I-LaSer GUI Instruction Manual

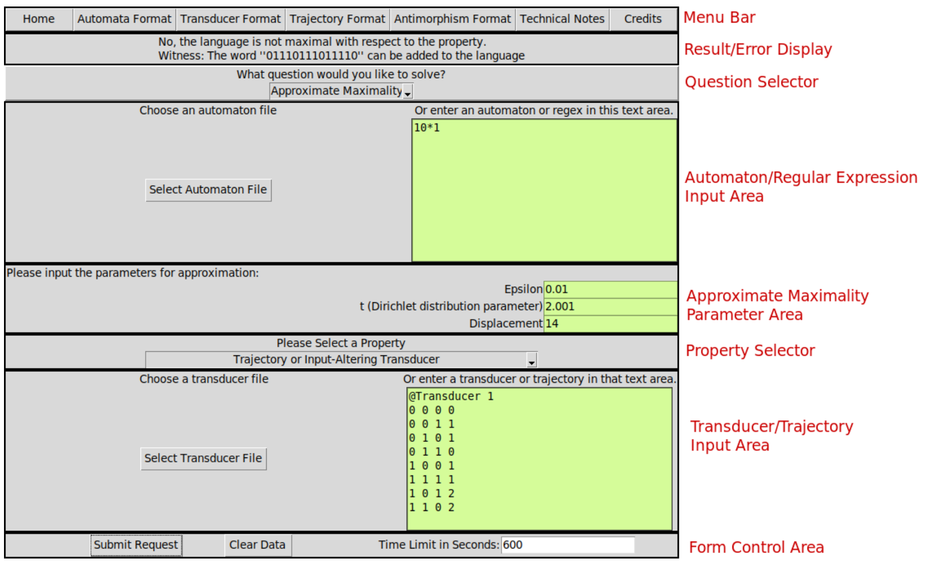
Baxter Madore

August 16, 2024

Contents

1 Introduction

The Independent Language Server (I-LaSer) is a program, originally built in June 2010 and currently on its sixth feature release, which answers questions relating to the satisfaction and maximality of formal languages [5]. There is also an option to construct a language of a given size using certain parameters. It accepts a description of a regular language, which can be provided by either a regular expression or a finite automaton, and a description of a language property, which can be either a fixed type, provided by a trajectory, or provided by a transducer [3]. I-LaSer uses the formal languages library FAdo[1], created by Rogério Reis and Nelma Moreira, to perform all of its backend calculations.



No, the language is not maximal with respect to the property.
Witness: The word ''01110111011110'' can be added to the language

What question would you like to solve? Approximate Maximality

Or enter an automaton or regex in this text area.

```
10*1
```

Please input the parameters for approximation:

Epsilon 0.01
t (Dirichlet distribution parameter) 2.001
Displacement 14

Please Select a Property: Trajectory or Input-Altering Transducer

Or enter a transducer or trajectory in that text area.

```
@Transducer 1
0 0 0 0
0 0 1 1
0 1 0 1
0 1 1 0
1 0 0 1
1 1 1 1
1 0 1 2
1 1 0 2
```

Time Limit in Seconds: 600

Figure 1: Example usage of I-LaSer GUI (Ubuntu)

2 Motivation for Creating the Local Version

The I-LaSer web version has some drawbacks, and the creation of a locally-run GUI aims to fix these issues. The main issue that the web version of I-LaSer faced was that the behind-the-scenes calculations took a lot of time and computing power. For more complex languages and/or properties, this caused too much strain on the server. Another issue is that the web server faced cyberattacks, resulting in its temporary removal. Having a locally run version protects against any future server downtime, as a user will have their own program available to run without depending on the server, and provides the ability for users to bypass server limits on language complexity and calculation time. A smaller benefit is that the local GUI can be run without Internet access.

2.1 Problems with Program Generation

The web version's drawbacks are partially alleviated by allowing the user to generate a Python program, which, when run, produces the correct answer on the user's machine without forcing the web server to run calculations. However, this is not a perfect solution for a few reasons. Running the program requires the user have a working Python installation, and take the extra steps of unzipping the folder, running the Python program, then deleting the zipped folder and the extracted folder. Due to the newlines in the text areas not translating well to generated programs, the generated program must also encode input strings in base-64 and decode then during program execution. For these reasons, program generation has been intentionally left out of the I-LaSer GUI. Furthermore, if there are future issues resulting in server downtime, this would also prevent access to generating programs.

