# Deciding Rational Property Definitions

Matthew Rafuse

April 4 2019

Degree of Bachelor of Science, Double Honours in Computer Science

Dr. Stavros Konstantinidis - Professor, Supervisor

Dr. Mitja Mastnak - Professor, Reader

**Abstract**

Language Equations is an area of Theoretical Computer Science that is concerned with expressing desirable constraints on formal languages of interest and answering algorithmic problems about such languages. Language Equations can be used as a means to express desirable properties a language can have. The LaSer system allows users to supply a regular language and a property described by a given transducer or trajectory, and return whether or not the language satisfies the given property. An important potential feature of LaSer is the ability to supply a rational language equation to LaSer, and allow it to decide this equation, that is, to test that the equation holds. This functionality has been implemented, and will be outlined in this paper.

# Contents

## 1 Introduction

Language Equations [1, 16] is an area of Theoretical Computer Science that is concerned with expressing desirable constraints on formal languages of interest and answering algorithmic problems about such languages. Like a polynomial equation $p(x) = 0$ involving numbers and arithmetic operations, a language equation $E(L) = \emptyset$ involves an unknown language L, constant languages, standard language operations and possibly application of transducers to the languages involved. A language equation is rational when the constant languages involved are regular, and the operations involved preserve regularity.

Language Equations can be used as a means to express desirable properties a language can have. The LaSer system maintained at Saint Mary's University allows users to supply a regular language and a property described by a given transducer or trajectory, and return whether or not the language satisfies the given property. In addition, the LaSer system can answer questions regarding maximality and language construction [14], but this paper will focus solely on the satisfaction functionality of LaSer.

A desirable feature of LaSer is the ability for users to supply a rational language equation to LaSer and allow LaSer to decide this language equation. Instead of needing to implement each new desirable equation directly into LaSer, the system can instead parse an equation supplied to it, allowing for the user to decide any type of equation. The equation can make use of regular expressions, automata, transducers and trajectories to describe more complex equations, and potentially even decide if a language satisfies multiple properties at once. This is the functionality that has been implemented, and will be outlined in this thesis.

In this thesis we discuss functionality added to the LaSer system. In section 2, we begin with a basic overview of the building blocks of the system, including the mathematical concepts of languages, properties and their various representations in LaSer. In Section 2.6.2, we discuss one of the foremost use cases for the new functionality. In Section 3, we outline the software tools used in the implementation. In Section 4 we move on to discuss the history and limitations of the previous implementation. In Sections 5 and 6, we discuss the changes to the program to enable the new functionality. In Section 7, we discuss what the new system can accomplish, and how. In Section 8, we then discuss some directions of future research. Included in the appendices are some proofs, as well as a more thorough breakdown of the parsing functionality introduced.

# 2 Math Tools & Background

To allow for a wider readership, we will quickly give an overview of some of the concepts explored in this thesis. More advanced readers can skip the more basic definitions and pick up the paper in section 4.

## 2.1 Languages

An **alphabet** is a set of characters used to form words i.e., $\{a, b\}$. We denote an alphabet by $\Sigma$ [21]. $\Sigma^*$ is the set of all words obtained by concatenating letters of the alphabet - i.e., for $\Sigma = \{a, b\}$ we have the words

$$abbba, baabbb, ab, a \in \Sigma^*$$

A **language** is any set of words over $\Sigma$, that is, any subset of $\Sigma^*$, A language can be finite or infinite. Let $K, L$ be languages, $\epsilon$ be the **empty** word. Some important operations on languages are:

1. Complement - $L^c = \Sigma^* \setminus L$
2. Concatenation - $KL = \{uv : u \in K, v \in L\}$
3. Power - $L^n = \{u_1 u_2 ... u_n : \text{each } u_i \in L\}$, $L^0 = \epsilon$
4. Kleene Star - $L^* = L^0 \cup L^1 \cup ... = \bigcup_{i=0}^{\infty} L^i$

Each of these operations can be performed in LaSer, under the new system. Some examples of languages include:

1. $L = \{a, b, ab, ba, aa, bb\}$ - all words over the alphabet $\{a, b\}$ of length $\leqslant 2$.
2. $L = \{a, b\}^5$ - all words over the alphabet $\{a, b\}$ of length 5.
3. $L = (\{0, 1\}^2)^*$ - all binary words of even length.

## 2.2 Regular Expressions

Regular expressions are a way to describe a language in a concise manner [21]. If a language can be described by a regular expression, it is called a **regular language**. A regular expression is a string over the alphabet

$$\Sigma \cup \{(,),+,*,\oslash\}$$

where $\Sigma$ is an alphabet. A **regular expression** is defined inductively as follows:

1. $\oslash$ and all $\sigma \in \Sigma$ are regular expressions.

2. if $t, r_1, r_2$ are regular expressions, then also $(t^*), (r_1 r_2), (r_1 + r_2)$ are.

The language $L(r)$ of a regular expression $r$ is defined as follows, where $t, r_1, r_2$ are regular expressions:

1. $r = \oslash \rightarrow L(r) = \emptyset$

2. $r = \sigma \rightarrow L(r) = \{\sigma\}, \sigma \in \Sigma$

3. $r = t^* \rightarrow L(r) = (L(t))^*$

4. $r = (r_1 r_2) \rightarrow L(r) = L(r_1)L(r_2)$

5. $r = (r_1 + r_2) \rightarrow L(r) = L(r_1) \cup L(r_2)$

We additionally define:

1. $r = t^+ \rightarrow L(r) = L(t)L(t)^*$

2. $r = t^k \rightarrow L(r) = L(t)L(t)...L(t)$ where $L(t)$ is concatenated $k$ times.

Regular expressions are an efficient way of tersely defining relatively simple languages, and thus are supported by LaSer. It is important to note that LaSer permits us to omit parentheses using the following precedence of operators: *, concatenation, +. Additionally, it's interesting to note that regular expressions are a way of using a finite string to represent an infinite language

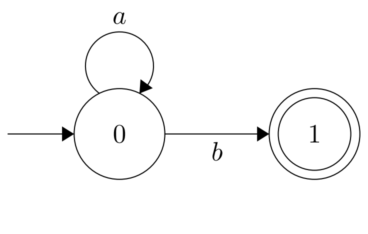
- i.e. $a^*$ is two characters but represents the infinite number of words containing only a's. The following are examples of regular expressions:

- $a^*b$ - defines the language consisting of the words that start with zero or more $a$'s followed by a $b$ - e.g. $aaaaab \in L(a^*b)$, $abaaab \notin L(a^*b)$
- $(a+b)^*b$ - defines the language consisting of the words that start with zero or more $a$ or $b$ followed by one $b$. In other words, this includes any words of length 1 or more that ends with a b - e.g. $aaaaab \in L((a+b)^*b)$, $abaaab \in L((a+b)^*b)$, $aba \notin L((a+b)^*b)$,
- $ab$ - defines the language consisting of the word $ab$.

## 2.3 Automata

Finite Automata are a construct used to define a language in theoretical computer science [18]. They are not only used to define a language however - they also permit us to algorithmically decide if given a word $w \in L$, where $L$ is the Language accepted by the automaton.

The basic structure of an automaton is a directed graph of nodes and connections between the nodes. We call these nodes **states**, and the connections **transitions**. Each transition between states is marked with a letter from the alphabet. Mathematically, transitions are a triple $(p, \sigma, q)$ where $p$ and $q$ are states and $\sigma \in \Sigma$. There is a **start state**, where we "enter" the automaton, and there can be several **final states**, where the automaton is completed. They are alternatively called **accepting states**. The automaton describes the language made up of the words formed by concatenating the labels of transitions leading from the start state to a final state. Below is a drawing of a basic automaton accepting the language $\{a\}^*\{b\}$.



We can see that the word $aab$ is formed by transitioning starting at state 0 to state 0, then to state 0 again, then finally to state 1. We denote this path using three transitions in the following format:

$$(0, a, 0), (0, a, 0), (0, b, 1)$$

The most used type of automaton is a Deterministic Finite Automaton, abbreviated from now on as DFA. An automaton is a DFA if no state has two transitions leading to different states with the same label - that is, at a given state there are no two transitions marked with the same label. This is in contrast to Non-deterministic Finite Automata, abbreviated as NFAs, which permit transitions with the same label leading to different states [18].

The rigorous definition of a DFA is a 5-tuple of the following structure:

$$(Q, \Sigma, q_0, \delta, F)$$

where $Q$ is the set of states, $\Sigma$ is an alphabet, $q_0 \in Q$ is the start state, $\delta : Q \times \Sigma \to Q$ is a function that describes the transitions of the automaton, and $F \subseteq Q$ denoting the final states. The transition function takes the current state and a label from the alphabet and returns the next state.

NFAs have a similar definition - a 5-tuple with the following structure:

$$(Q, \Sigma, q_0, T, F)$$

where $Q$, $\Sigma$, $q_0$, $F$ are as a above, and $T$ is a finite list of transitions $(q, \sigma, p) \in Q \times \Sigma \times Q$. Note we can no longer use a transition function as there can be multiple states we can go to with a given label from a given state.

