

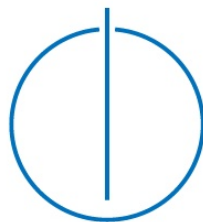
**Technische Universität
München**

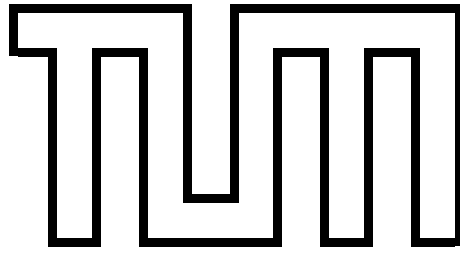
Department of Informatics

Bachelor's Thesis in Information Systems

A system for checking equivalence of representations of regular
languages

Fabio Bove





Technische Universität
München

Department of Informatics

Bachelor's Thesis in Information Systems

A system for checking equivalence of representations of regular
languages

Ein System zur Überprüfung der Äquivalenz von
Repräsentationen regulärer Sprachen

Author: Fabio Bove

Supervisor: Prof. Dr. Javier Esparza

Advisor: Prof. Dr. Javier Esparza

Submission: 15.08.2015

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

München, 15.08.2015

(Fabio Bove)

Abstract

Under the aspect of gamification I developed a system for checking equivalence of various representations of regular and ω -regular languages. Playing this game, the user is shown a representation of an (ω -)regular language and is supposed to give an equivalent representation of another kind. If the input is incorrect, the user is shown a word that distinguishes the given representation from the entered one and the user can try again. The game supports (non-)deterministic finite state automata, regular expressions, monadic second order logic (MSOL) formulas, Büchi automata, ω -regular expressions, and linear temporal logic (LTL) formulas. Additionally, a data structure and a parser for MSOL formulas were developed.

Inhaltsangabe

Unter dem Aspekt der Gamification habe ich ein System zur Überprüfung der Äquivalenz von verschiedenen Repräsentationen regulärer und ω -regulärer Sprachen entwickelt. Im Verlauf des Spiels sieht der Spieler eine Repräsentation einer (ω -)regulären Sprache und soll dazu eine äquivalente Repräsentation einer anderen Form eingeben. Ist die Eingabe nicht korrekt, so wird dem User ein Wort gezeigt, das die beiden Repräsentationen unterscheidet und der User kann einen neuen Versuch starten. Es werden (nicht-)deterministische endliche Automaten, reguläre Ausdrücke, Monadic Second Order Logic (MSOL) Formeln, Büchi-Automaten, ω -reguläre Ausdrücke und Linear Temporal Logic (LTL) Formeln unterstützt. Zusätzlich habe ich eine Datenstruktur und einen Parser für MSOL-Formeln implementiert.

Contents

1	Introduction	3
1.1	Related work	4
1.2	Problem definition	5
1.2.1	Functional requirements	5
1.2.2	Non functional requirements	5
2	Theoretical foundation	7
2.1	Regular languages	7
2.2	ω -regular languages	10
3	The conversion game	14
3.1	Functionality	14
3.1.1	Menu structure	14
3.1.2	Game process	15
3.1.3	Game logic	16
3.1.4	User interface design	17
3.2	Modifying the existing conversion game modes	18
3.2.1	(De-)Activating and adding MSOL macros	18
3.2.2	Changing available levels	19
3.3	Adding new functionalities to the conversion game	20
3.3.1	Adding new representations of (ω)-regular languages	20
3.3.2	Adding new game modes	21
4	Implementation of the Monadic Second Order Logic	26
4.1	Data structure	26
4.2	Parser	27
4.3	Conversion to FSA	28
4.4	Help, editor, and input dialog	29
5	Conclusion	30

<i>CONTENTS</i>	2
List of figures	33
List of tables	34
Appendices	34
A UML class diagrams	35
B User Manual	38
C JFLAP Copyright	45

Chapter 1

Introduction

Computer Science students often have more difficulties with automata theory courses than with other computer science lectures[Gram 99]. According to Gramond and Rodger this has two reasons. Firstly, the automata theory traditionally requires more mathematical basics than other courses. Without those, students can't follow the important proofs. Secondly, immediate feedback when working with automata theory problems is crucial, yet not given if students try to solve these with pencil and paper. Even animations can be useless, if they are simply shown to the students as the solution[Lawr 94]. They need to work on problems and receive immediate feedback in order to understand the concepts of automata theory. For example, Rodger has found that access to the JFLAP tool [Rodg 06] made automata theory course concepts easier to comprehend and rendered the course more enjoyable to the students[Rodg 09].

