$, $\mathrm{Sing}(Y)$, $\mathrm{Succ}(X, Y)$, $X \subseteq W_a$, $Y \subseteq W_a$.

Finally, by projecting on the quantified variables X and Y we obtain the automaton for L, given in Figure 5.   □
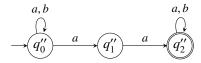

Fig. 5: Automaton for $L = \{a, b\}^* aa \{a, b\}^*$.

4   Discussion

The logical characterization of a class of languages, together with the decidability of the associated *containment* problem (i.e., checking whether a language is a subset of another language in that class), is the main door towards automatic verification techniques. Suppose that a logic formalism $\mathfrak{L}$ is recursively equivalent to an automaton family $\mathfrak{A}$; then, one can use a formula $\varphi_\mathfrak{L}$ of $\mathfrak{L}$ to specify the requirements of a given system and an abstract machine $\mathcal{A}$ in $\mathfrak{A}$ to implement the desired system: the correctness of the design defined by $\mathcal{A}$ w.r.t. to the requirements stated by $\varphi_\mathfrak{L}$ is therefore formalized as $L(\mathcal{A}) \subseteq L(\varphi_\mathfrak{L})$, i.e., all behaviors realized by the machine are also satisfying the requirements. This is just the case with FSAs and MSO logic for Regular Languages.

Unfortunately, known theoretical lower bounds state that the decision of the above containment problem is PSPACE-complete and therefore intractable in general. The recent striking success of model-checking [3], however, has produced many refined results that explain how and when practical tools can produce results of "acceptable complexity" – although the term "acceptable" is context-dependent, since in some cases even running times of the order of hours or weeks can be considered acceptable. In a

nutshell, normally—and roughly—we trade a lower expressive power of the adopted logic, typically linear temporal logic, for a complexity that is "only exponential" in the size of the logic formulas, whereas the worst case complexity for MSO logic can be even a non-elementary function [4].[4] In any case, our interest in these notes is not on the complexity issues, but it is focused on the equivalence between automata recognizers and MSO logics, which leads to the decidability of the above fundamental containment problem.

References

1. J. R. Büchi. Weak Second-Order Arithmetic and Finite Automata. *Mathematical Logic Quarterly*, 6(1-6):66–92, 1960.
2. C. C. Elgot. Decision problems of finite automata design and related arithmetics. *Trans. Am. Math. Soc.*, 98(1):21–52, 1961.
3. E. A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 995–1072. 1990.
4. M. Frick and M. Grohe. The complexity of first-order and monadic second-order logic revisited. *Ann. Pure Appl. Logic*, 130(1-3):3–31, 2004.
5. J. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Tools and Algorithms for the Construction and Analysis of Systems, First International Workshop, TACAS '95, LNCS 1019*, 1995.
6. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
7. R. McNaughton and S. Papert. *Counter-Free Automata*. Research Monograph. The M.I.T. Press, 1971.
8. W. Thomas. Handbook of theoretical computer science (vol. B). chapter Automata on infinite objects, pages 133–191. MIT Press, Cambridge, MA, USA, 1990.
9. B. A. Trakhtenbrot. Finite automata and logic of monadic predicates (in Russian). *Doklady Akademii Nauk SSR*, 140:326–329, 1961.

---

[4] There are, however, a few noticeable cases of tools that run satisfactorily at least in some particular cases of properties expressed in MSO logic [5].