3 Time Limit

All submissions will disable further access to the form controls until the calculation either completes or times out. This will lock all buttons and textboxes. Setting a good time limit will help the function time out as quickly as necessary and return control. By default, the time limit is 60 seconds for the calculation. This limit can be lowered in the case of a simple query which should not take very long, raised for complicated queries intended to run for a long time, or eliminated entirely. To eliminate the time limit, enter a non-positive number or a non-number in the Time Limit field. Non-integer numbers of seconds are permitted to be used as the time limit.

4 Independences

4.1 Satisfaction

Roughly speaking, a language is said to *satisfy* an independence property if and only if there does not exist a pair of words w_1 and w_2 such that w_2 can be created by performing operation T on w_1. The operation T is defined by the property. If a language satisfies a property, every subset of that language satisfies the same property. An empty language satisfies all independence properties, as there are no words to transform into other words in a language. For similar reasons, a language with only a single word (which may be the empty word) also satisfies all independence properties.

4.2 Maximality

A language L satisfying property P is said to be *maximal* with respect to P if and only if no more words can be added to L without losing its satisfaction of P. When testing maximality and approximate maximality using I-LaSer, you will get an error if the language does not satisfy the property, as the (approximate) maximality question is meaningless on such languages, since the language has already lost satisfaction of the property without adding any words.

4.3 Approximate Maximality

Testing code maximality is PSPACE-hard, and in the worst case takes exponential time [4]. The Approximate Maximality feature of I-LaSer 6 alleviates this problem by not testing for complete maximality, but instead, generates a sample of words over the alphabet, testing them to see if they can be added to the language. If the sample includes a word that can be added to the language, then the language is definitely not maximal and I-LaSer will provide the word that can be added to the language. If the sample does not include a word that can be added to the language, I-LaSer will conclude that the language is approximately maximal. Such a statement does not definitively mean that the language is truly maximal with respect to the property, but it can say that the language is close to being maximal. Setting up the random distribution of words takes time, so although calculating approximate maximality is in theory faster than true maximality, in practice it is only faster for complex languages and properties.

I-LaSer uses a polynomial randomized approximation (PRAX) algorithm and the Dirichlet distribution over the positive integers to generate the samples. Due to the randomness involved, questions of approximate maximality may not result in the same output from I-LaSer when run multiple times. For more information on the theory behind approximate maximality, please read [4].

5 Fixed Properties

This section contains simplified explanations of all fixed properties available to select in I-LaSer

5.1 Prefix

A language is a *prefix code* if no word in the language is a prefix of another word in the language. If one word in the language can be formed by removing symbols from the end of another word in that language, then the language is not a prefix code.

An example of a language that is a prefix code is `aa*b`, the language with all words containing one or more a's followed by a single b. This is a prefix code since removing the last symbol of any word will remove the b, and make the result a word that is not in the language. This language, however, is not a *maximal* prefix code, since we could add the word "b", and the language would still be a prefix code.

A language that is **not** a prefix code is `a*b*`, zero or more a's followed by zero or more b's. The empty string, with zero a's and zero b's, is in the language, and if you keep 0 symbols from the start of a different word, and discard the rest, the result will be the empty string. In general, if a language includes the empty string and at least one non-empty word, it will not satisfy any of the fixed code properties shown by I-LaSer.

5.2 Suffix

A language is a *suffix code* if no word in the language is a suffix of another word in the language. It is very similar to the prefix code. Discard zero or more symbols from the start, and keep the rest, if you end up with another word in the language, then the language is **not** a suffix code.

One language that is not a suffix code is the language `(b+bb)aa*`, consisting of words with one or two b's, then one or more a's. This is not a suffix code because you can remove the first b from any word with two b's, and createa another word in the language.

5.3 Bifix

A language is a *bifix code* if and only if it is both a prefix code and a suffix code.

5.4 Infix

A language is an *infix code* if no word in the language can be found inside another word in the language, whether at either end or somewhere in the middle. It is necessary for a language to be a bifix code in order to be an infix code, but this is not a sufficient condition.