In order to achieve higher student engagement, the education sector increasingly implements gamification [Kapp 12]. Searching the literature, one finds multiple definitions for gamification, mostly depending on the context. The most commonly used one can be found in [Dete 11] where Deterding et al. define gamification as „the use of game design elements in non-game contexts“. The higher engagement can be explained by Deci and Ryan's model of extrinsic and intrinsic motivation[Ryan 00a]. According to their model, games may trigger the latter. That means that students don't play the game because they are forced to or because they are offered a reward for doing so, but rather because it's actually fun. They also found out that intrinsic motivation may lead to better long term learning outcomes in academic contexts[Ryan 00b]. McGrath and Bayerlein show the benefits of gamification in the context of online learning in [McGr 13].

On the other hand, it still isn't clear how exactly gamification influences engagement with contents and subsequently learning outcome in educational context. A literature review[Hama 14] showed that gamification mostly provides positive effects on engagement but varies greatly on the context it is used in. Koivisto and Hamari have shown that aspects such as age and gender have an effect on the benefits of gamifying exercises[Koiv 14]. Some researchers even sound a note of caution when it comes to gamification. When focusing solely on the game or offering too great rewards, the extrinsic motivation is augmented and the intrinsic one is reduced[Ryan 00a]. For example, Hanus[Hanu 15] conducted a longitudinal study where he found out that a lecture which was completely focused on scoring high in a course intern game led to lower motivation and satisfaction of the students and even to worse exam results compared to the non gamified pendant.

Nevertheless, gamification offers a good opportunity to bring variety to university courses if used moderately.

1.1 Related work

There already is a number of simulation tools for (ω -)regular languages. For example, the language emulator tool [Viei 04] can simulate all kinds of finite word automata as well as regular expressions but doesn't offer the functionality of comparing. Rodger's open source project JFLAP [Rodg 06] incorporates functionalities for comparing (non-)deterministic finite word automata as well as converting regular expressions, DFAs, and NFAs into one another. Norton[Nort 09] has implemented an extension for JFLAP that gives feedback on comparing two finite word automata. The second generation of the GOAL project[Tsai 13](<http://goal.im.ntu.edu.tw>) handles (non-)deterministic finite word automata and regular expressions as well as Büchi automata, ω -regular expressions, and linear temporal logic formulas. A comparison of any two representations returns a distinguishing word in the case of nonequivalence.

Yet none of these offer the combined functionality of converting one representation into another in the form of a dialog with immediate feedback. Moreover, monadic second order logic formulas are handled in none of the mentioned.

1.2 Problem definition

The scientific findings mentioned above suggest an experimental use of a conversion game for various kinds of representations of (ω -)regular languages as a supplement for the automata theory lecture. Considering the lack of such a feature in all related work, the goal of this project is to implement such a system. Beforehand, the following requirements have been laid down in consultation with Prof. Esparza:

1.2.1 Functional requirements

The tool must contain an ordinary checking mode. It allows the user to input two representations of (ω -)regular languages and the system computes if they are equivalent. If they are representations of different languages, the system ought to return a word of minimal length that distinguishes the languages. This way the user cannot only play with given representations but also work with examples that come to his/her mind.

Secondly, the system must implement an exercise mode. In this mode the system proposes a representation of an (ω -)regular language. The user is then challenged to give an equivalent representation of another kind. If the user's input proves not equivalent to the given representation, the system must return a word of minimal length distinguishing the given from the entered representation and display it along with a note about which language contains it. Afterwards, the user must have the possibility to easily modify his input to try again. To help the user keep track of his past tries, a history must be available that displays the distinguishing word of each guess. In order to help the students of the lecture „Automata theory“ [Espa 12], all language representations occurring in that course must be dealt with. These are (non-)deterministic finite state automata, regular expressions, and monadic second order logic formulas for regular languages as well as Büchi automata, ω -regular expressions, and linear temporal logic formulas for ω -regular languages.

