Model Checking and Automaticity

Klaus Sutner
Carnegie Mellon University

Fall 2017
1 Model Checking

- Rational Relations
- Properties of Rat
- Synchronous Relations
- Model Checking Automatic Structures
Recall that a formula $\varphi$ of first-order logic is valid iff it holds in all structures (or the right signature). Validity certainly is a key problem for FOL, but here is a variant that is more interesting for our purposes.

**Problem:** Model Checking  
**Instance:** A FO structure $C$ and a FO sentence $\varphi$.  
**Question:** Is $\varphi$ valid over $C$?

Note that if we are trying to find algorithms for this problem we will need to be able to specify the structure $C$ as part of the input. Obviously this is going to be a bit tricky. At least for finite structures there is no problem, though.
Model checking is CS terminology, in standard mathematical logic one usually speaks about the first-order theory of a structure.

\[ \text{Th}(C) = \{ \varphi \text{ sentence} \mid C \models \varphi \} \]

Model checking then comes down to determining whether \( \varphi \in \text{Th}(C) \).

Since formulae are just syntactic objects (that can easily be coded as natural numbers or strings), \( \text{Th}(C) \) might well be, say, decidable. This requires that the underlying language is countable; an entirely reasonable assumption.

Note that we are not assuming a constant for every element in the structure, so this is different from complete diagrams.
It can be very interesting to solve the model checking problem even when the structure $C$ is fixed. For fixed structures one often speaks of expression model checking.

**Problem:** Expression Model Checking  
**Instance:** A FO sentence $\varphi$.  
**Question:** Is $\varphi$ valid over $C$?

E.g., we could try to decide arithmetic, i.e. which formulae are true over the fixed structure $\mathbb{N}$.

This could easily provide answers to questions like the infinitude of prime twins or the Goldbach conjecture. Perhaps somewhat unexpectedly, even the Riemann hypothesis could be handled this way, it is equivalent to a $\Pi_1$ statement (difficult).
For FO arithmetic we typically use a signature \((2, 2, 0, 0; 2)\):

\[
\mathcal{N} = \langle \mathbb{N}; +, \cdot, 0, 1, < \rangle
\]

Note that the atomic diagram of this structure is easily decidable, even primitive recursive.

Alas, one can show that for any partial recursive function \(f: \mathbb{N} \hookrightarrow \mathbb{N}\) there is a formula \(\varphi\) such that

\[f(x) \downarrow \iff \mathcal{N} \models \exists w \varphi(x, w)\]

where \(\varphi\) has only bounded quantifiers of the form \(\forall z < u\) and \(\exists z < u\).

But then the Halting Problem could be solved if we could handle EMC over \(\mathcal{N}\).

Doom and disaster.
Just to be clear, not only is $\text{Th}(\mathbb{N})$ undecidable even with just one unbounded existential quantifier, we need an oracle of type $\emptyset^{(\omega)}$ to deal with a general formula.

More precisely, the compute power needed to determine the truth of an arithmetical formula depends on the complexity of the quantifiers. In particular to deal with $\Sigma_k$ formulae, one needs an oracle like $\emptyset^{(k)}$.

Again, just a single existential quantifier is already enough to produce the Halting Problem.

Interestingly, replacing integers by rationals does not help:

**Theorem (J. Robinson, 1948)**

The theory of the rationals with addition and multiplication is undecidable.
Somewhat counterintuitively, replacing the rationals by the reals (an uncountable structure, no less) makes things easier: now the theory is decidable by a famous result by Tarski.

**Theorem (A. Tarski, 1948)**

*The theory of the reals with addition and multiplication is decidable.*

As a consequence, basic geometry is decidable. This is interesting e.g. for robotics.

The theorem is proved by a very interesting technique that provides a direct decision algorithm: *quantifier elimination*.

Tarski’s original method was highly inefficient, though (not bounded by a stack of exponentials). There are better methods now, but the complexity is provably doubly exponential.
Here is another variant of model checking that may seem slightly less natural: the formula is fixed and the structure varies.

<table>
<thead>
<tr>
<th>Problem:</th>
<th>Data Model Checking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance:</td>
<td>A FO structure $C$.</td>
</tr>
<tr>
<td>Question:</td>
<td>Is $\varphi$ valid over $C$?</td>
</tr>
</tbody>
</table>

Usually the class of structures under consideration is fairly narrow: one can think of this as a classification problem. For example, one might want to know whether a cellular automaton is reversible.

In CS one often speaks of data complexity in connection with this variant.
Cellular automata are a now classical example of discrete dynamical systems. One can think of them as structures of the form

\[ C = \langle 2^\mathbb{Z}, f \rangle \]

where \( f : 2^\mathbb{Z} \to 2^\mathbb{Z} \) operates on bi-infinite sequences of bits, but is described by a finite lookup table \( 2^w \to 2 \) (a Boolean function).

The following formula naturally expresses the fact that such a system is reversible:

\[ \forall x, y \ (f(x) = f(y) \Rightarrow x = y) \]

From a computational perspective, the underlying space is a horror. But we still can ask simple questions about it and compute the correct answer.
Data model checking here comes down to trying to understand (some part of) the structure of a particular cellular automaton. This can be interesting even when $w = 3$, so the whole system is described by a single ternary Boolean function. E.g., there is a width 3, binary cellular automaton that is computationally universal. Unfortunately, this property can not be expressed in FOL.

Expression model checking, on the other hand, is used to classify cellular automata according to simple criteria such as reversibility, existence of certain cycles, surjectivity, and so on.