One language that is a bifix code but not an infix code is the language `(ab*a)+(bbb)`, which consists of all words containing one a at the beginning and end of the word, and any number of b's in between, and the word "bbb". When looking at the word "abbba", it's a prefix code because you cannot remove only letters from the start of the word to get 'bbb", and you cannot remove only letters from the end of the word to get "bbb", but removing letters from both ends of the word at once can result in the word "bbb" and disqualify the language from being an infix code.

5.5 Outfix

A language is an *outfix code* if no word in the language can be formed by removing a single contiguous block of letters from any other word in the language. This piece may be at the start of the word, the end of the word, or anywhere in between. All outfix codes are also bifix codes.

A language that is not an outfix code is `ab*a`, since you can remove the block of b's, however long it may be, and get "aa", which is in the language.

5.6 UD Code

A language is not a *uniquely decodable code* if there exists a string of characters, made up of multiple words concatenated together, has multiple places to put the word breaks, that is, some ambiguity in "un-concatenating" the words. All prefix, suffix, infix, outfix, and bifix codes are uniquely decodable codes. One example of a language that isn't is `a(aa)*`, all words with an odd number of a's, since, when given a string of 5 a's, it is impossible to tell whether it is a single word, or two "a"s and one 'aaa" in some order.

5.7 Hypercode

A language is a hypercode if it is not possible to remove any symbols from one word in the language to get another word in the language. The removed symbols need not be contiguous, and there is no restriction on the position of the removed symbols. A language satisfying the hypercode property is sufficient to conclude that the same language also satisfies the prefix, suffix, bifix, infix, outfix, and UD code properties. One such language is the language of all four-letter words over a given alphabet, which can be written as `(0+1)^4` or `(0+1)(0+1)(0+1)(0+1)`. Removing any letter from a word makes it a shorter word, so it wouldn't be in the language.

6 Input Areas

6.1 File Input

I-LaSer can accept typed input in the large text boxes, for automata, transducers, and theta properties, or can accept file input. Clicking the "Upload File" button will open a file selector, from which you may select a text file. Selecting a text file will replace all input in the text box with the contents of the file. Selecting a non-text file or an unreadable file will result in the input area turning red and displaying an error. Cancelling file selection will preserve the data in the text area.

6.2 Validation

I-LaSer will attempt to perform validation on the inputs to the text box, no matter if the inputs originated from the keyboard or from a file. If the input passes validation, the text box will turn green. If the input fails validation, the text box will remain white. It is still possible to submit inputs that the validator deems "invalid" and have them run without errors. Similarly, the green text box does not necessarily mean that no errors will occur. The validation is a guide, indicating which inputs are likely to result in an unsuccessful computation.

For the construction question, the user is prompted to enter three integers, each of which are validated seperately. Similarly, all of the parameters for approximate maximality are independent of each other, in terms of validation, so there may be some green areas and some white areas. For these two input areas, if they don't pass validation (with all three input areas green), the calculation will not succeed and there will be an error.

6.3 Acceptable Characters

When entering an automaton, transducer, regular expression, or theta property, the alphabets are limited to the digits 0-9 and the Latin letters A-Z. I-LaSer alphabets are case-sensitive, so A is a different symbol than a. In most cases, the string "@epsilon" can be used to indicate the empty word. This is useful when entering regular expressions and allowing a variable quantity of a certain symbol, or in a transducer to either insert an output symbol without any input, or to remove an input symbol without creating output.

6.4 Automaton Area

See the "automata format" tab of I-LaSer for the basics of what to enter in the automaton area or an automaton file. The set of symbols in the state transitions is considered the alphabet of the language, unless an alphabet is otherwise specified. If there is no defined state transition on a symbol from the current state, that symbol cannot appear when the automaton is in the current state.

The first line of an automaton is the declaration line. The declaration line consists of the following fields, separated by spaces, where anything enclosed in square brackets is optional.
Automaton Type (@NFA or @DFA), Final States, [* Initial States], [$Alphabet]. The below example shows a delcaration line for an NFA which can start at states 0 or 5, can end at states 7 and 12, and uses the alphabet a, b, c, 0, 1.

If the automaton uses symbols outside the defined alphabet, they are implicitly added.

```
@NFA 7 12 * 0 5 $abc01
```

If a language has an explicitly defined start state, and the language does not have any state transitions coming from that state, the language is empty. Similarly, if there is no series of state transitions from a start state to any defined end state, the language will also be empty. The list of initial states is only permitted when entering an NFA. A DFA must only have one start state, so the start state is defined as the initial state of the first transition and listing the start state is unnecessary.