1.2.2 Non functional requirements

It is crucial that the user can concentrate on the task of converting one representation to another. Therefore, he must not be frustrated by a cumbersome user interface. The UI design has to be comfortable for the user. A graphical user interface for drawing automata is required to ensure convenient automaton handling.

Furthermore, the user must not be kept waiting by the computation of the equivalence. Small examples (automata with less than ten states) have to require less than a second for being compared. As the system is intended to be used supplementary for a university course (as opposed to being used in research), large examples don't need to be considered. To enable the user to start playing conversion games quickly, a user manual that is to be offered alongside the system must be provided.

Course contents are not fixed forever. In order to ensure easy future enhancement of the system with new language representations or new types of conversion games, a modular implementation is necessary.

Chapter 2

Theoretical foundation

As the implemented conversion game is intended to serve as support for students attending the lecture „Automa Theory“, the theoretical approach behind this game will follow the corresponding script[Espa 12]. The following sections briefly explain the concepts of regular and ω -regular languages. Furthermore, the theory behind the representation forms that are so far implemented in the conversion game will be examined. For more details please refer to the aforementioned script.

2.1 Regular languages

Regular languages are a subset of formal languages. They are defined over a set, whose elements are called letters. A word is formed by a finite, possibly empty sequence of letters. The empty word is commonly denoted as ϵ .

A language is called regular if it is represented by a regular expression. But they can also be represented by (non)deterministic finite automata and monadic second order logic sentences. All of these ways of representations are equal in terms of power. That means that there is no MSOL sentence, finite automaton or regular expression, for whose language there is no instance of any of the other representation forms. Hence, MSOL sentences, regular expressions, and finite automata represent the same class of formal languages, namely the regular ones. In this introduction we will look at the regular language $L_{example} := \{w \in \Sigma^* \mid w \text{ starts and ends with an } a \text{ or is the empty word}\}$ as an example where we assume the alphabet $\Sigma = \{a, b\}$.

Regular expressions

Regular expressions over an alphabet Σ are defined by the grammar

$$r := \emptyset \mid \epsilon \mid a \mid r_1 r_2 \mid r_1 + r_2 \mid r^*$$

where a is a letter of the alphabet ($a \in \Sigma$) and r_1, r_2 are regular expressions over Σ themselves. The regular language $L(r)$ represented by the regular expression r is inductively defined as:

$$\begin{aligned} L(\emptyset) &= \emptyset \\ L(\epsilon) &= \{\epsilon\} \\ L(a) &= \{a\} \\ L(r_1 r_2) &= L(r_1) L(r_2) = \{uv \mid u \in L(r_1), v \in L(r_2)\} \\ L(r_1 + r_2) &= L(r_1) \cup L(r_2) \\ L(r^*) &= L(r)^* \end{aligned}$$

One correct regular expression for the example language $L_{example}$ is the following: $\epsilon + a + a(a + b)^* a$.

Deterministic finite automata

A deterministic finite automaton (DFA) is a quintupel $A = (Q, \Sigma, \delta, q_0, F)$, where

- Q is a set of states,
- Σ is an alphabet,
- $\delta: Q \times \Sigma \rightarrow Q$ is a transition function,
- q_0 is the initial state, and
- F is a set of final states.

We say that A accepts the word $w = a_0 a_1 a_2 \dots a_n \in \Sigma^*$ if there exists a finite sequence $q_0 \rightarrow q_1 \rightarrow \dots \rightarrow q_{n+1}$ where q_{n+1} is a final state ($q_{n+1} \in F$) and $\forall 0 \leq i < n : \delta(q_i, a_i) = q_{i+1}$. The regular language accepted by the DFA A is denoted by $L(A) = \{w \in \Sigma^* \mid w \text{ is accepted by } A\}$. The DFA in figure 2.1a represents $L_{example}$.

Nondeterministic finite automata

A nondeterministic finite automaton (NFA) is a quintupel $A = (Q, \Sigma, \delta, q_0, F)$, where

- Q is a set of states,
- Σ is an alphabet,
- $\delta: \mathcal{P}(Q) \times (\Sigma \cup \{\epsilon\}) \rightarrow \mathcal{P}(Q)$ is a transition function,
- q_0 is an initial states, and
- F is a set of final states.