Incidentally, reversibility can be solved in quadratic time (in the size of the table for $f$) via an algorithm that can be derived directly from the corresponding DMC algorithm.
For the general model checking problem we need, as part of the input, structures of the form

\[ A = \langle A; f_1, f_2, \ldots, R_1, R_2, \ldots \rangle \]

So we have to specify a set as well as a collection of functions and relations on this set.

This is quite tricky; obviously we have to move away from considering purely set-theoretic objects, at the very least we should have some nice finitary description of the structure.

For example, the structure suitable for (elementary) arithmetic, \( \mathbb{N} \), certainly has a nice description: we could think of \( \mathbb{N} \) as \( 2^{\star} \) and then define the operations correspondingly.
Here is a sledge-hammer constraint: consider only finite structures.

In this case we can write down all the functions and relations over $\mathcal{A}$ as explicit tables. For example, here is a multiplication table for a finite group over $A = \{a, b, c, d, e, f, g, h\}$

<table>
<thead>
<tr>
<th></th>
<th>$e$</th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>$d$</th>
<th>$f$</th>
<th>$g$</th>
<th>$h$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e$</td>
<td>$e$</td>
<td>$a$</td>
<td>$b$</td>
<td>$c$</td>
<td>$d$</td>
<td>$f$</td>
<td>$g$</td>
<td>$h$</td>
</tr>
<tr>
<td>$a$</td>
<td>$a$</td>
<td>$c$</td>
<td>$e$</td>
<td>$b$</td>
<td>$f$</td>
<td>$h$</td>
<td>$d$</td>
<td>$g$</td>
</tr>
<tr>
<td>$b$</td>
<td>$b$</td>
<td>$e$</td>
<td>$c$</td>
<td>$a$</td>
<td>$g$</td>
<td>$d$</td>
<td>$h$</td>
<td>$f$</td>
</tr>
<tr>
<td>$c$</td>
<td>$c$</td>
<td>$b$</td>
<td>$a$</td>
<td>$e$</td>
<td>$h$</td>
<td>$g$</td>
<td>$f$</td>
<td>$d$</td>
</tr>
<tr>
<td>$d$</td>
<td>$d$</td>
<td>$f$</td>
<td>$g$</td>
<td>$h$</td>
<td>$e$</td>
<td>$a$</td>
<td>$b$</td>
<td>$c$</td>
</tr>
<tr>
<td>$f$</td>
<td>$f$</td>
<td>$h$</td>
<td>$d$</td>
<td>$g$</td>
<td>$a$</td>
<td>$c$</td>
<td>$e$</td>
<td>$b$</td>
</tr>
<tr>
<td>$g$</td>
<td>$g$</td>
<td>$d$</td>
<td>$h$</td>
<td>$f$</td>
<td>$b$</td>
<td>$e$</td>
<td>$c$</td>
<td>$a$</td>
</tr>
<tr>
<td>$h$</td>
<td>$h$</td>
<td>$g$</td>
<td>$f$</td>
<td>$d$</td>
<td>$c$</td>
<td>$b$</td>
<td>$a$</td>
<td>$e$</td>
</tr>
</tbody>
</table>

Exercise

Determine the properties of this group.
In principle, it is not difficult to determine the validity of a sentence $\varphi$ over a finite structure; essentially one can just implement Tarski’s definition of truth.

Testing truth of the matrix of a formula, given bindings for all the quantified variables, really comes down to repeated table lookup. To deal with quantifiers we use loops ranging over the finite universe. So the algorithm is certainly primitive recursive.

In fact, a closer look shows that PSPACE is already enough. Alas, that is where things end: even when $A = \{0, 1\}$ and we only have the standard Boolean operations, validity checking for FOL formula is PSPACE-complete (this problem is often called QBF: quantified Boolean formulae).
A much less radical idea is to consider structures where all components are computable. This leads to computable model theory (or “recursive model theory” in classical parlance). Note that any computable structure is by necessity countable, but this is not a major concern for us. Alas, as \( \mathcal{M} \) shows, a (trivially) computable structure can still have a hugely complicated theory.

Presburger arithmetic uses the language \( \mathcal{L}(+, -, 0, 1; <) \) of signature \( (2, 2, 0, 0; 2) \), informally this is arithmetic without multiplication.

**Theorem (M. Presburger, 1929)**

*Presburger arithmetic is decidable.*

We’ll see a proof next week. Unsurprisingly, Presburger’s algorithm is triple exponential: even for quantifier-free formulae the problem is \( \text{NP} \)-hard.
Here is a Crazy Idea™: how about structures that can be described by finite state machines?

The carrier set would be a regular set of words, but that is not a particularly drastic constraint (in particular since one can consider infinite words).

We need to do a bit of groundwork first, though. In particular, we have to explain how a finite state machine can deal with functions and relations.

Actually, we will get around functions by simply assuming there are no function symbols in our language. This is not a big restriction, we can always translate a function into its graph, a relation.
A *relational structure* is a FO structure of the form

\[ C = \langle A; R_1, R_2, \ldots, R_k \rangle \]

In other words, we simply do not allow any functions.

This may seem too radical, but we can always fake functions as relations:

\[ F(x, y) \iff f(x) = y \]
But note that this changes our formulae a bit.

For example, consider the formula \( f(f(x)) = y \).

There really is a quantifier hidden there:

\[
\exists z \ (f(x) = z \land f(z) = y)
\]

and so any decision algorithm has to cope with this invisible quantifier.