Importantly, all DFAs are a special class of NFAs that happen to have just a single transition per label. Thus, every DFA can be represented as an NFA - this is fairly intuitive. Less intuitive, however, is the converse, which is also true. That is, every NFA can also be represented as a

DFA. This is important in the implementation of LaSer, since FAdo converts NFAs into DFAs in order to complement a language [5].

## 2.4 Transducers

Transducers are a construct similar to a Non-deterministic Finite Automaton [15]. They take a word as input and use the labels of transitions to return an output. However, where an NFA or DFA simply returns either YES or NO, a transducer instead returns nothing, or one of several possible words (transducers are generally non-deterministic). A popular application of this is modelling data transmission errors, particularly error-detecting and error-correcting transducers, as discussed in Section 2.6.1 [8]. A transducer representing transmission errors takes in a word and returns all possible modifications of the word, where for example 1, 2, 3 or more letters in the word have been mutated. For example, limit this idea 1 letter modifications, and call this transducer $t$. Then $t(01)$ would be:

$$\{11, 00\}$$

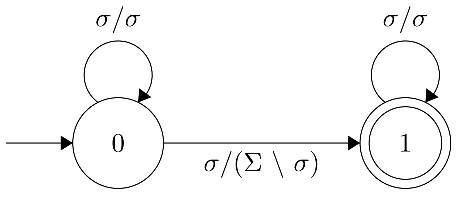
Because each of these words is obtained by modifying one letter of the word 01.

In general, if $w$ is a word then $t(w)$ is the set of possible outputs of $t$ when $w$ is used as input. A transducer is a 6-tuple

$$M = (Q, \Sigma, \Gamma, q_0, T, F)$$

where $Q$ is the set of states, $\Sigma$ and $\Gamma$ are the input and output alphabets, respectively, $q_0 \in Q$ is the start state, $T$ is the set of transitions of the form $(p, \sigma/w, q)$ such that $p$, $q$ are states in $Q$, $\sigma \in \Sigma \cup \{\epsilon\}$ (Where $\epsilon$ is the empty string), and $w \in \Gamma^*$, and finally $F \subseteq Q$ is the set of final states [15]. Note that this definition permits $\sigma = \epsilon$ and so we can add symbols to the input word. Additionally, it permits $w = \epsilon$, so we are permitted to remove letters from the input word.

We can think of transducers in the context of LaSer as a way to check a language for a specific attribute, or property [14]. Let us consider the language described by the regular expression $a^*b$, and $t$, the 1-substitution transducer - that is, a transducer that takes in a word and returns the set of all words that differ from the original word by a single letter. The 1-substitution transducer looks like:



where $\sigma \in \Sigma, \Sigma = \{a, b\}$. Then $t(a^*b)$ is the set of all words that differ from a word in $a^*b$ by one letter, e.g., $\{abab, aabb, aaaa\} \subseteq t(aaab)$, or $\{bb, aa\} = t(ab)$. In general, if $L$ is a language, then $t(L)$ is the language consisting of all words that are possible outputs of $t$ when a word from $L$ is used as input.

Transducers can be used to express properties (See Section 2.6) a language may have, and allow us to test that language for the property. LaSer allows the use of several different types of transducers, each type having a distinct application. This is discussed further in Section 2.6.1.

## 2.5 Trajectories

Trajectories, on their face, are significantly simpler to define than transducers. They are a regular expression where $\Sigma = \{0, 1\}$ [2], e.g., $0^*1^+ + 1^+0^*$ is a valid trajectory.

To understand trajectories, we must first understand the shuffle operation. The shuffle operation can be seen as a more general version of concatenation, defined by:

$$x \sqcup\!\sqcup\, y = \text{the set of words obtained by injecting the letters of y into x in order}$$

For a more rigorous definition, see [7, 2]. For example, consider $x = x_1x_2,\ y = y_1y_2$. Then

$$x \sqcup\!\sqcup\, y = \{x_1x_2y_1y_2,\ x_1y_1x_2y_2,\ x_1y_1y_2x_2,\ y_1y_2x_1x_2,\ y_1x_1y_2x_2,\ y_1x_1x_2y_2\}$$

We can restrict $\sqcup\!\sqcup$ to a set of specific orderings using a word over the binary alphabet, denoted as $\sqcup\!\sqcup_t$. To do this, we say that if the next symbol of the binary word is a 0, we insert the next symbol of $x$. If it is a 1, we insert the next symbol of $y$. For example, take $t = 0110$:

$$x \sqcup\!\sqcup_{0110} y = \{x_1 y_1 y_2 x_2\}$$

If t does not contain $|x|$ zeros and $|y|$ ones, the shuffle operation is not defined, and we return $\emptyset$. Furthermore, we can consider restricting the shuffle operation to a set of words, by $x \sqcup\!\sqcup_T y = \bigcup_{t \in T} x \sqcup\!\sqcup_t y$. For example, $T = \{0011, 1100\}$:

$$x \sqcup\!\sqcup_T y = \{x_1 x_2 y_1 y_2,\ y_1 y_2 x_1 x_2\}$$

From here, it is easy to see how regular expressions come into play. For brevity, we will instead express $T$ as a regular expression. For example, $T = 0^*1^*$ is the the regular expression for concatenation of two words, i.e., $x \sqcup\!\sqcup_T y = xy$. In this case, $T$ contains words such as 000111, or 0011111. Finally, we can expand the shuffle operation to operate on languages instead of single words. Consider two languages, $X$ and $Y$. Then we define:

$$X \sqcup\!\sqcup_T Y = \bigcup_{t \in T} \bigcup_{x \in X, y \in Y} x \sqcup\!\sqcup_t y$$

Trajectories can be used to specify a property, as discussed in Section 2.6.

## 2.6 Properties & Independences

In surveying various languages, one notices that many languges share similar traits - for example, we could consider languages which contain words only of length 3, or who contains only one instance of the letter a. These are fairly naturally known as properties.

A property $P$ is defined as a set of languages:

$$P \subseteq \{L : L \subseteq \Sigma^*\}$$

If $L \in P$, we say $L$ **satisfies** $P$. Normally, P is an infinite set of languages that satisfy a certain condition. For example, the Prefix Code Property is the set of all languages where no word in the language is the prefix of another word.

There are properties that exhibit more rigorous behaviour, called **independences**. Independences are properties $P$ that have the following trait:

$$L \in P \implies L' \in P, \forall L' \subseteq L$$

That is, if a language $L$ satisfies the property, all subsets of L will also satisfy the property. The Prefix Code Property is also an independence - if no two words in the language are prefixes of each other, clearly any subset of that language will have the same property.

We can also consider subsets up to a specific size - **n-independences** are independences such that:

$$L \in P \implies L' \in P,\ \forall L' \subseteq L,\ |L'| < n$$

That is, any $n-1$ subset of $L$ also satisfies the property. For example, the prefix code property is a 3-independence: Consider any two elements of the language $a^*b$. It is clear it is impossible for one to be a prefix of the other, since both words will end with a $b$, and that must be the final letter of the word.

There is a subset of 3-Independences that can be represented via the specific condition:

$$\forall w \in L : t(w) \cap (L - \{w\}) = \emptyset$$

Any language satisfying this condition given a transducer $t$ is called **t-independent** [9]. We say that the transducer $t$ **represents** the set of all $t$-independent languages.

Another important way to represent properties are via **language equations**. Language Equations are similar to polynomial equations. Like a polynomial equation $p(x) = 0$ involving numbers and arithmetic operations, a language equation $E(L) = \emptyset$ involves an unknown language L, constant languages, standard language operations and possibly application of transducers on the languages involved. A language equation is rational when the constant languages involved are regular, and the operations involved preserve regularity. We call a property a **rational language property** if it can be expressed via a rational language equation $E(L)$ :

$$P = \{L \subseteq \Sigma^* : E(L) = \emptyset\}$$

For example, the Comma-Free Code Property can be represented via the language equation

$$\Sigma^+ L \Sigma^+ \cap LL = \emptyset$$

### 2.6.1 Types of Properties

Different transducers can exhibit different behavior with regard to their output. For example, consider the transducer representing the Prefix Code Property in Figure 1. This transducer clearly does not return the input word ($w \notin t(w)$), whereas the transducer representing the Prefix Code Property in Figure 2 does ($w \in t(w)$). This fact was important in previous implementations of LaSer, as properties represented by transducers that differed on the allowance of $w \in t(w)$ used different conditions in LaSer. To understand this we will outline the different types of transducers supported by LaSer.

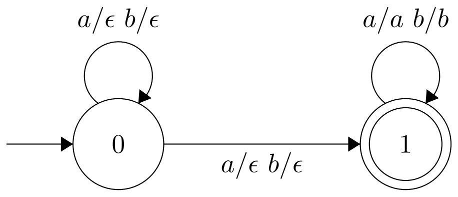


Figure 1: Input-Altering Transducer representing Prefix Code Property.