The initial set of states is $Q_0 = \{q_0\}$. We say that A accepts the word $w = a_0a_1a_2\dots a_n \in \Sigma^*$ if there exists a finite sequence $Q_0 \rightarrow Q_1 \rightarrow \dots \rightarrow Q_{n+1}$ where $Q_{n+1} \cap F$ is not empty ($Q_{n+1} \cap F \neq \emptyset$) and $\forall 0 \leq i < n : \delta(Q_i, a_i) = Q_{i+1}$. Again $L(A) = \{w \in \Sigma^* \mid w \text{ is accepted by } A\}$ indicates the regular language accepted by the NFA A . An NFA accepting $L_{example}$ is given in figure 2.1b. As figure 2.1b has been created with JFLAP, the ϵ is represented by a λ .

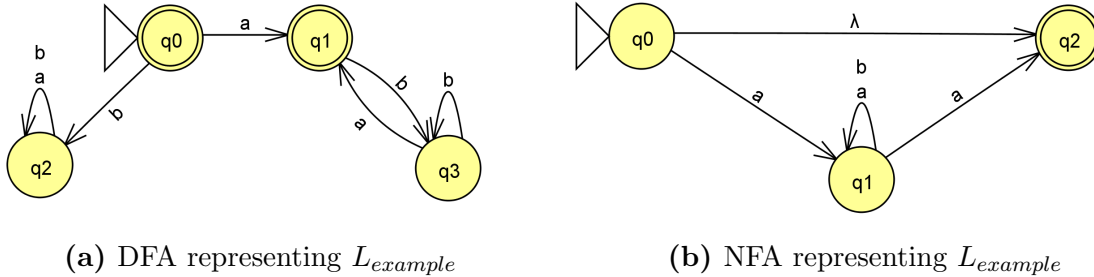


Figure 2.1: DFA and NFA representing the example language $L_{example}$

Monadic second order logic

Regular languages can also be expressed in monadic second order logic (MSOL) over an alphabet Σ . First order variables ranging over the possible positions of letters in a word are denoted by lower case letters: $x \in \mathbb{N}_0$. Second order variables are sets of first order variables and denoted by upper case letters: $X \in \mathcal{P}(\mathbb{N}_0)$ An MSOL formula is defined inductively by the grammar

$$\varphi := Q_a(x) \mid x < y \mid x \in X \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \exists x: \varphi \mid \exists X: \varphi$$

where a is an arbitrary letter of the alphabet and φ_1, φ_2 are MSOL formulas themselves. The intended meaning of the predicate $Q_a(x)$ is that the letter at position x is an a . As usual in predicate logic, variables within the scope of an existential quantifier are called bounded, otherwise free. We call a formula without free variables a sentence.

For our purposes we only look at the interpretation of MSOL **sentences** on words. We inductively define when a word $w = a_0a_1a_2\dots a_n \in \Sigma^*$ satisfies (\models) an MSOL sentence as

$$\begin{aligned} w \models Q_a(x) & \quad \text{iff} \quad a_x = a \\ w \models x < y & \quad \text{iff} \quad x < y \\ w \models x \in X & \quad \text{iff} \quad x \in X \\ w \models \neg\varphi & \quad \text{iff} \quad w \not\models \varphi \\ w \models \varphi_1 \vee \varphi_2 & \quad \text{iff} \quad w \models \varphi_1 \text{ or } w \models \varphi_2 \\ w \models \exists x: \varphi & \quad \text{iff} \quad |w| \geq 1 \text{ and } \exists i \in \{1, \dots, n\}: w \models \varphi[i/x] \\ w \models \exists X: \varphi & \quad \text{iff} \quad |w| \geq 1 \text{ and } \exists S \subseteq \{1, \dots, n\}: w \models \varphi[S/X] \end{aligned}$$

where $w \models \varphi[i/x]$ means that w satisfies the sentence φ , but with all occurrences of x replaced by i and, similarly, $w \models \varphi[S/X]$ means that w satisfies the sentence φ , but with all occurrences of X replaced by S . Macros for easier construction of MSOL formulas can be defined without great effort. Some of the most commonly used ones are

$$\begin{aligned} \varphi_1 \wedge \varphi_2 & := \neg(\neg\varphi_1 \vee \neg\varphi_2) & \varphi_1 \rightarrow \varphi_2 & := \neg\varphi_1 \vee \varphi_2 \\ \text{first}(x) & := \neg\exists y: y < x & \text{last}(x) & := \neg\exists y: y > x \end{aligned}$$