In a purely relational structure everything is out in the open, we have to write something like

\[
\exists z \ (F(x, z) \land F(z, y))
\]
So the structures we are interested in have the restricted form

\[ C = \langle A; R_1, R_2, \ldots \rangle \]

where

- \( A \subseteq \Sigma^* \) is a regular set of words, and
- \( R_i \subseteq A^{k_i} \), a regular relation on words of arity \( k_i \).

We already know how to handle the carrier set, but we do not have anything like a “regular relation” at this point.
- Model Checking

2 Rational Relations

- Properties of Rat

- Synchronous Relations

- Model Checking Automatic Structures
The author (along with many other people) has come recently to the conclusion that the functions computed by the various machines are more important—or at least more basic—than the sets accepted by these devices.

So the next project is to generalize regular languages to some reasonable class of regular relations.

We have a few basic options to tackle this problem:

- Invent some kind of memoryless machine that takes $k$-tuples of words as input.
- Generalize labeled transition systems to deal with $k$-tuples of words.
- Exploit algebraic characterization of regular languages.

As it turns out, the algebraic approach is arguably the simplest and cleanest.
More precisely, we will use Kleene’s theorem as the starting point for our definition of “regular relation.”

Theorem (Kleene 1956)

*Every regular language over \( \Sigma \) can be constructed from \( \emptyset \) and singletons \( \{a\} \), \( a \in \Sigma \), using only the operations union, concatenation and Kleene star.*

It follows that there is a convenient notation system (regular expressions) for regular languages that is radically different from finite state machines: we can use an algebra (albeit a slightly weird one) to concoct regular languages.

One direction is easy, given the closure properties of regular languages we already have: every regular expression denotes a regular language.
The opposite direction is handled by dynamic programming. Unfortunately, the regular expressions involved grow exponentially, so the algorithm is not practical.

Still, one very nice feature of Kleene’s characterization is that a good definition often generalizes. In this case, the monoid $\Sigma^*$ is perhaps the most natural setting, but there are other plausible choices.

In particular we could use the product monoid $\Sigma^* \times \Sigma^*$ instead: since we are dealing with sets of pairs of strings we naturally obtain binary relations this way.

The relevant algebraic structures are called Kleene algebras. We will not study them in any detail and just pull out the pieces that we need for our project.
Suppose $\langle M, \cdot, 1 \rangle$ is a monoid. Here is a general way to construct a Kleene algebra on top of $M$. The carrier set is $\mathcal{P}(M)$ and the operations are

- set theoretic union,
- pointwise multiplication, and
- Kleene star.

More precisely, define

$$K \cdot L = \{ x \cdot y \mid x \in K, y \in L \}$$

$$K^0 = \{1\} \quad K^{n+1} = K \cdot K^n$$

$$K^* = \bigcup_{n \geq 0} K^n$$
Definition

A \textit{k-ary rational relation} is a relation $R \subseteq M$ where

\[ M = \Sigma_1^* \times \Sigma_2^* \times \ldots \times \Sigma_k^* \]

and $R$ is generated in the Kleene algebra over $M$ from elements

\[ (\varepsilon, \ldots, \varepsilon, a, \varepsilon, \ldots, \varepsilon) \]

Strictly speaking, this should be a singleton set, but in this context it is best not to distinguish between $z$ and $\{z\}$. Trust me.

Note that in the special case $k = 1$ we get back ordinary regular languages.
It is easy to see that we could also allow generators of the form

\[(x_1, x_2, \ldots, x_{k-1}, x_k)\]

where \(x_i \in \Sigma_i^*\).

Using these and customary operation symbols \(+, \cdot\) (often implicit) and \(*\) we obtain rational expressions that provide a notation system for rational relations.
We will mostly deal with the case \( k = 2 \) and consider the monoid

\[
M = \Sigma^* \times \Gamma^*
\]

We often write \( x/y \) for an element of \( M \), so \( x \in \Sigma^* \), \( y \in \Gamma^* \). This is just fancy notation for a pair of words.

Writing rational expression can be a bit confusing, so on occasion we use vector notation, in particular in the case \( k = 2 \), and write \( (x \ y) \). Think of this a two tracks with one word in each track.

Note that multiplication here is componentwise:

\[
(x \ y) \cdot (u \ v) = (xu \ yv)
\]
Let $\Sigma = \{a, b\}$.

The universal relation on $\Sigma^*$ is given by

$$( (\varepsilon a) + (\varepsilon b) + (a \varepsilon) + (b \varepsilon) )^* = \{ (x y) \mid x, y \in \Sigma^* \}$$

The identity relation on $\Sigma^*$ is given by

$$( (a a) + (b b) )^* = \{ (x x) \mid x \in \Sigma^* \}$$

Exercise

Show that the un-equal relation is rational.
By definition, a rational relation has the form

$$\rho \subseteq \Sigma^* \times \Gamma^*$$

But we may also think of such a relation as a so-called transduction, a map

$$\rho : \Sigma^* \longrightarrow \mathcal{P}(\Gamma^*)$$

This is the reason for the $x/y$ notation: think of $x$ as being mapped to $y$.

Of course, there is no fundamental difference between the two interpretations. We will switch back and forth as convenient.
We can also use transition systems to describe rational relations, but this time the labels will be from $M = \Sigma^* \times \Gamma^*$.