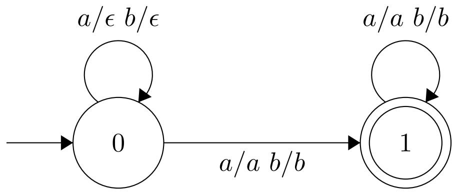


Figure 2: Input-Preserving Transducer representing Prefix Code Property.

**Input-Altering Properties** are the most basic type of property used in LaSer, and are represented by an Input Altering Transducer. An input-altering transducer is a transducer $t$ that does not return the input word $w$ in the output. That is, $w \notin t(w)$. This allows an Input Altering Property to be represented by a simplified equation:

$$t(L) \cap L = \emptyset$$

This satisfies the requirement for $t$-independence. If $t(L) \cap L = \emptyset$, clearly $\forall w \in L : t(w) \cap (L - \{w\}) = \emptyset$, because

$$\bigcup_{w \in L} t(w) = t(L)$$

and since $t(L) \cap L = \emptyset$, any subset of $t(L)$ also does not intersect with any subset of $L$.

However, an assumption made in this equation is that $\forall w \in L : w \notin t(w)$. So what if we want to represent a property with a transducer that contains the input word in the output?

**Input-Preserving Properties** are a type of property whose transducer representation allow the input word in the output. The condition that defines this type of property is as follows:

$$\forall w \in L : t(w) \cap (L - \{w\}) = \emptyset$$

This is exactly the necessary condition for a property to be a $t$-independence, and allows the input word to be in the output. These types of properties are also known as Error-Detecting Properties, as they have many applications in information processing or information transmission [8].

**Error-Correcting Properties** are another type of property. They are defined by the following condition:

$$\forall w \in L : t(w) \cap t(L - \{w\}) = \emptyset$$

What this is saying, is that any word obtained by applying a transducer $t$ to a word $w \in L$ will not be in the output of any other word in $L$ - i.e. the sets $t(w) : w \in L$ have no overlap ($\forall w, r \in L : t(w) \cap t(r) = \emptyset$).

This type of property is used extensively in information transmission, in order to counteract channel noise, uncertainty of word boundaries, and decoding cost of unique decodability [8].

It does not immediately follow that an Error-Correcting property is a $t$-independence, as the conditions the properties must satisfy don't match. In fact, an Error-Correcting Property represented by a transducer $s$ is an $(s^{-1} \circ s)$-independence. The proof of this is shown in Appendix A.

Something of note is that as discussed in [4], is that every Input-Altering Property is also an Error-Detecting Property, and every Error-Correcting Property is also an Error-Detecting Property.

### 2.6.2 Hierarchy of Independences

In order to understand part of the usefulness of the new implementation, we must disuss the relative size of different classes of independences. That is, it is important to note that every t-independence is a 3-independence. This is clear from the fact that transducers have essentially two channels - the input and output tapes. Another way to show this is to consider the equation again: $t(L) \cap L = \emptyset$. Breaking the notation down, we see this is equivalent to $\forall u, v \in L, u \notin t(v)$.

We can also show any n-independence is also an (n+1)-independence. Consider any given n-independence $P$. Take a subset $L$ of size n. Since any subset $L' \subseteq L$ (where $|L|$ = n-1) will satisfy the property, $L$ must also satisfy the property. Thus $P$ is also an (n+1)-independence.

The converse is not true in general. That is, (n+1)-independence $\not\Longrightarrow$ n-independence.

Note that the Comma Free Code Property is not a 3-independence like the Prefix Code Property. Consider the language $L = \{011, 111, 110\}$. Each subset of size $n = 2$ satisfies the property, but the entire set does not, since 011110 contains 111. Thus the comma free code cannot be a 3-independence, and we claim it is a 4-independence.

***Proof.*** The equation for the Comma-Free code is $\Sigma^+ L \Sigma^+ \cap LL = \emptyset$. Breaking that down: $\forall u, v, w \in L : uv \neq xwy$ for $x, y \in \Sigma^+$. We will show L is a CFC $\iff \forall L' \subseteq L \ni 0 < |L'| < 4$, $L'$ is a CFC.

$\implies$ : Assume L is a CFC, and take any $L' \subseteq L$ with $0 < |L'| < 4$. Take any $u, v, w \in L'$ and any $x, y \in \Sigma^+$. As $u, v, w \in L$ and $L$ is a CFC, $uv \neq xwy$. So $L'$ is a CFC.

$\impliedby$ : Assume that $\forall L' \subseteq L$ with $0 < |L'| < 4$, $L'$ is a CFC. Take any $u, v, w \in L$ and any $x, y \in \Sigma^+$. But $|\{u, v, w\}| < 4$, so $\{u, v, w\}$ is a CFC, therefore, $uv \neq xwy$. ■

Additionally, not all 3-independences are t-independences. A 3-independence not describable by any transducer (or not t-independent for any transducer t) is the reversal property:

$$\forall u, v \in L : u \neq v^R$$

where $v^R$ is the reversal of $v$ - i.e., $123^R = 321$, $1100^R = 0011$. This is because due to their structure, transducers have limits on what they can remember - a pushdown transducer (a transducer with a stack) can represent this property, but that is outside the scope of this thesis. For more information, refer to [3].

We can now start to see a hierarchy take shape.

$$t\text{-independences} \subset \text{3-independences} \subset \text{4-independences} \subset \ldots$$

The aim of this research now becomes clear. LaSer has been expanded to permit representation not only of t-independences, but also all rational language properties, vastly expanding the potential use of LaSer. It is important to note, however, that the new functionality does not render existing methods obselete - we conjecture that there are certain examples where the condition $\forall w : t(w) \cap (L - \{w\}) = \emptyset$ cannot be represented as a language equation.

# 3 Programming Tools & Background

## 3.1 FAdo

FAdo is a python library that aims to provide an open source extensible high-performance software library for the symbolic manipulation of automata and other models of computation. FAdo permits programmers to execute most standard procedures on regular languages. These procedures include union, intersection, disjoint, concatenation, kleene star, etc.

FAdo forms the backbone of LaSer and all of its functionality. Once the input from the user has been parsed into a FAdo object, we manipulate the object using FAdo's built in functionality. The latest improvement to FAdo involved closely linking Lark's parsing to FAdo's language operations.

`readOneFromString` [5] is a function in FAdo's `fio` module, which takes in a string representation of an automaton and returns the FAdo automaton. It is used to parse finite automata and transducers.

`str2regexp` [5] is a function in FAdo's `reex` module, which takes in a string representation of a regular expression and return a FAdo regular expression. It is used to parse regular expressions and trajectories.

There are also a number of classes and class methods used by the functionality. These functions are discussed in Appendix C.

## 3.2 Lark

Lark is an open source project that allows programmers to easily define context free grammars and parsing for these grammars [13]. In LaSer, it is used to parse rational property definitions, splitting the input into chunks for further parsing by Yappy Parser into FAdo structures. Lark forms the backbone of the latest improvement, allowing the creation of variables, parsing set operations in the correct order, and other operations.

Definition of the Grammar for Lark is done using EBNF format. Lark works in two main steps - parsing of the input into a parse tree, and using a transformer to evaluate the generated parse tree. The Parsing is handled by Lark, using the Grammar defined in Appendix B. From there, LaSer implements the Transformer class from Lark to evaluate the code and decide the equation, as outlined in Appendix C.

# 4 History

The following is an excerpt from the ”technical notes” section of the LaSer Site:

> This server accepts the description of a regular language via a finite automaton or regular expression and the description of a language property, and returns YES/NO, depending on whether the language satisfies the property. In case the answer is NO, appropriate counterexamples (also called witness words) are provided. If the language satisfies the property, the server can also return YES/NO, depending on whether the language is maximal with respect to the property. The Maximality question is PSPACE-hard. In the case of the Construction question [11], I-LaSer accepts the description of a property and three positive integers s, N, k, and generates a language of (up to) N words of length k satisfying the given property. The integer s must be less than 10 and specifies that the alphabet must be ’0’,’1’,...,’s’.

In LaSer, we can represent properties via transducers, thanks to research in [4]. Additionally, LaSer allows us to provide a trajectory to represent a property. A trajectory $t$ and a language $L$ are supplied. The Trajectory is interpreted where 0 represents the next letter of a word in $L$ and 1 is an element of the alphabet $\Sigma$ of L. Thus in FAdo, if $L = a^*b$ and $t = 0^*1^*$, we consider the equation

$$L \sqcup\!\sqcup_t \Sigma^+ \cap L = \emptyset$$

equivalent to

$$L\Sigma^* \cap L = \emptyset$$

which is the Prefix Code Property. This is significantly more terse than using the regular syntax to declare a transducer representing this property.

Prior to the work outlined in this thesis, LaSer was able to decide properties under very rigid conditions. In order for a property to be decided by LaSer, the user would supply one language and one property, as well as the type of the property (Input Altering, Error Detecting, Error Correcting, etc.). LaSer also allowed users to use one of several fixed properties - Prefix, Suffix, Infix, Outfix, Hypercode, and Code properties.

The user would input the language to test, either in a file or a text box, to be parsed by FAdo. Similarly, the user would then input a transducer in the same way. They would then submit the file, and LaSer would decide the equation.