When entering a finite automaton, there are two options. @DFA begins a Deterministic Finite Automaton, and comes with more restrictions. DFAs may not have any state transitions caused by @epsilon, and may not have a symbol causing more than one state transition. The following list of state transitions would be legal as an NFA but illegal as a DFA, because there are multiple paths from state 1 when symbol a is encountered:

```
1 a 2
1 b 3
1 a 3
```

For a more detailed description of the grammar behind FAdo automata parsing, read the FAdo instruction manual [8].

6.5 Grail Automata

Along with the preferred FAdo syntax, automata in the Grail[7] format are also supported. The state transition lines are written identically to FAdo's format, with the source state, the symbol, and the state to advance to. The differences between the two formats are in defining the start and end states. In Grail, the start state is defined with the string `(START) |- N`, where N is a start state. Final states are defined with the string `F -| (FINAL)`, where F is a final state. For languages with more than one start state and/or more than one final state, these lines may be repeated as many times as necessary. Start and end state lines may be declared anywhere before, in between, or after the state transitions lines. It is not possible to define an alphabet in Grail syntax with symbols that do not cause a state transition, FAdo format must be directly used in that case.

Grail-formatted automata are implicitly converted to FAdo format before being processed and are always treated as NFAs.

6.6 Regular Expressions

Regular expressions are a simpler way to express regular languages, and can be entered in the language input area to describe a language. When entering a mathematical (Kleene) regular expression, the only operators allowed are:

- xy - the concatenation of symbols x and y
- x* - 0 or more of symbol "x" (Kleene star)
- x+y - either symbol "x" or symbol "y"
- (x) - treat x as one symbol
- `x^n` - repeat symbol x n times. It is recommended but not required to enclose the entire expression `(x^n)` in parentheses. n must not have seperate parentheses and must be a single number (e.g. The expression `a^(2)` is not permitted). This operator has the highest precedence.

Some quantifiers used in computer programming regular expressions that can be expressed in mathematical regular expressions are shown in the below table for convenience.

Programming RegEx	⇒	Mathematical Regular Expression
`x?`	⇒	`(x+@epsilon)`
`x+`	⇒	`xx*`
`x{3,}`	⇒	`x^3x*`
`x{3,6}`	⇒	`x^3(x|@epsilon)^3`
`x{,6}`	⇒	`(x+@epsilon)^6`

6.7 Transducer Area

See the "transducer format" tab of I-LaSer for the basics of what to include in the transducer area or the transducer file. A transducer can be thought of as a non-deterministic automaton, with an output attached to each state transition.

Similarly to the automaton format, the transducer input starts with the word @Transducer, a list of final states, and optionally, a list of starting states and an alphabet. When entering a transducer, the (input) alphabet of the transducer must cover all symbols in the language, or else I-LaSer will throw an error.

A language is said to satisfy a transducer property if and only if no word in the language, when inputted to the transducer, outputs another word in the language. For example, the below transducer, when given the word "001", outputs "001", "101", "011", and "111", so a language including "001" and "111" would not satisfy the property defined by that transducer.

```
@Transducer 0
  0 0 0 0
  0 0 1 0
  0 1 1 0
  0 1 0 1
```

The above transducer is also *input-preserving*, because any input may produce itself when run through the transducer. A transducer which is not input-preserving is said to be *input-altering*. An input-alterng transducer is one in which every word output from the transducer is different from its input, for all input words.

6.8 Trajectories

It is also possible to input a trajectory in the area for transducers, if the property type "Trajectory or Input-Altering Transducer" is selected. Trajectories have the same operator set as regular expressions, though their alphabet is restricted to $\{0, 1\}$. Trajectories offer more property options than fixed types, though considerably less than transducers. For instance, a language satisfies the trajectory property (01)* if and only if it is impossible to make one word in the language using all of the symbols in the odd-numbered positions of another word. For more detail on trajectory expressions, see [2].