Thus the transitions look like

$$p \xrightarrow{x/y} q$$

where $x \in \Sigma^*$ and $y \in \Gamma^*$. As for ordinary transition systems, we can define the label of a path $\pi$

$$\pi = p_0 \xrightarrow{x_1/y_1} p_1 \xrightarrow{x_2/y_2} p_2 \ldots p_{n-1} \xrightarrow{x_n/y_n} p_n$$

in the diagram as the product of the respective labels in the monoid:

$$\text{lab}(\pi) = (x_1x_2 \ldots x_n, y_1y_2 \ldots y_n) \in M.$$
As usual, we will save a tree and only deal with $k = 2$.

**Definition**

A **transducer** is an automaton $\mathcal{T} = \langle S; I, F \rangle$ where $S$ is a finite transition system over $\Sigma^* \times \Gamma^*$; the acceptance condition is given by $I, F \subseteq Q$.

The **transduction** associated with $\mathcal{T}$ is the relation

$$L(\mathcal{T}) = \{ \text{lab}(\pi) \mid \pi \text{ path from } I \text{ to } F \} \subseteq \Sigma^* \times \Gamma^*$$

One also speaks about the **behavior** of $\mathcal{T}$, written $[\mathcal{T}]$, and many say that $\mathcal{T}$ recognizes $L(\mathcal{T})$. 


We already know that identity $x = y$ is a rational relation. Here is a transducer whose behavior is the relation $x \neq y$.

In the diagram, $a$ and $b$ are supposed to range over $\Sigma$, and $a \neq b$. * means eternal bliss.
Theorem

A relation is rational if, and only if, it is the behavior of a (finite) transducer.

The proof is an exact re-run of the argument for regular languages. A careful inspection of the argument shows that all one needs is labels chosen from a monoid; the fact that the monoid is free in the language case plays no role.

Exercise

Write out a detailed proof of the theorem.
Another good way to think about this is to consider finite state machines with two read-only, uni-directional input tapes (for $x$ and $y$ respectively). There are two separate input heads that read from these tapes. Since we are interested in memoryless devices, there are no work tapes.

One important point here is that the heads can move independently; in particular, one head can get arbitrarily far ahead of the other.

We can express the two-tape situation via a transition relation of the type

$$\tau \subseteq Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \times Q$$

A similar approach works for $k$-tape machines.
- Model Checking
- Rational Relations
- Properties of Rat
- Synchronous Relations
- Model Checking Automatic Structures
Wurtzelbrunft remembers the Banach quote about analogies and immediately concludes:

Every result about regular languages carries over, *mutatis mutandis*, to rational relations.

After all, it’s just about the same Kleene algebra we are working in, so what could possibly change? For example, we should be able to come up with a nice machine model, figure out how to determinize and minimize these devices, and so on.

Fortunately, life is so much more interesting than that.

Some results do indeed carry over, almost verbatim. But others a plain false and one has to be very careful.
Consider the binary rational relations

\[ A = (\frac{a}{c})^* (\frac{b}{\varepsilon})^* \quad B = (\frac{a}{\varepsilon})^* (\frac{b}{c})^* \]

Then

\[ A \cap B = \{(\frac{a^i b^i}{c^i}) \mid i \geq 0\} \]

It is easy to see that the intersection cannot be recognized by a finite state transducer, essentially for the same reasons that \( \{a^i b^i \mid i \geq 0\} \) fails to be regular.

**Exercise**

*Prove that \( A \cap B \) really fails to be rational.*
Rational relations are closed under union by definition.

So the last result shows that we fail to have closure under intersection and complement.

Remember that we ultimately want to tackle first-order logic over simple structures, so this looks like a total fiasco. Indeed, we will have to adjust our definitions in a while.

But for the time being, let’s stick with rational relations.
Example

If $K \subseteq \Sigma^*$ and $L \subseteq \Gamma^*$ are regular, then $K \times L$ is rational.

Example

If $\rho \subseteq \Sigma^* \times \Gamma^*$ is rational, then $\text{supp}(\rho) \subseteq \Sigma^*$ and $\text{rng}(\rho) \subseteq \Gamma^*$ are regular.

Example

All the relations “$x$ is a prefix of $y$”, “$x$ is a suffix of $y$”, “$x$ is a factor of $y$” and “$x$ is a subword of $y$” are rational.

Example

Recall the definition of shuffle:

\[
\varepsilon \parallel y = y \parallel \varepsilon = \{y\}
\]
\[
xa \parallel yb = (x \parallel yb) a \cup (xa \parallel y) b.
\]

So $x \parallel y$ is the set of all possible interleavings of the letters of $x$ and $y$ (preserving relative order).

The map $(x, y) \mapsto x \parallel y$ is rational.
Disregarding state complexity, in the world of regular languages, there is no difference between NFAs and DFAs: nondeterminism does not increase the power of the machines.

One might wonder if there is some notion of deterministic rational relation and a corresponding deterministic transducer.

The basic idea is simple: there should be at most one computation on all inputs.

Unfortunately, the technical details are a bit messy (use of endmarkers) and we’ll skip this opportunity to inflict mental pain on the student body.
Consider the binary rational relations

\[ A = \left( \frac{aa}{b} \right)^* \quad B = \left( \frac{a}{bb} \right)^* \]

It is clear that both \( A \) and \( B \) are deterministic rational relation.