# 5 Rearranging Program Structure

Before implementation of new features could start, it was necessary to restructure the program structure. This involved moving from a single monolithic function handling the entirety of LaSer's functionality to a larger number of smaller functions. In this way, new functionality could be focused in a single area of the application instead of needing to keep in mind the entirety of the program's structure.

Prior to this change, the bulk of functionality resided in only a few files:

1. `views.py` - Handled all server logic, routing requesting of all types from the server request into the correct FAdo functions.

2. `laserShared.py` - Utility functions for preparing input for FAdo.

The remainder of functionality was in need of restructuring and documentation. After cleaning up the rest of the codebase, the final structure was this:

```
app/
  templates/ - contains html templates for the given pages of LaSer
  testing/ - contains tests and test files for LaSer Unit Testing
  transducer/ - contains the bulk of the site, including the views and handlers that
       integrate with FAdo.
laser/ - contains configuration files for the server.
  apache_settings/ - contains configuration files
media/ - contains the output when programs are generated using LaSer
static/ - contains the static css/js files that help run the webpage
  css/ - css files
  js/ - js files
```

With these structural changes, implementation of the functionality outlined in the following section was significantly easier.

## 6 Grammar

This section will detail the parsing component of the new functionality - the Grammar written in Lark and how the Grammar is implemented and connects to the python code. We will break down the lark file and how it works - a line by line breakdown of the source code is available in Appendix B and Appendix C.

In order to visualize the process undertaken by the program, we will use a simple example. We will consider the Suffix Code as represented by a trajectory and the language $a^*b$:

```
L = /a*b/

t = 1*0*

t(L) & L
```

The first step is to run this code through our parser, lark. This will break down the source code into something we can process. After parsing, we will have a tree, which will look something like the tree in Figure 3.

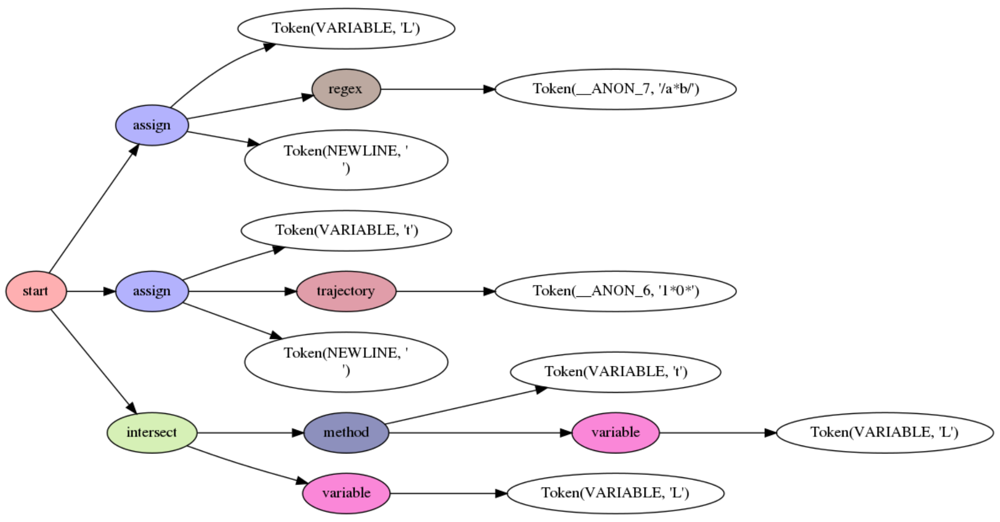


Figure 3: A graphical representation of Lark's parsed tree.

At this point, we use a `Transformer` from lark to run the computation. The first step is to parse the regular expression, which is done through FAdo, giving us the automaton in Figure 4.

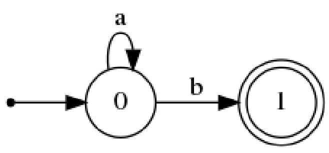


Figure 4: An automaton representing the regular expression $a^*b$.

The next step is to parse the trajectory, giving us the transducer in Figure 5.

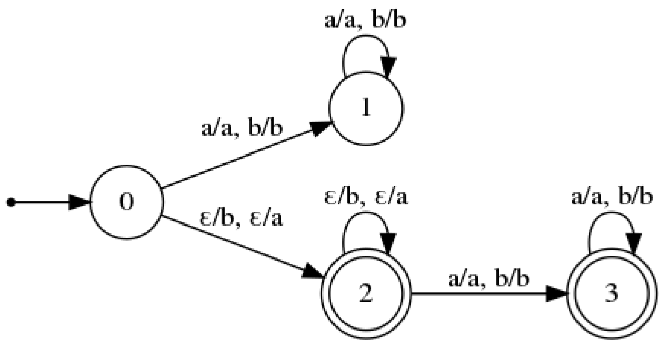


Figure 5: A transducer representing the trajectory $1^*0^*$.

We then run the transducer on the NFA (`t.runOnNFA(L)`), and get the automaton in Figure 6.

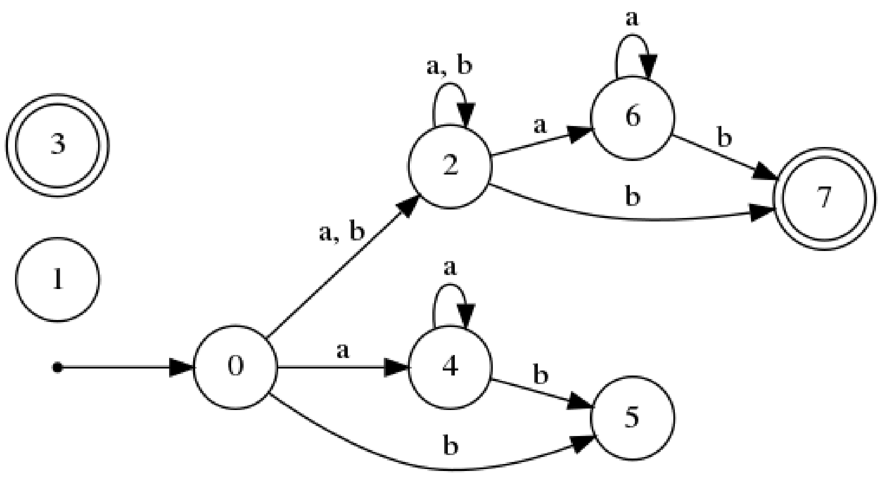


Figure 6: An automaton representing the language $t(L)$.

Finally, we compute the intersection of the two automata, obtaining the result in Figure 7. This is the automaton for which we check for an accepting path. We can clearly see that $ab$ is accepted by this automaton, and thus it's clear $L$ does not satisfy the Suffix Code.

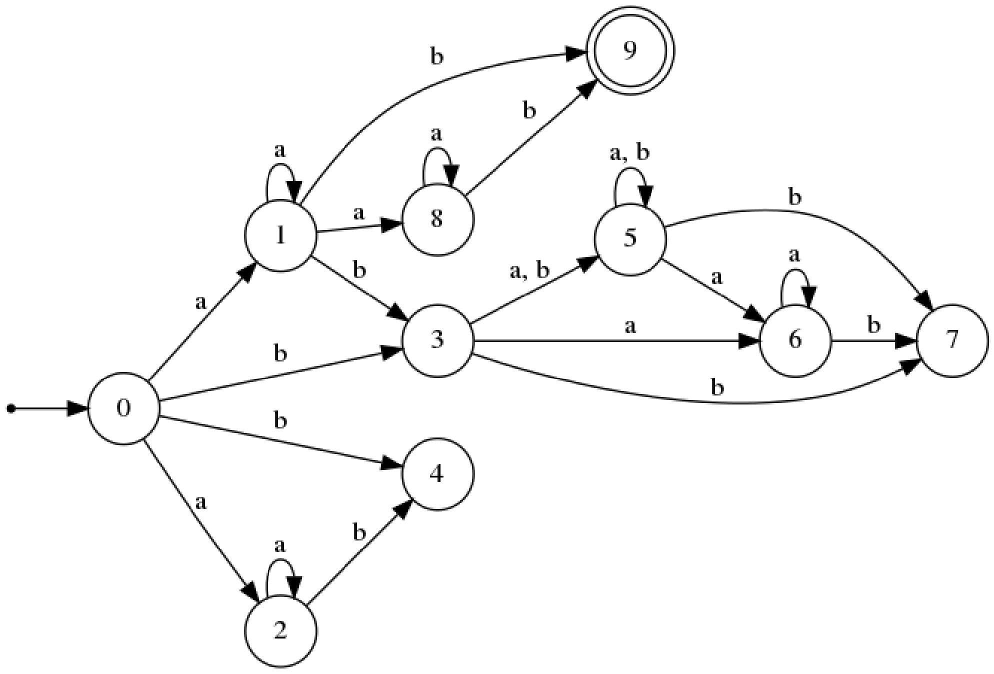


Figure 7: An automaton representing the language $t(L) \cap L$.