Lastly, we define the language $L(\varphi)$ accepted by the MSOL sentence φ as the set of words over the given alphabet that satisfy φ : $L(\varphi) = \{w \in \Sigma^*: w \models \varphi\}$. In order to stay uniform here is an MSOL formula that accepts L_{example} : $(\neg\exists x: \text{first}(x)) \mid (\exists x: \text{first}(x) \wedge a(x)) \wedge (\exists x: \text{last}(x) \wedge a(x))$. As stated before, the set of languages accepted by MSOL sentences is equivalent to the set of regular languages.

2.2 ω -regular languages

So far we only considered regular languages consisting of arbitrarily long, but finite words. Now we are going to look at languages of infinite words. We define an ω -word over an alphabet Σ to be an infinite sequence of letters in Σ : $w = a_1a_2\dots$. By Σ^ω we denote the set of all ω -words over Σ . A subset $L_\omega \subseteq \Sigma^\omega$ is called an ω -language. The ω -iteration of any formal language L is the ω -language $L^\omega = \{w_1w_2w_3\dots \mid \forall i \in \mathbb{N}: w_i \in L\}$. We call an ω -language L_ω ω -regular if it is described by an ω -regular expression s (as described below). It can be shown that ω -regular expressions, nondeterministic Büchi automata, and

linear temporal logic are equally powerful in expressing regular languages. As an example we take $L_\omega := \{w \in \Sigma^* \mid w \text{ starts with an } a \text{ and contains infinitely many } b\text{'s}\}$ over the alphabet $\Sigma = \{a, b\}$.

ω -regular expressions

One way of describing an ω -languages lies in giving an ω -regular expression. All ω -regular expressions over an alphabet Σ are defined by the grammar

$$s := r^\omega \mid rs_1 \mid s_1 + s_2$$

where r is a regular expression over Σ and s_1, s_2 are ω -regular expressions themselves. The language $L_\omega(s) \subseteq \Sigma^\omega$ described by an ω -regular expression s is inductively defined as

$$\begin{aligned} L_\omega(r^\omega) &= (L(r))^\omega, \\ L_\omega(rs_1) &= L(r)L_\omega(s_1), \text{ and} \\ L_\omega(s_1 + s_2) &= L_\omega(s_1) \cup L_\omega(s_2). \end{aligned}$$

The ω -regular expression $a(a^*b)^\omega$ represents exactly L_ω .

Linear Temporal Logic

Unlike all representations hitherto, linear temporal logic (LTL) formulas are not constructed over an alphabet but over a set of atomic propositions AP . These are combined by the usual boolean operators and the temporal operators **X** („next“) and **U** („until“). Valid LTL formulas are given by the grammar

$$\varphi := \text{true} \mid p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \mathbf{X}\varphi \mid \varphi_1 \mathbf{U}\varphi_2$$

where $p \in AP$.

LTL formulas are interpreted over so called computations, which are infinite sequences $\sigma = \sigma_0\sigma_1\sigma_2\dots$ where $\forall i: \sigma_i \subseteq \mathcal{P}(AP)$. Each σ_i is called a configuration. Furthermore, by σ^j we denote the suffix $\sigma_j\sigma_{j+1}\dots$ of σ . We inductively define that a computation σ satisfies an LTL formula φ ($\sigma \models \varphi$) after the following fashion:

$$\begin{aligned}
\sigma \models \mathbf{true} & \\
\sigma \models p & \text{ iff } p \in \sigma_0 \\
\sigma \models \neg\varphi & \text{ iff } \sigma \not\models \varphi \\
\sigma \models \varphi_1 \wedge \varphi_2 & \text{ iff } \sigma \models \varphi_1 \text{ and } \sigma \models \varphi_2 \\
\sigma \models \mathbf{X}\varphi & \text{ iff } \sigma^1 \models \varphi \\
\sigma \models \varphi_1 \mathbf{U}\varphi_2 & \text{ iff } \exists k \geq 0: \sigma^k \models \varphi_2 \text{ and } \sigma^i \models \varphi_1 \text{ for every } 0 \leq i < k
\end{aligned}$$

where p is an atomic proposition. Similarly to the MSOL formulas there are macros for LTL formulas. The most often used ones are $\mathbf{F}\varphi := \mathbf{true} \mathbf{U}\varphi$ and $\mathbf{G}\varphi := \neg\mathbf{F}\neg\varphi$. That means that $\sigma \models \mathbf{F}\varphi$ iff there exists a $k \geq 0$ such that $\sigma^k \models \varphi$ and $\sigma \models \mathbf{G}\varphi$ iff $\sigma^k \models \varphi$ for all $k \geq 0$.

The set of all computations that satisfy an LTL formula φ is denoted by $L(\varphi)$. LTL formulas can also be used to describe ω -languages, though a little less intuitively. To do so, we see an ω -word $w = w_0w_1w_2\dots$ as a computation $\sigma = \sigma_0\sigma_1\sigma_2\dots$ where each letter w_k corresponds to a configuration σ_k . Moreover, we define an atomic proposition for each symbol that appears in the alphabet of the ω -language. In our example that would be $AP = \{a, b\}$. Now each proposition is assigned the following meaning: σ_k fulfills $z \in AP$ if and only if $w_k = z$. That implies that each configuration can only fulfill one atomic proposition. Consequently, an ω -word is accepted by an LTL formula constructed as described if it satisfies the formula.

However, it can be shown that LTL formulas have the same expressive power regarding ω -languages as NBAs and ω -regular expressions. A correct LTL formula for our example L_ω is $\varphi = (a \wedge \neg b)\mathbf{G} \mathbf{F} (\neg a \wedge b)$.

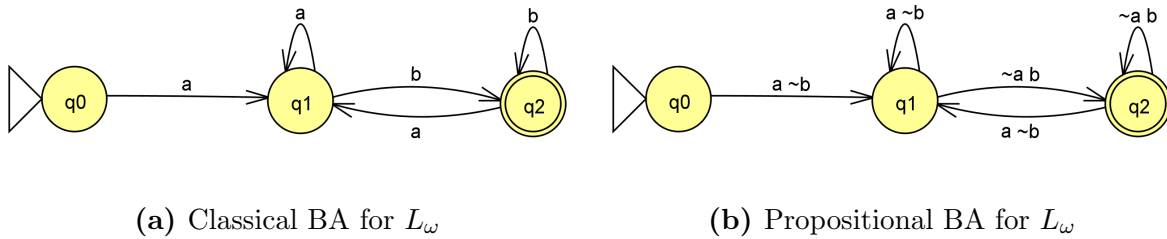
Nondeterministic Büchi automata

Nondeterministic Büchi automata (NBAs) have the same syntax as NFAs. That is, they are denoted by a quintuple $(Q, \Sigma, \delta, q_0, F)$, where

- Q is a set of states,
- Σ is an alphabet,
- $\delta: Q \times \Sigma \rightarrow \mathcal{P}(Q)$ is a transition function,
- q_0 is an initial states, and
- F is a set of accepting states.

Unlike in the case of finite words, we now call F the set of accepting states. We say that an NBA accepts an ω -word $w = a_0a_1a_2\dots$ if there exists an infinite sequence $\rho = q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow \dots$ such that $\forall i \in \mathbb{N}_0: q_{i+1} \in \delta(q_i, a_i)$ and $F \cap \text{inf}(\rho) \neq \emptyset$ where $\text{inf}(\rho) = \{q \in Q \mid q_i = q \text{ for infinitely many } i\}$. Consistently, we define the language $L(A)$ accepted by an NBA A as the set of ω -words that are accepted by the automaton: $L(A) = \{w \in \Sigma^\omega \mid A \text{ accepts } w\}$. For example, figure 2.2a shows a classic Büchi automaton for L_ω .