Now consider

\[ A \cup B = \left\{ \left( \frac{a^i}{b^j} \right) \mid i = 2j \lor j = 2i \right\} \]

For the union, your intuition should tell you that nondeterminism is critical: initially, we don’t know which type of test to apply. This indicates that determinization is not going to work in general for rational relations (which is to be expected since we already know that complementation fails in general).
Consider the binary relation $<_\text{len}$ on $\Sigma^*$ defined by

$$x <_{\text{len}} y \iff |x| < |y|.$$ 

We obtain a strict pre-order called length order; the corresponding classes of indistinguishable elements are words of the same length.

Given an ordered alphabet $\Sigma$ consider the binary relation $<_s$ on $\Sigma^*$ defined by

$$x <_s y \iff \exists a < b \in \Sigma, u, v, w \in \Sigma^* (x = uav \land y = ubw)$$

This produces another strict pre-order, the so-called split order; this time indistinguishable words are prefixes of one another.
Again assume an ordered alphabet $\Sigma$. The lexicographic order is a mix of prefix order and split order:

$$x <_\ell y \iff x \sqsubseteq y \lor x <_s y$$

Here $x \sqsubseteq y$ means that $x$ is a proper prefix of $y$. Note that lexicographic order is a total order, there are no indistinguishable elements.

**Proposition**

*Length order, split order and lexicographic order are all rational.*

**Exercise**

*Construct rational expressions that prove the proposition.*
Another important way of ordering words is the product order of length order and lexicographic order, the so-called length-lex order.

\[
x \ll y \iff x <_{\text{len}} y \lor (|x| = |y| \land x <_{\ell} y)
\]

Length-lex order is easily seen to be a well-order and there are many algorithms on strings that are naturally defined by induction on length-lex order.

Needless to say, length-lex order is also rational.
Usually one thinks of concatenation as a binary operation. But we can also model it as a ternary relation $\gamma$:

$$\gamma(x, y, z) \iff x \cdot y = z$$

**Proposition**

*Concatenation is rational.*

**Proof.** For simplicity assume $\Sigma = \{a, b\}$

$$\gamma = (a:\varepsilon:a + b:\varepsilon:b)^* \cdot (\varepsilon:a:a + \varepsilon:b:b)^*$$
Consider the ternary relation $\alpha$ on $\mathbb{2}$ defined by

$$\alpha(x, y, z) \iff \text{bin}(x) + \text{bin}(y) = \text{bin}(z)$$

where $\text{bin}(x)$ is the numerical value of $x$ assuming the LSD is first (reverse binary).

**Proposition**

*Binary addition in reverse binary is rational.*

**Proof.** The kindergarten algorithm for addition shows that $\alpha$ is rational. \qed

**Warning:** there is no analogous result for multiplication (for reverse binary encoding; but beware of exotic encodings).
Here is a central result: rational relations are closed under composition. Suppose we have two binary relations $\rho \subseteq \Sigma^* \times \Gamma^*$ and $\sigma \subseteq \Gamma^* \times \Delta^*$. Their composition $\tau = \rho \circ \sigma \subseteq \Sigma^* \times \Delta^*$ is defined to be the binary relation

$$x \tau y \iff \exists z (x \rho z \land z \sigma y)$$

**Theorem (Elgot, Mezei 1965)**

*If both $\rho$ and $\sigma$ are rational, then so is their composition $\rho \circ \sigma$.***
Assume we have transducers $A$ and $B$ for $\rho$ and $\sigma$, respectively. We may safely assume that the labels in $A$ have the form $a/\varepsilon$ or $\varepsilon/b$ where $a \in \Sigma$, $b \in \Gamma$; likewise for $B$. Add self-loops labeled $\varepsilon/\varepsilon$ everywhere.

We construct a product automaton $C$ with transitions

$$(p, q) \xrightarrow{a/c} (p', q')$$

whenever there are transitions $p \xrightarrow{a/b} p'$ and $q \xrightarrow{b/c} q'$ in $A$ and $B$, respectively, for some $a \in \Sigma_\varepsilon$, $b \in \Gamma_\varepsilon$ and $c \in \Delta_\varepsilon$.

Initial and final states in $C$ are $I_1 \times I_2$ and $F_1 \times F_2$. It is a labor of love to check that $C$ accepts $x/z$ if, and only if, $x \rho y$ and $y \sigma z$ for some $y \in \Gamma^*$. $\square$
Let $\rho = \left(\frac{a}{bb}\right)^*$ and $\sigma = \left(\frac{b}{\varepsilon}\right)\left(\frac{b}{c}\right)^*$; thus $\rho \circ \sigma = \left(\frac{a}{c}\right)\left(\frac{a}{cc}\right)^*$. Here are the two machines, without the $\varepsilon/\varepsilon$ self-loops.
And here is the product.

Of course, there is a “better” transducer, but this is the one obtained by blind application of the algorithm.
Here is another important closure property. Suppose $\rho$ is a $k$-ary relation on words. We define the projection of $\rho$ to be

$$\rho'(x_2, \ldots, x_k) \iff \exists z \, \rho(z, x_2, \ldots, x_k)$$

**Lemma**

*Whenever $\rho$ is rational, so is its projection $\rho'$.*

**Proof.**

Erase the first track in the $k$-track alphabet:

$$p \xrightarrow{a_1, a_2, \ldots, a_k} q \leadsto p \xrightarrow{a_2, \ldots, a_k} q$$

That’s it! Of course, the new machine will be nondeterministic in general. $\square$
One might wonder what happens when we move to the transitive reflexive closure $\text{tcl}(\rho)$. Recall that

$$\text{tcl}(\rho) = \bigsqcup_{k} \rho^{o_k}$$

where $\rho^{o_k}$ indicates the standard iterate, the $k$-fold composition of $\rho$ with itself.