Now we define the grammar that specifies the syntax of rational equations. The chosen mathematical notation is Extended Backus-Naur Format, due to similarity with Lark's source files.

```
start: (property | (NEWLINE comment))+ equation

equation: item
    | equation " || " equation
    | equation " & " equation
    | equation " | " equation
    | equation " - " equation

item: VARIABLE
    | method
    | "(" equation ")"
    | "~" item
    | item "*"
    | item "+"

```

```
method: VARIABLE "(" equation ")"

property: VARIABLE " = " (trajectory | transducer | automata | regex) NEWLINE

trajectory: /[01*+ ]+/
regex: /\/.+\//

state: NUMBER
arg: NUMBER | LETTER | "@epsilon"
comment: "#" /.+/

transducer: trans_header trans_transition+
trans_header: "@Transducer" (" " state)+
trans_transition: NEWLINE state " " arg " " arg " " state comment?
    | NEWLINE comment

automata: auto_header auto_transition+
auto_header: auto_type (" " state)+
dfa: "@DFA"
nfa: "@NFA"
auto_type: dfa | nfa
auto_transition: NEWLINE state " " arg " " state comment?
    | NEWLINE comment
```

## 6.1 Explanation of Grammar

```
start: (property | (NEWLINE comment))+ equation
```

This is the top-level entry point of the parser. The parser takes in some number of properties and comments, followed by an equation.

```
equation: item
```

```
    | equation " || " equation
    | equation " & " equation
    | equation " | " equation
    | equation " - " equation
```

This is the equation definition - here, we can see the various operations outlined in the program. These are covered in section Section 7.3. The notation for each operation is defined here - note the requirement for spaces in the equation. This is a limitation of the current iteration of the parser - further improvements could be made here in making these optional, to allow for non-significant whitespace. Additionally, an improvement could be made for implicit concatenation - it is far more organic to write `LL*` than it is to write `L || L*`.

```
item: VARIABLE
    | method
    | "(" equation ")"
    | "~" item
    | item "*"
    | item "+"

method: VARIABLE "(" equation ")"
```

This is where we get into the specifics of parsing individual chunks of the equation.

First, we check to see if the item is a variable, i.e. of the form `a` or `L`, which would prompt our transformer to insert the language represented by the variable. We also check the basic operations, including complementation, kleene star and at least one.

If the item is a method, we proceed to parse the method call. The method call is fairly simple. We retrieve the property/transducer represented by a given variable, and run it on the given equation.

```
property: VARIABLE " = " (trajectory | transducer | automata | regex) NEWLINE
```

This is the rule for declaring and assigning a new variable. As is evident, any transducer, automaton, trajectory or regular expression can be assigned to a variable. Crucially, only transducers and trajectories can be used as a method. A further improvement to the grammar would be to ensure users cannot use automata as a method. Again, note the required spaces, as well.

```
trajectory: /[01*+ ]+/
regex: /\/.+\//
```

Simple rules for regex and trajectories.

```
state: NUMBER
arg: NUMBER | LETTER | "@epsilon"
comment: "#" /.+/
```

Some boilerplate rules used in the rule definitions of automata and transducers.

```
transducer: trans_header trans_transition+
trans_header: "@Transducer" (" " state)+
trans_transition: NEWLINE state " " arg " " arg " " state comment?
    | NEWLINE comment

automata: auto_header auto_transition+
auto_header: auto_type (" " state)+
dfa: "@DFA"
nfa: "@NFA"
auto_type: dfa | nfa
```

```
auto_transition: NEWLINE state " " arg " " state comment?

    | NEWLINE comment
```

This is the high level definition of transducers and automata. We can be pretty hand-wavy in this part of the grammar, as the actual parsing of the structures is handled by FAdo.

# 7 Deciding Rational Equations

As mentioned, previous properties that could be decided by LaSer were limited in scope, and there was a desire to have the ability to represent more complex properties. Examples include 3-independences, 4-independences, etc. An example of such a property is the comma-free code property shown earlier in this paper:

$$\Sigma^+ L \Sigma^+ \cap LL = \emptyset$$

Clearly this is a rational language equation. However it is impossible for us to create a transducer that can represent it, due to this code property being a 4-independence.

The new implementation allows users to input any rational equation. So instead of needing to represent a property via a transducer, users can alternatively input an equation and represent a rational language property.

There are several parts to the input of the new system, including the variables (transducers, languages, trajectories, etc), operations, and finally the equation. The next sections will detail each component.

## 7.1 Equations

Equations are the backbone of the improved system. They allow the user to dictate the operations required to test a given language. Every equation consists of variables and operations, and obey operator precedence:

1. Parentheses
2. Complementation
3. Kleene Star
4. Concatenation

5. Intersection

6. Union

The equation is the final line of the input to the system. We can view the equation like a tree, as outlined in Section 6. We start the the bottom of the three, and run the computation to the top, and then output a decision.

Consider the equation representing the Comma Free Code Property:

$$LL \cap \Sigma^+ L \Sigma^+ = \emptyset$$

In LaSer, this equation is represented by the string

```
(L || L) & t(L)
```

which corresponds to the tree in Figure 8.

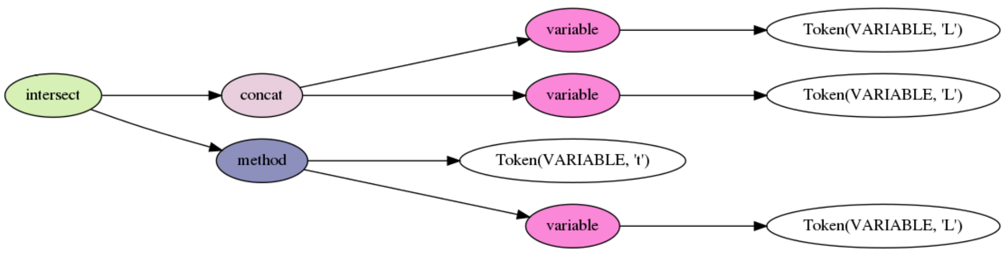


Figure 8: A graphical representation of parse tree Lark generates for the equation $LL \cap \Sigma^+ L \Sigma^+$.

## 7.2 Variables

When creating an equation, the languages and transducers are represented via a single-letter variable. The syntax is:

```
a = [Language, transducer, etc]
b = [2nd Language, transducer, etc]
```

The variable's value can span multiple lines, and a new entry is denoted by a newline:

```
L = @DFA 1
0 a 1
1 b 1
M = @NFA 2
0 a 1
1 b 2
2 c 2
t = 0*1*
z = 1*0*
```

This would be equivalent to

$$L = ab^*, M = abc^*, t = 0^*1^*, z = 1^*0^*$$

where $L$ and $M$ are languages represented by a DFA and NFA, respectively, and $t, z$ are trajectories.

## 7.3 Operations

### 7.3.1 Intersection: &

Equivalent to the $\cap$ operation. Runs `aut1 & aut2` in python, which runs the `__and__` method on the automata. An example of the operation in use:

```
L = /a*b/
t = 1*0*
t(L) & L
```

The above is equivalent to the equation $t(L) \cap L = \emptyset$, as we have been discussing.

### 7.3.2 Union: |

Equivalent to the $\cup$ operation. Runs `aut1 | aut2` in Python, which runs the `__or__` method as implemented in FAdo. An example of use for this operation includes deciding if a language satisfies multiple properties. A a simple example would be a rewrite of the bifix property (Prefix and Suffix Properties combined):

The trajectory $t = 0^*1^* + 1^*0^*$ applied to the equation $t(L) \cap L = \emptyset$ is an expression of the bifix property. However, this is equivalent to the equation $(p(L) \cup s(L)) \cap L = \emptyset$ where $p = 0^*1^*$ and $s = 1^*0^*$. So the following is a representation of the bifix property:

```
L = /ba*b/

p = 0*1*

s = 1*0*

(p(L) | s(L)) & L
```

As shown above, this example could be easily represented in the old implementation as mentioned. However, this concept can allow the decision of any number of properties simultaneously - for example, the Comma-Free Code Property and the 1-substitution property.

### 7.3.3 Set Difference: −

Equivalent to the set difference operator on sets, i.e. $A \setminus B = \{a, aa, aaa\}$. An example of the operation in use:

```
L = /a*b/
```

```
t = 0*1*

S = /(a*b*)*/

t(L) - S
```

The above is equivalent to the equation:

$$t(L) \setminus \Sigma^* = \emptyset$$

This equation checks if $t(L)$ is equivalent to $\Sigma^*$. This relates to maximality, as per the example in Section 7.4.2.

### 7.3.4 Concatenation: $||$

Represented by " $||$ ". Equivalent to set concatenation, i.e.

$$L = \{a, b\}, LL = \{aa, ab, ba, bb\}$$

An example of the operation in use:

```
L = /a*b/

t = 11*0*11*

(L || L) & t(L)
```

The above is equivalent to the equation $LL \cap \Sigma^+ L \Sigma^+ = \emptyset$, the Comma-Free Code Property from previous examples.