6.9 Trajectory Equivalents for Fixed Types

Prefix	0*1*
Suffix	1*0*
Bifix	1*0* + 0*1*
Infix	1*0*1*
Outfix	0*1*0*
Hypercode	(0+1)*

6.10 Theta Properties

See the Antimorphism tab in the I-LaSer GUI for basic information about theta properties. Theta properties are used for DNA computing, and the only question that I-LaSer currently supports using theta-properties for is the Satisfaction question. Antimorphic "swaps" are specified in the theta text area, and may include any symbols legal in languages.

Theta properties start with the string "@THETA", followed by a newline, and a list of swaps to be made. Satisfaction for theta-properties is handled quite differently than satisfaction for non-antimorphic properties. Before testing to see whether a word operation leads to another word in the language, the word first needs to be permuted. The permutation of the word involves reversing its symbols, then applying each of the specified swaps.

For example, on the DNA alphabet with swaps a, t and c, g, the word "gcct" would be reversed to 'tccg", then the swaps would be performed, turning the original word into "aggc". That would then be compared to the other words in the language via the operation for the property.

7 Language and Property Examples

7.1 Automaton Examples

Automaton accepting aaa(aaa)*b + aa(ba)*a(aa(ba)*a)*b:

FAdo:
```
@NFA 8
1 a 2
2 a 3
3 b 2
3 a 4
4 a 2
4 b 8
1 a 5
5 a 6
6 a 7
7 a 5
7 b 8
```

Grail:
```
(START) |- 1
1 a 2
2 a 3
3 b 2
3 a 4
4 a 2
4 b 8
1 a 5
5 a 6
6 a 7
7 a 5
7 b 8
8 -| (FINAL)
```

Automaton accepting all words with an odd number of A's and an odd number of B's

FAdo:
```
@NFA 3
0 A 1
0 B 2
1 A 0
1 B 3
2 A 3
2 B 0
3 B 1
3 A 2
```

Grail:
```
(START) |- 0
3 -| (FINAL)
0 A 1
0 B 2
1 A 0
1 B 3
2 A 3
2 B 0
3 B 1
3 A 2
```

Deterministic automaton accepting all 3-symbol words over the alphabet $\{a, b\}$ with an even number of b's:

FAdo:	Grail:
@DFA 5	(START) |- 0
0 a 2	0 a 2
0 b 1	0 b 1
1 a 3	1 a 3
1 b 4	1 b 4
2 a 4	2 a 4
2 b 3	2 b 3
3 b 5	3 b 5
4 a 5	4 a 5
	5 -| (FINAL)

7.2 Transducer Examples

Input-altering transducer describing the *thin* property, where all words must be of different lengths, over the alphabet $\{8, 9\}$. This transducer will output every word that is not the input, but is the same length as the input.

```
@Transducer 2
1 8 8 1
1 9 9 1
1 8 9 2
1 9 8 2
2 8 8 2
2 9 9 2
2 8 9 2
2 9 8 2
```

Input-preserving transducer describing the *1-substitution-detecting* property over the alphabet $\{a, b, c\}$. When word w is input, the transducer will output w and any word created by changing one symbol in w. To make this an input-altering transducer which does not output w and only words formed by substituting a symbol, remove the 0 from the list of final states.

```
@Transducer 0 1
0 a a 0
0 b b 0
0 c c 0
0 a b 1
0 a c 1
0 b a 1
0 b c 1
0 c a 1
0 c b 1
```

```
1 a a 1
1 b b 1
1 c c 1
```

Input-preserving transducer which outputs all words formed by transposing two adjacent bits of its input word:

```
@Transducer 4 * 1
1 0 0 1
1 1 1 1
1 0 @epsilon 2
1 1 @epsilon 3
2 1 1 5
5 @epsilon 0 4
3 0 0 6
6 @epsilon 1 4
4 0 0 4
4 1 1 4
```

8 Approximate Maximality Parameters

- Epsilon - A higher epsilon value correlates to more error tolerance, and fewer words being sampled over the distribution. Epsilon must be greater than 0 and less than 1.
- t - The Dirichlet distribution that the approximate maximality calculations use requires a parameter t. This is used in the creation of the word sample space and correlates to the uniformity of the distribution of word lengths across the sample space. A smaller value of t means that the sample will be more uniform in terms of word length, but will also take much longer to set up. t must be greater than 1.
- Displacement - No words will be included in the sample if they have fewer than d characters. This is useful when a language has a few uninteresting short words which could be added to the language. Displacement must be a non-negative integer.