**Mental Health Warning:** Unfortunately, the transitive closure is often written $\rho^*$, in direct clash with the standard notation for the Kleene star of a relation.

Alas, the two are quite incompatible. For example, let $\rho$ be lexicographic order. Clearly, $\text{tcl}(\rho) = \rho$.

But $ab \rho^* aabb$ since $a \rho aa$ and $b \rho bb$. So Kleene star clobbers the order completely.
Theorem

The transitive closure $\text{tcl}(\rho)$ of a rational relation is semidecidable.

Proof.

By definition $x \text{ tcl}(\rho) y$ iff $\exists k \ (x \ \rho^o_k \ y)$.

Obviously, $\rho^o_k$ is primitive recursive, uniformly in $k$.

So we are conducting an unbounded search over a primitive recursive relation; semidecidability follows. \qed
What would happen if we add tcl to the closure operations that produce the rational relations?

**Theorem**

*Adding tcl to the closure operations produces precisely all semidecidable relations.*

**Proof.**

Clearly every relation obtained this way is semidecidable.

For the opposite direction, note that the one-step relation of a Turing machine is rational, even synchronous, see below.

Then transitive closure is all that is needed to produce any semidecidable relation.
Recall the infamous Collatz problem: Does the following program halt for all $x \geq 1$?

```java
while( x > 1 ) // x positive integer
    if( x even )
        x = x/2;
    else
        x = 3 * x + 1;
```

If we write $x$ in reverse binary, and right-pad with 00, the transducer on the last slide computes one execution of the loop body.

So iterating the composition of the trivial $x \mapsto x00$ and the transducer leads to an open problem in number theory.
Life should be much easier with **length-preserving** relations:

\[ x \tau y \Rightarrow |x| = |y| \]

In fact, let’s only consider the functional case: we are just iterating a map \( y = \tau(x) \).

But if \( \tau \) is length-preserving then all orbits must be finite, in fact they cannot be longer than \( |\Sigma||x| \).

**Theorem**

*For length-preserving transductions, transitive closure is PSPACE-complete in general.*
Recall that $x^{op}$ is the word $x$ written backwards.

It is clear that the map $x \mapsto x^{op}$ cannot be computed by a FSM.

But iteration of a length-preserving transduction can be used to “compute” $x^{op}$ as follows.

Define a new alphabet $\Gamma = \Sigma \cup \{ \overline{a} \mid a \in \Sigma \}$.

There is a length-preserving rational function $\tau$ such that $\tau(\varepsilon) = \varepsilon$ and

$$\tau(auv) = u\overline{a}v$$

where $au \in \Sigma^*$ and $v \in \overline{\Sigma}^*$. Let $f$ be the homomorphism $f(\overline{a}) = a$. Then

$$x^{op} = f \left( x \text{ tcl}(\tau) \cap \overline{\Sigma}^* \right)$$
But as soon as we try to make a global statement, decidability vanishes. For example:

**Proposition**

*It is undecidable whether all orbits of a functional length-preserving transduction end in a fixed point.*

**Sketch of proof.**

Simulate a register machine without input, operating on bounded memory. Set things up so that all orbits end in a fixed point iff the register machine computation diverges.

So the fixed point means: the computation has run out of space. If, on the other hand, the computation converges for some sufficiently long initial setup, then we periodically repeat the whole computation.

In fact this problem turns out to be co-r.e.-complete.
- Model Checking
- Rational Relations
- Properties of Rat

4 Synchronous Relations

- Model Checking Automatic Structures
Rational relations in general are just a little too powerful for our purposes, we need to scale back a bit.

One sledge-hammer restriction is to insist that all the relations are length-preserving. In this case we have $\rho \subseteq (\Sigma \times \Gamma)^*$, so we are actually dealing with words over the product alphabet $\Sigma \times \Gamma$. These can be checked by an ordinary FSM over this alphabet:

$$
\begin{array}{cccc}
  x_1 & x_2 & \ldots & x_n \\
  y_1 & y_2 & \ldots & y_n \\
\end{array}
$$

Nothing new here, a length-preserving relation is rational iff it is regular as a language over $\Sigma \times \Gamma$. 
There is one particularly simple type of transducer that is often useful to recognize length-preserving relations. In a Mealy machine, the transitions are described by a function

$$\delta : Q \times \Sigma \rightarrow \Gamma \times Q.$$ 

The idea is that transitions are labeled by pairs in $\Sigma \times \Gamma$, so each input letter is transformed into an output letter.

And, the transitions are deterministic.

Exercise

Concoct a binary Mealy machine that implements the successor function modulo $2^k$ on words of length $k$ (using reverse binary representation).
Alas, length-preserving relations are bit too restricted for our purposes. To deal with words of different lengths, first extend each component alphabet by a padding symbol #: $\Sigma_\# = \Sigma \cup \{\#\}$ where $\# \notin \Sigma$.

The alphabet for “two-track” words is $\Delta_\# = \Sigma_\# \times \Gamma_\#$.

This pair of padded words is called the convolution of $x$ and $y$ and is often written $x:y$.

\[
x:y = \begin{array}{cccccc}
  x_1 & x_2 & \cdots & x_n & \# & \cdots & \#
  \\
y_1 & y_2 & \cdots & y_n & y_{n+1} & \cdots & y_m
\end{array}
\]

Another example of bad terminology, convolutions usually involve different directions.
Note that we are not using all of $\Delta^*_\#$ but only the regular subset coming from convolutions. For example,

\[
\begin{array}{ccc}
  a & \# & b & \#
  \\
  a & b & a & \#
\end{array}
\]

is not allowed.