## 7.4 Examples

### 7.4.1 Multiple Properties

The following tests the language $L$ for the one-substitution and Comma-Free Code Properties:

```
L = /a*b/
c = 11*0*11*
t = @Transducer 0 1
0 a a 0
0 b b 0
0 a b 1
0 b a 1
1 a a 1
1 b b 1
(t(L) & L) | (c(L) & L || L)
```

A quick look at the automaton this represents in Figure 9 shows us why this is such a powerful new feature.

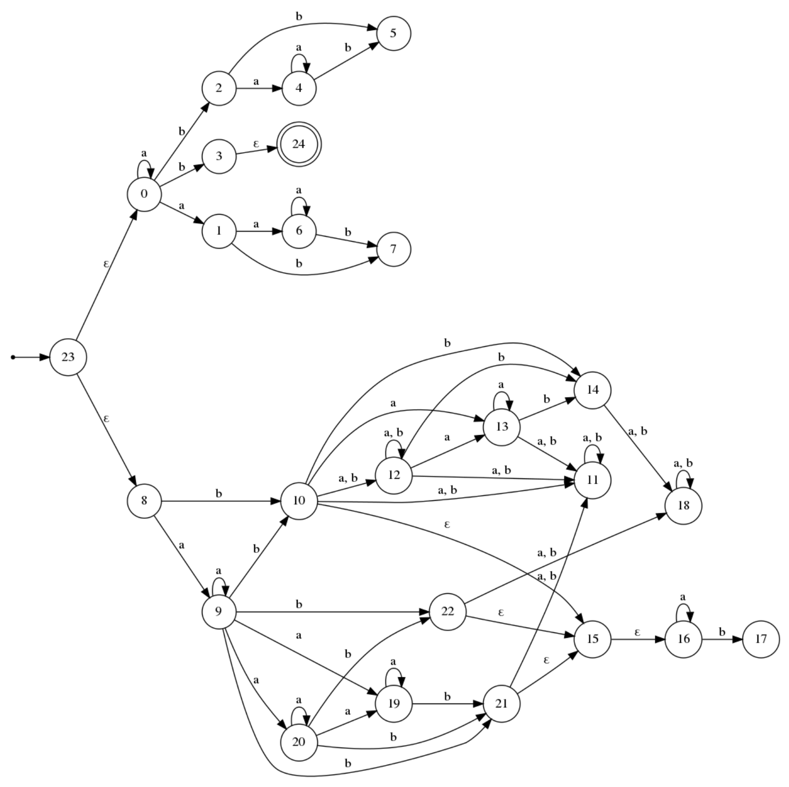


Figure 9: A graphical representation of a complex parse tree

While this question is actually fairly simple to decide without the assistance of a computer, the more complex the equation the more difficult to decide. This system can take many rational equations and quickly decide the answer.

### 7.4.2 Maximality

Consider an independence $P$, and consider some language $L \in P$. We say $L$ is **P-maximal** if $\forall w \in \Sigma^* \setminus L : L \cup \{w\} \notin P$. That is, there is no word we can add to $L$ and still have $L$ satisfy $P$. For $t$-independences, this is shown in [10] to be equivalent to

$$(t(L) \cup t^{-1}(L) \cup L) = \Sigma^*$$

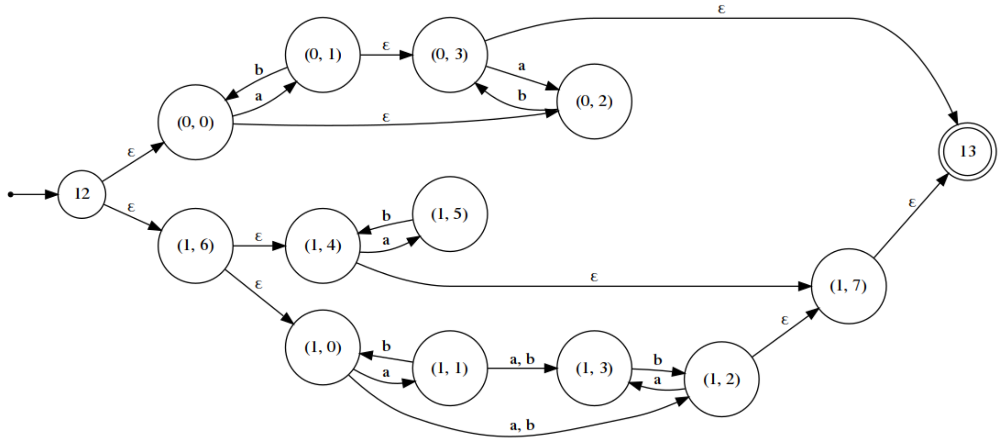
Or, equivalently,

$$(t(L) \cup t^{-1}(L) \cup L)^c = \emptyset$$

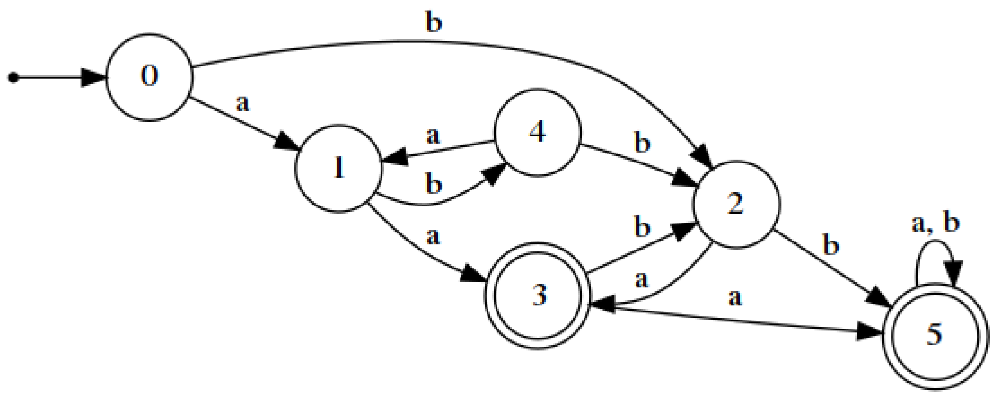
This is, in fact, an equation we can write and decide in LaSer. Consider the following input, where $L = a^*b$ and $t$ is the 1-deletion-detecting property:

```
L = /a*b/

t = @Transducer 1

0 a a 0

0 b b 0

0 a @epsilon 1

0 b @epsilon 1

1 a a 1

1 b b 1

s = @Transducer 1

0 a a 0

0 b b 0

0 @epsilon a 1

0 @epsilon b 1

1 a a 1

1 b b 1

~(t(L) & s(L) & L)
```

Clearly, this is a rational language equation and thus we we can solve it using our system. The automaton prior to negation is as follows:



This is the final automaton:



Clearly, the word $aa$ is in $(t(L) \cap s(L) \cap L)$ and so it is not empty. Thus $L \cup \{aa\} \in P$, and $L$ is not P-maximal.

## 8 Witnesses

Current research shows how to check if a language is t-independent. Additionally, tools from automata theory [6, 20] permit us to give witness words when L is not $t$-independent.

However, all our current approaches are specific to the given condition of $t$-independence. This leads to the question, is it possible to return witness words of non-satisfaction for more complex equations? For example, instead of asking if a language is t-independent, could we ask if the output of two different transducers, ($s$ and $t$) when applied to different languages ($L$ and $M$) share any common words:

$$t(L) \cap s(M) = \emptyset$$

and return a witness if this is the case? What would this witness even look like? Clearly, a word from each language would likely be sufficient. We could then run the words through the transducers and verify ourselves. However, consider this example:

$$\big(t(L) \cap r(M|N)\big) \cap MLN = \emptyset$$

this is far too complex for a two word witness to suffice, and so we can infer witnesses must be significantly more complex for some rational language equations.

Another example of interest is the Comma Free Code Equation:

$$\Sigma^{+} L \Sigma^{+} \cap LL = \emptyset$$

Current research has a specific method for checking if $L$ satisfies the above equations, and for returning a witness if $L$ does not satisfy the Comma-Free Code. However, there is no general approach that works for any given rational language equation. Developing a systematic approach is for further research.

# 9 Conclusions

In this thesis, we outlined the functionality added to LaSer to support the input and processing of rational language equations, allowing the decision of rational language properties in addition to current LaSer functionality. This new functionality allows the representation of properties such as the Comma-Free Code Property, and alternative solutions for other properties, such as the Prefix Code Property.

This thesis included an accessible introduction to the theory behind the functionality, as well as an explanation of the motivation of the thesis. Additionally, in order to better understand these changes, we broke down the grammar and ran though some examples, and provided a list of functionality included in the grammar.

We also outlined some avenues for improvement and further research areas, including witnesses, and areas in which the defined grammar can be improved.

The appendices provide a more detailed breakdown of the specific implementation details of the functionality, in order to make further development easier and explain the rationale of the setup.

## References

[1] J. A. Brzozowski and E. Leiss. On equations for regular languages, finite automata, and sequential networks. *Theoretical Computer Science*, 10(1):19–35, 1980.

[2] M. Domaratzki. Trajectory-based codes. *Acta Informatica*, 40:491–527, 2004.