9 Construction

The construction question allows I-LaSer to accept a language property and build a block language of a given size which satisfies that property [6]. A block language is one where the words in the language are of a uniform length.

9.1 Parameters

- S - The number of symbols in the construction alphabet. There is currently no way to set the construction alphabet to anything other than the digits 0 to $S-1$, and since there are only 10 digits, construction alphabet size has a strict maximum of 10. It is not possible to create more than one word of a given length over the unary alphabet, so construction alphabet size has a strict minimum of 2. Similarly to testing questions about satisfaction and maximality, the construction alphabet must be a subset of the transducer's alphabet, and should ideally be the same size, that is, for a property over the alphabet 0, 1, 2, the parameter S must be at most 3, and ideally would be exactly 3.

- N - The construction query will attempt to generate N unique words over the alphabet. This may not be possible due to the property inputted, or the length of the strings and the alphabets entered. For example, if $N > L^S$, it will be impossible to create N unique strings of length L, and I-LaSer will stop when it cannot add any more words to the language.
- L - The length of the words to be constructed. All words constructed will be the same length.

The output of the construction question is arbitrary, and may be different when run with the same inputs multiple times. If the Construction query outputs fewer than N words, that means that I-LaSer could not generate any other words in the language which can coexist with the words it has already generated. This may mean that no more words exist in the language, but it does not necessarily mean that I-LaSer has generated the language with the most possible words for the given length (a block maximal language). It does mean that the generated language is close to being block maximal.
The requirement that the construction alphabet must be a subset of the transducer's prohibits transducers with only letters in their input alphabet, or a numerical alphabet that does not contain 0 or other digits less than S.
Fixed types and trajectories are not useful as properties for language construction, as any langauge made up of words which all have the same length will satisfy every fixed type and every trajectory, and the construction option only creates words of a uniform length, so in practice, using either of these property options will simply generate arbitrary words over the alphabet.
If I-Laser generates 40 or more words of Construction output, they will not fit neatly on the screen, so a file dialog will appear, propmting for a directory to save the result in. The file saved in that directory will be named "I-LaSer-Result-[Timestamp]", and I-LaSer will display the name of the file in the result display.

10 References

References

[1] André Almeida, Marco Almeida, José Alves, Nelma Moreira, and Rogério Reis. "FAdo and GUItar". In: vol. 5642. Lecture Notes in Computer Science. Springer, 2009. DOI: 10.1007/978-3-642-02979-0_10. URL: https://doi.org/10.1007/978-3-642-02979-0%5C_10.

[2] Michael Domaratzki. "Trajectory-based codes". In: *Acta Informatica* 40 (2004).

[3] Krystian Dudzinski and Stavros Konstantinidis. "Formal Descriptions of Code Properties: Decidability, Complexity, Implementation". In: *Int. J. Found. Comput. Sci.* 23 (2012). DOI: 10.1142/S0129054112400059. URL: https://doi.org/10.1142/S0129054112400059.

[4] Stavros Konstantinidis, Mitja Mastnak, Nelma Moreira, and Rogério Reis. "Approximate NFA universality and related problems motivated by information theory". In: *Theor. Comput. Sci.* (2023). DOI: 10.1016/J.TCS.2023.114076. URL: https://doi.org/10.1016/j.tcs.2023.114076.

[5] Stavros Konstantinidis, Casey Meijer, Nelma Moreira, and Rogério Reis. "Symbolic Manipulation of Code Properties". In: *J. Autom. Lang. Comb.* 23 (2018). DOI: 10.25596/JALC-2018-243. URL: https://doi.org/10.25596/jalc-2018-243.

[6] Stavros Konstantinidis, Nelma Moreira, and Rogério Reis. "Randomized generation of error control codes with automata and transducers". In: *RAIRO Theor. Informatics Appl.* 52 (2018). DOI: 10.1051/ITA/2018015. URL: https://doi.org/10.1051/ita/2018015.

[7] Darrell R. Raymond. "Grail: a C++ library for finite-state machines and regular expressions". In: 1994. URL: https://dl.acm.org/citation.cfm?id=782245.

[8] Rogério Reis and Nelma Moreira. *FAdo Documentation*. URL: https://www.dcc.fc.up.pt/~rvr/FAdo.pdf (visited on 12/06/2023).