As always, a similar approach clearly works for $k$-ary relations

$$\Sigma_1,\# \times \Sigma_2,\# \times \ldots \times \Sigma_k,\#$$

**Exercise**

*Show that the collection of all convolutions forms a regular language.*
Here is an idea going back to Büchi and Elgot in 1965.

**Definition**

A relation $\rho \subseteq \Sigma^* \times \Gamma^*$ is synchronous or automatic if there is a finite state machine $A$ over $\Gamma$ such that

$$L(A) = \{ x:y \mid \rho(x, y) \} \subseteq \Delta_*^*$$

$k$-ary relations are treated similarly.

Note that this machine $A$ is just a language recognizer, not a transducer: since we pad, we can read one symbol in each track at each step.

In a sense, synchronous relations are the most basic examples of relations that are not entirely trivial.

By contrast, one sometimes refers to arbitrary rational relations as asynchronous.
- Lexicographic order is synchronous.
- The prefix-relation is synchronous.
- The ternary addition relation is synchronous.
- The suffix-relation is not synchronous.
- The relations “$x$ is a factor of $y$” and “$x$ is a subword of $y$” are not synchronous.
Our motivation for synchronous relations was taken from length-preserving relations: it is plausible that two words of the same length should be processed in lock-step fashion. The justification for this idea is the following result.

**Theorem (Elgot, Mezei 1965)**

*Any length-preserving rational relation is already synchronous.*

The proof is quite messy, we’ll skip. Writing up a nice and elegant proof would be reasonable project.
Claim

*Given two $k$-ary synchronous relations $\rho$ and $\sigma$ on $\Sigma^*$, the following relations are also synchronous:*

$$\rho \sqcup \sigma, \rho \sqcap \sigma, \rho - \sigma,$$

The proof is very similar to the argument for regular languages: one can effectively construct the corresponding automata using the standard product machine idea.

This is a hugely important difference between general rational relations and synchronous relations: the latter do form an effective Boolean algebra, but we have already seen that the former are not closed under intersection (nor complement).
Synchronous relations are not closed under concatenation (or Kleene star). For example, let

\[ \rho = (a)^* (\varepsilon)^* \]
\[ \sigma = (b)^* \]

Then both \( R \) and \( S \) are synchronous, but \( \rho \cdot \sigma \) is not (the dot here is concatenation, not composition).

**Exercise**

*Prove all examples and counterexamples.*
On the upside, synchronous relations are closed under composition.

Suppose we have two binary relations $\rho \subseteq \Sigma^* \times \Gamma^*$ and $\sigma \subseteq \Gamma^* \times \Delta^*$.

**Theorem**

*If both $\rho$ and $\sigma$ are synchronous relations then so is their composition $\rho \circ \sigma$.***

**Exercise**

*Prove the theorem.*
More good news: synchronous relations are closed under projections.

**Lemma**

*Whenever \( \rho \) is synchronous, so is its projection \( \rho' \).*

The argument is verbatim the same as for general rational relations: we erase a track in the labels.

Again, this will generally produce a nondeterministic transition system even if we start from a deterministic one. If we also need complementation to deal with logical negation we may have to deal with exponential blow-up.
Model Checking

Rational Relations

Properties of Rat

Synchronous Relations

Model Checking Automatic Structures
To simplify matters, suppose we are looking at a structure over the alphabet $\mathbb{2}$ with just one binary relation:

$$\mathcal{E} = \langle C, \rightarrow \rangle$$

In fact, $\rightarrow$ could be a function, interpreted as a binary relation.

Just to be clear: this scenario includes Turing machines, we could easily code up the one-step relation over a binary alphabet.

So we cannot ask about orbits, only questions that can be phrased in FOL with a single binary predicate (and equality).
\[ \forall x, y, z (x \rightarrow y \land x \rightarrow z \Rightarrow y = z) \]

\[ \forall x, y, z (x \rightarrow y \land z \rightarrow y \Rightarrow x = z) \]

\[ \forall x \exists y (x \rightarrow y) \]

\[ \exists x, y, z (x \rightarrow y \land y \rightarrow z \land z \rightarrow x \land x \neq y) \]

\[ \forall x (\exists y, z ((y \rightarrow x \land z \rightarrow x \land y \neq z) \land \forall u (u \rightarrow x \Rightarrow u = y \lor y = z)) \]
So suppose we have the finite state machines describing $\mathcal{C} = \langle C, \rightarrow \rangle$ and some FO sentence $\varphi$.

As always, we may assume that quantifiers use distinct variables and that the formula is in prenex-normal-form, say:

$$\exists x_1 \forall x_2 \forall x_3 \ldots \exists x_k \varphi(x_1, \ldots, x_k)$$

The matrix $\varphi(x_1, \ldots, x_k)$ is quantifier-free, so all we have there is Boolean combinations of atomic formulae.
In our case, there are only two possible atomic cases:

- $x_1 = x_2$
- $x_1 \rightarrow x_2$.

Given an assignment for $x_1$ and $x_2$ (i.e., actual strings) we can easily test these formulae using two synchronous transducers $A_\leftarrow$ and $A_\rightarrow$. 
Back to the matrix, the quantifier-free formula

$$\varphi(x_1, x_2, \ldots, x_k)$$

with all variables as shown. We construct a $k$-track machine by induction on the subformulae of $\varphi$.