[3] K. Dudzinski. A system for describing and deciding properties of regular languages using input altering transducers. Master's thesis, Dept. Mathematics and Computing Science, Saint Mary's University, Halifax, NS, Canada, 2011.

[4] K. Dudzinski and S. Konstantinidis. Formal descriptions of code properties: decidability, complexity, implementation. *International Journal of Foundations of Computer Science*, 23:1:67–85, 2012.

[5] FAdo. Tools for formal languages manipulation. URL address:
`http://fado.dcc.fc.up.pt/` Accessed in October of 2017.

[6] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 2001.

[7] J. Jean-Eric Pin. Syntactic semigroups. *Handbook of Language Theory, Vol. I*, pages 679–746, 1997.

[8] H. Jürgensen and S. Konstantinidis. Codes. In Rozenberg and Salomaa [19], pages 511–607.

[9] S. Konstantinidis. Applications of transducers in independent languages, word distances, codes. In G. Pighizzini and C. Câmpeanu, editors, *Proceedings of DCFS 2017*, number 10316 in Lecture Notes in Computer Science, pages 45–62, 2017.

[10] S. Konstantinidis and M. Mastnak. Embedding rationally independent languages into maximal ones. *Journal of Automata, Languages and Combinatorics*, 21(4):311–338, 2017.

[11] S. Konstantinidis, N. Moreira, and R. Reis. Generating error control codes with automata and transducers. In H. Bordihn, R. Freund, B. Nagy, and G. Vaszil, editors, *Proceedings of NCMA 2016*, number 321 in Österreichische Computer Gesellschaft, pages 211–226, 2016.

[12] S. Konstantinidis and P. V. Silva. Maximal error-detecting capabilities of formal languages. *Journal of Automata, Languages and Combinatorics*, 13:55–71, 01 2008.

[13] Lark. Lark - a modern parsing library for python. URL address: `https://github.com/lark-parser/lark` Accessed in February of 2019.

[14] LaSer. Independent LAnguage SERver. URL address: `http://laser.cs.smu.ca/independence/` Accessed in January of 2019.

[15] A. Mateescu and A. Salomaa. Aspects of classical language theory. In Rozenberg and Salomaa [19], pages 175–252.

[16] A. Okhotin. Decision problems for language equations. *Journal of Computer and System Sciences*, 76:251–266, 2010.

[17] R. E. Pattis. Ebnf: A notation to describe syntax, 2013.

[18] G. Rozenberg and A. Salomaa, editors. *Handbook of Formal Languages*. Springer-Verlag, Berlin, 1997.

[19] G. Rozenberg and A. Salomaa, editors. *Handbook of Formal Languages, Vol. I*. Springer-Verlag, Berlin, 1997.

[20] J. Sakarovitch. *Elements of Automata Theory*. Cambridge University Press, Berlin, 2009.

[21] S. Yu. Regular languages. In Rozenberg and Salomaa [19], pages 41–110.

## A Error-Correction and t-Independence

As mentioned, a property is a t-independence if it can be represented by a transducer satisfying the equation

$$\forall w \in L : t(w) \cap (L - \{w\}) = \emptyset$$

However it is not immediately apparent that Error-Correcting Properties satisfy this requirement, and thus are t-indepenences, as they instead satisfy

$$\forall w \in L : t(w) \cap t(L - \{w\}) = \emptyset$$

This second equation seems to be more rigorous than the first, so we would expect it to also satisfy the first requirement. In this appendix we will show that is the case, and that these statements are logically equivalent.

There is a proof relying on the transducer $t$ being an input-preserving transducer that exists [12], but the following proof imposes no requirements on $t$.

**Proposition:** Any error-correcting property represented by a transducer $t$ is a $(t^{-1} \circ t)$ - independence. That is,

$$\forall w \in L : t(w) \cap t(L - \{w\}) = \emptyset \tag{1}$$

$$\iff \forall w \in L : (t^{-1} \circ t)(w) \cap (L - \{w\}) = \emptyset \tag{2}$$

***Proof.***

$\Longrightarrow$ : Suppose toward contradiction (1) holds, and (2) fails. Then

$$\exists w \in L, \exists z \in L - \{w\} \ni z \in t^{-1}(t(w)) \tag{3}$$

and so

$$t(z) \cap t(w) \neq \emptyset \tag{4}$$

and since $t(z) \subseteq t(L - \{w\})$,

$$t(w) \cap t(L - \{w\}) \neq \emptyset \tag{5}$$

which is a contradiction. Thus $\implies$ holds.

$\impliedby$ : Suppose toward contradiction (2) holds, and (1) fails. Then

$$\exists w \in L, \exists z \in t(w) \cap t(L - \{w\}) \tag{6}$$

note that

$$z \in t(w) \iff w \in t^{-1}(z) \tag{7}$$

and

$$\big(\exists w_1 \in L - \{w\} \ni z \in t(w_1)\big) \iff \big(w_1 \in t^{-1}(z)\big) \tag{8}$$

combining (7) and (8), we get

$$w_1 \in t^{-1}(t(w)) \cap (L - \{w\}) \tag{9}$$

which is a contradiction. Thus $\impliedby$ holds, concluding the proof. ■

# B Lark Source Code

The grammar used in LaSer is written for Lark. The grammar is essentially a preprocessor, and passes much of the more complex parsing on to FAdo and yappy parser, which complete the parsing. Lark files are written in Extended Backus-Naur form [17], which should be comfortably understood by those who have written compilers.

The source is as follows:

```
%import common.NUMBER
%import common.LETTER
%import common.NEWLINE
%import common.WS_INLINE
%import common.WS
%import common.CNAME -> VARIABLE

start: (property | (NEWLINE comment))+ equation

?equation: item
    | equation " || " equation -> concat
    | equation " & " equation -> intersect
    | equation " | " equation -> union
    | equation " - " equation -> disjoint

?item: VARIABLE -> variable
    | method
    | "(" equation ")"
    | "~" item -> negate
    | item "*" -> kleene
    | item "+" -> at_least

method: VARIABLE "(" equation ")"
    | VARIABLE "!(" equation ")" -> inout

property: VARIABLE " = " (trajectory | transducer | automata | regex) NEWLINE -> assign

```

```
// Trajectory definition
trajectory: /[01*+ ]+/
regex: /\/.+\//

state: NUMBER
arg: NUMBER | LETTER | "@epsilon"
comment: "#" /.+/

// Transducer definition
transducer: trans_header trans_transition+
trans_header: "@Transducer" (" " state)+
trans_transition: NEWLINE state " " arg " " arg " " state comment?
    | NEWLINE comment

// Automata definition
automata: auto_header auto_transition+
auto_header: auto_type (" " state)+
dfa: "@DFA"
nfa: "@NFA"
auto_type: dfa | nfa
auto_transition: NEWLINE state " " arg " " state comment?
    | NEWLINE comment
```

In order to better understand the functionality of this parser, we will look at each segment of the code, and how it relates to the system overall.

```
%import common.NUMBER
%import common.LETTER
%import common.NEWLINE
%import common.WS_INLINE
%import common.WS
%import common.CNAME -> VARIABLE
```

Various built in functionality of Lark [13]. Importing some of the defaults allow skipping of simple declarations.

```
start: (property | (NEWLINE comment))+ equation
```

This is the top-level entry point of the parser. The parser takes in some number of properties and comments, followed by an equation.

```
?equation: item

    | equation " || " equation -> concat

    | equation " & " equation -> intersect

    | equation " | " equation -> union

    | equation " - " equation -> disjoint
```

This is the equation definition - here, we can see the various operations outlined in the program. These are covered in section Section 7.2.3. The notation for each operation is defined here - note the requirement for spaces in the equation. This is a limitation of the current iteration of the parser - further improvements could be made here in making these optional, to allow for non-significant whitespace. Additionally, an improvement could be made for implicit concatenation - it is far more organic to write `LL*` than it is to write `L || L*`.

The question-mark preceding the identifier (`?equation`) instructs Lark's tree-builder to inline this branch if it has only one member. That is, if the equation is a single item, the processing for `equation` will be skipped.

The `-> [identifier]` signifies an alias. An alias tells lark to assign the python method of name `identifier` for processing a specific part of the rule.

```
?item: VARIABLE -> variable

    | method

    | "(" equation ")"
    | "~" item -> negate
    | item "*" -> kleene
    | item "+" -> at_least
```

This is where we get into the specifics of parsing individual chunks of the equation.

First, we check to see if the item is a variable, i.e. of the form `a` or `L`, which would prompt our transformer to insert the language represented by the variable. We also check the basic operations, including negation/inverse, kleene star and at least one.

If the item is a method, we proceed to parse the method call.

```
method: VARIABLE "(" equation ")"
```

The method call is fairly simple. We retrieve the property/transducer represented by a given variable, and run it on the given equation.

```
property: VARIABLE " = " (trajectory | transducer | automata | regex) NEWLINE -> assign
```

This is the rule for declaring and assigning a new variable. As is evident, any transducer, automaton, trajectory or regular expression can be assigned to a variable. Crucially, only transducers and trajectories can be used as a method. A further improvement to the grammar would be to ensure users cannot use automata as a method. Again, note the required spaces, as well.

```
// Trajectory definition
trajectory: /[01*+ ]+/
regex: /\/.+\//
```

Simple rules for regex and trajectories.

```
state: NUMBER

arg: NUMBER | LETTER | "@epsilon"

comment: "#" /.+/
```

Some boilerplate rules used in the rule definitions of automata and transducers.

```
// Transducer definition

transducer: trans_header trans_transition+

trans_header: "@Transducer" (" " state)+

trans_transition: NEWLINE state " " arg " " arg " " state comment?

    | NEWLINE comment


// Automata definition

automata: auto_header auto_transition+

auto_header: auto_type (" " state)+

dfa: "@DFA"

nfa: "@NFA"

auto_type: dfa | nfa

auto_transition: NEWLINE state " " arg " " state comment?

    | NEWLINE comment
```

As mentioned, this grammar simply passes the parsing of all more complex structures to FAdo, so the python code handling this grammar will simply concatenate all the sections of the transducer together and pass it into FAdo. While in the long term, integrating the parsing of transducers and automata into this file would be optimal, the fact FAdo currently relies on yappy parser means it would require a complete rewrite of FAdo's parsing system.

# C Excerpts of Python Source Code

In this section we will look over some choice sections of the parser's python code, the file `parser.py`. Lark interfaces with python via a derivative of the `Transformer` class. Each method of the class corresponds to a rule or alias given in the Lark Grammar.

```python
class ParseEquation(Transformer):
    def __init__(self):
        self.vars = dict()
        self.sigma = set()

    def automata(self, vars):
        aut = readOneFromString(''.join(vars) + '\n')

        self.sigma = self.sigma.union(aut.Sigma)

        return aut

    def auto_header(self, vars):
        return str(' '.join(vars))

    dfa = lambda self, _: "@DFA"
    nfa = lambda self, _: "@NFA"
    auto_type = lambda self, vars: vars[0]
    state = lambda self, vars: str(vars[0])

    def arg(self, vars):
        val = "@epsilon"

        try:
            val = str(vars[0])
        except:
            pass

        return val

    def auto_transition(self, vars):
        return str('\n' + ' '.join(vars[1:]))

    def assign(self, vars):
        key = str(vars[0])
        self.vars[key] = vars[1]

    def variable(self, vars):
        return self.vars.get(str(vars[0]), '')

    def trajectory(self, vars):
        return buildTrajPropS(str(vars[0]) + '\n', self.sigma).Aut

    def kleene(self, vars):
        return vars[0].star()

    def at_least(self, vars):
        return vars[0].concat(vars[0].star())

    def concat(self, vars):
        return vars[0].concat(vars[1])

    def method(self, vars):
        trans = self.vars.get(str(vars[0]))

        aut = vars[1]

        result = trans.runOnNFA(aut)

        return result

    def inout(self, vars):
        trans = self.vars.get(str(vars[0]))

        aut = vars[1]

        result = trans.inIntersection(aut).outIntersection(aut).toOutNFA()

        return result

    def intersect(self, vars):
        aut = vars[0] & vars[1]

        return aut

    def union(self, vars):
        aut = vars[0] | vars[1]

        return aut

    def start(self, vars):
        aut = vars[-1:][0]

        return aut.witness()

    def comment(self, vars):
        return ''

    def regex(self, vars):
        regex = reex.str2regexp(str(vars[0])[1:-1])

        aut = regex.toNFA()

        self.sigma = self.sigma.union(aut.Sigma)

        return aut

    def transducer(self, vars):
        trans = readOneFromString(''.join(vars) + '\n')

        self.sigma = self.sigma.union(trans.Sigma)

        return trans

    def trans_header(self, vars):
        return "@Transducer " + str(' '.join(vars))

    def trans_transition(self, vars):
        return str('\n' + ' '.join(vars[1:]))

    def negate(self, vars):
        return ~vars[0]

    def disjoint(self, vars):
        return vars[0] & (~vars[1])
```

We will break down the implementation by each method.

```
class ParseEquation(Transformer):
    def __init__(self):
        self.vars = dict()
        self.sigma = set()
```

Initialization of the dictionary containing our variables, and sigma which will contain our overall alphabet. The variables are stored by the name given to them in the file. For example, `L = /a*b/` will perform `self.vars['L'] = /a*b/`. The sigma is necessary, as if we use multiple languages of different alphabets (i.e. $a * b$ and $c * d*$), we must ensure we use the same sigma overall.

```
    def automata(self, vars):
        aut = readOneFromString(''.join(vars) + '\n')

        self.sigma = self.sigma.union(aut.Sigma)

        return aut
```

```
def auto_header(self, vars):
    return str(' '.join(vars))


dfa = lambda self, _: "@DFA"
nfa = lambda self, _: "@NFA"
auto_type = lambda self, vars: vars[0]
state = lambda self, vars: str(vars[0])
```

This is the parsing for automata. As we can see, the parser does not, in fact, parse the automata itself, instead simply passing the rebuilt string into FAdo, specifically the `readOneFromString` [5] function.

```
def arg(self, vars):
    val = "@epsilon"

    try:
        val = str(vars[0])
    except:
        pass

    return val


def auto_transition(self, vars):
    return str('\n' + ' '.join(vars[1:]))
```

This part of the code deals with the argument of the transition - that is, for a given transition $(q_1, \sigma, q_2), q_1, q_2 \in Q, \sigma \in \Sigma$, the letter $\sigma$. The possible arguments are either `@epsilon` or an element of the alphabet of the automaton.

```
def assign(self, vars):
    key = str(vars[0])
    self.vars[key] = vars[1]
```

```
def variable(self, vars):
    return self.vars.get(str(vars[0]), '')
```

Here we deal with the assignment and retreival of automata, transducers, etc. to their respective variable name. `vars[0]` is of type string, and `vars[1]` is the type of structure (?) being assigned (DFA, NFA, Trajectory, etc.). This will allow us to recall them when needed by that same name.

```
def trajectory(self, vars):
    return buildTrajPropS(str(vars[0]) + '\n', self.sigma).Aut
```

This passes through the parsing of trajectories as discussed in Section 2.5, using the `buildTrajPropS` [5] function from FAdo.

```
def kleene(self, vars):
    return vars[0].star()

def at_least(self, vars):
    return vars[0].concat(vars[0].star())

def concat(self, vars):
    return vars[0].concat(vars[1])
```

Various operations on languages. Uses `FA.star()` and `FA.concat()` [5] from FAdo. It is important to note that both variables will be of type `NFA` or of type `DFA` - this is a prevailing theme throughout the implementation.

```
def method(self, vars):
    trans = self.vars.get(str(vars[0]))

    aut = vars[1]

    result = trans.runOnNFA(aut)

    return result
```

This outlines the running of a transducer on an NFA. It is important to note that `SFT.runOnNFA()` returns an NFA, not a transducer.

```
def intersect(self, vars):
    aut = vars[0] & vars[1]

    return aut

def union(self, vars):
    aut = vars[0] | vars[1]

    return aut
```

Here we see the use of the built in `__and__` and `__or__` operations of Python. FAdo implements these methods to allow concise operations on languages. This is also the origin of the identifiers used in the grammar for intersection and union.

```
def start(self, vars):
        aut = vars[-1:][0]

        return aut.witness()
```

This is, counterintuitively, the end of the computation. We are now back at the top of the tree, so we call `FA.witness()` [5] to check if there is still any valid path through the automaton. If there is, we know that the language is not, in fact, empty, and the decision is that the equation is not satisfied. If, on the other hand, the NFA does not contain a valid path, the equation is satisfied.

```
    def comment(self, vars):
        return ''
```

This just ignores any comments in the input.

```
    def regex(self, vars):
        regex = reex.str2regexp(str(vars[0])[1:-1])

        aut = regex.toNFA()

        self.sigma = self.sigma.union(aut.Sigma)

        return aut
```

This is the parsing of the regex, once again passed to FAdo via the `str2regexp` [5] function. Additionally, we convert the regular expression into an NFA.

```
    def transducer(self, vars):
        trans = readOneFromString(''.join(vars) + '\n')
        self.sigma = self.sigma.union(trans.Sigma)

        return trans

    def trans_header(self, vars):
        return "@Transducer " + str(' '.join(vars))

    def trans_transition(self, vars):
        return str('\n' + ' '.join(vars[1:]))
```

This is the parsing for transducers. As we can see, the parser does not, in fact, parse the transducer itself, instead simply passing the rebuilt string into FAdo, specifically the `readOneFromString` [5] function.

This is functionally very similar to how automata are handled.

```
    def negate(self, vars):
        return ~vars[0]
```

This returns the complement of an automata, i.e. $\Sigma^* \setminus L$. The negation is handled in FAdo [5]. This is can be an extremely computationally complex operation.

```
    def disjoint(self, vars):
        return vars[0] & (~vars[1])
```

This is equivalent to the operation $L \setminus M$. Note the use of the complement.