The atomic pieces read from two appropriate tracks and check $\rightarrow$ or $=$.

More precisely, we use variants $A_{\rightarrow,i,j}$ and $A_{=,i,j}$ of the machines from above that check that track $i$ evolves to track $j$ in one step or check equality.

These machines are all defined over the join alphabet $2^k$. Note that the alphabet grows exponentially with $k$, so there is an efficiency problem.
Suppose $\varphi = \psi_1 \land \psi_2$ with corresponding machines $A_{\psi_1}$ and $A_{\psi_2}$. We can use a product machine construction to get $A_\varphi$.

Disjunctions are even easier: take the disjoint union.

But negations are potentially expensive: we have to determinize first.

At any rate, we wind up with a composite automaton $A_\varphi$ for the whole matrix:

$$L(A_\varphi) = \{ u_1 : u_2 : \ldots : u_k \in (2^k)^* \mid C \models \varphi(u_1, u_2, \ldots, u_k) \}$$
It remains to deal with quantifiers, say

$$\exists x \psi(x)$$

We have a machine $A_\psi$ that has a track for variable $x$ within alphabet $2^\ell$.

Simply erase the $x$-track from all the transition labels.

This corresponds exactly to existential quantification, done!

For universal quantifiers we use the old equivalence $\forall \equiv \neg \exists \neg$.

This is all permissible, since projections and negations do not disturb automaticity.
In the process of removing quantifiers, we lose a track at each step:

\[ \mathcal{L}(\mathcal{B}_\varphi) = \{ u_1:u_2:...:u_\ell \in 2^*_\ell \mid \mathcal{C} \models \varphi(u_1,u_2,...,u_\ell) \} \]

In the end \( \ell = 0 \), and we are left with an unlabeled transition system. This transition system has a path from \( I \) to \( F \) iff the original sentence \( \varphi \) is valid.

So the final test is nearly trivial (DFS anyone?), but it does take a bit of work to get there.
Efficiency

- $\lor$ and $\exists$ are linear if we allow nondeterminism.

- $\land$ is at most quadratic via a product machine construction.

- $\neg$ is potentially exponential since we need to determinize first.
  Note that universal quantifiers produce two negations.

So this is a bit disappointing: we may run out of computational steam even when the formula is not terribly large.

A huge amount of work has gone into streamlining this and similar algorithms to deal with instances that are of practical relevance.
The question whether there is a $k$-cycle under $\rightarrow$ requires a bit of care. E.g., the brute-force version of the sentence $\varphi$ for $k = 3$ looks like

$$\exists x, y, z \,(x \rightarrow y \wedge y \rightarrow z \wedge z \rightarrow x \wedge x \neq y \wedge x \neq z \wedge y \neq z)$$

But note that checking for each inequality doubles the size of the machine, so we get something 8 times larger than the machine for the raw 3-cycle. It is much better to realize that the last formula is equivalent to

$$\exists x, y, z \,(x \rightarrow y \wedge y \rightarrow z \wedge z \rightarrow x \wedge x \neq y)$$

Exercise

*Figure out how to deal with $k$-cycles for arbitrary $k$.***
So, based on the better formula, we use the 3-track alphabet \( \Sigma = 2 \times 2 \times 2 \).

Given \( \mathcal{A} \) we concoct a product machine for the conjunctions:

\[
\mathcal{C}_3 = \mathcal{A}_{\rightarrow,1,2} \times \mathcal{A}_{\rightarrow,2,3} \times \mathcal{A}_{\rightarrow,3,1} \times \mathcal{D}_{1,2}
\]

where \( \mathcal{A}_{\rightarrow,i,j} \) tests if the word in track \( i \) evolves to the word in track \( j \).

Machine \( \mathcal{D}_{i,j} \) tests if the word in track \( i \) is different from the word in track \( j \) and consists essentially of two copies of \( \mathcal{A}_{\rightarrow,1,2} \times \mathcal{A}_{\rightarrow,2,3} \times \mathcal{A}_{\rightarrow,3,1} \).
So we get a machine $C_3$ that is roughly cubic in the size of $A_{\rightarrow}$ (disregarding possible savings for accessibility).

Once $C_3$ is built, we erase all the labels and are left with a digraph (since $\varphi$ has no universal quantifiers there is no problem with negation).

This digraph has a path from an initial state to a final state if, and only if, there is a 3-cycle under $\rightarrow$.

Note, though, how the machines grow if we want to test for longer cycles: the size of $C_k$ is bounded only by $m^k$, where $m$ is the size of $A_{\rightarrow}$, so this will not work for long cycles. And, we need several products with $D_{i,j}$. 


Again, the machines produced by our algorithm tend to be very large, so one has to be careful to deal with state-complexity.

One way of getting smaller machines is to rewrite the formula under consideration by hand as in the 3-cycle example. A logically equivalent formula may produce significantly smaller machines. Unfortunately, it can be quite difficult to find better ways to express a first-order property.

If the outermost block of quantifiers is universal, the last check can be more naturally phrased in terms of Universality rather than Emptiness. In this case one should try to use Universality testing algorithms without complementation.
We can easily augment our decision machinery by adding additional predicates so long as these predicates are themselves synchronous.

This can be useful as a shortcut: instead of having a formula that defines some property (which then is translated into an automaton), we just build the automaton directly and in an optimal way.

Interestingly, this trick can also work for properties that are not even definable in FOL. We can extend the expressibility of our language and get smaller machines for the logic part that way.