Figure 2.2: Büchi automata for the example language ω



However, in the ω -regular case the alphabet may be replaced by a set of atomic propositions $AP = \{p_1, p_2, \dots, p_n\}$ and the transition function by one of the form $\delta: Q \times \mathcal{P}(\overline{AP}) \rightarrow Q$ where $\overline{AP} = AP \cup \{\neg p \mid p \in AP\}$. This kind of representation is used when considering computations. Let SAP_i be the set of satisfied atomic propositions in configuration σ_i . For all $i \in \mathbb{N}$ and for all $1 \leq k \leq n$ SAP_i must exclusively satisfy either $p_k \in SAP_i$ or $\neg p_k \in SAP_i$. We say that a BA with propositional alphabet accepts the computation $\sigma = \sigma_0\sigma_1\sigma_2\dots$ if there exists an infinite sequence $\rho = q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow \dots$ such that $\forall i \in \mathbb{N}_0: q_{i+1} \in \delta(q_i, SAP_i)$ and $F \cap \text{inf}(\rho) \neq \emptyset$ where $\text{inf}(\rho) = \{q \in Q \mid q_i = q \text{ for infinitely many } i\}$ and SAP .

Considering the existence of a symbol $a \in \Sigma$ at position $i \in \mathbb{N}_0$ as the atomic proposition $a \in AP$ fulfilled at configuration σ_i , we get the Büchi automaton for the example language in figure 2.2b.

Chapter 3

The conversion game

The main feature of this work is the conversion game. It allows students to rationally guess a representation of a (ω)-regular language based on another representation of the same language. For example, the student is shown a deterministic finite automaton and is supposed to give a regular expression that accepts the same language as the automaton. This chapter explains all its functionalities and how to add and change game types.

3.1 Functionality

Throughout the project the requirements (see section 1.2) have served as a guide for the implementation. The following sections describe which functionalities have been realized and how.

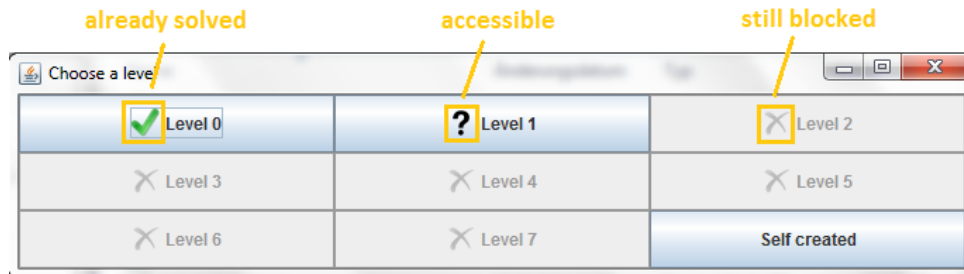
3.1.1 Menu structure

This conversion game has been implemented as an extension to JFLAP. Therefore a menu item „Conversion Game“ has been appended at the bottom of the existing JFLAP menu. Once selected, the user has to choose if he wants to play a conversion game with regular or with ω -regular languages. He then has the choice which kind of representation to guess and which to guess for. As one can see in table 3.1 guessing the same kind of representation as is already given is not supported as it makes no sense. Furthermore, guessing an NFA for a DFA is not offered since every DFA already is an NFA by definition.

Table 3.1: Supported conversion games

(a) Regular languages					(b) ω -regular languages			
	DFA	NFA	RE	MSOL		BA	ORE	LTL
DFA	✗	✓	✓	✓		✗	✓	✓
NFA	✗	✗	✓	✓	BA	✓	✗	✓
RE	✓	✓	✗	✓	ORE	✓	✓	✗
MSOL	✓	✓	✓	✗	LTL			

After having chosen the game type, the user can select one of the given levels (see figure 3.1). A question mark in the corresponding button signals that the level is accessible but not yet solved. Already solved (and therefore also accessible levels) bear a checkmark. Blocked levels have to be made accessible by solving the antecedent one and are marked with a cross. The user also has the option to create a conversion game himself. To do so, the „Self created“ button has to be clicked and an instance of the chosen kind of representation must be entered.

**Figure 3.1:** Menu for choosing a game level

3.1.2 Game process

All games have the same process and basic frame structure in common.

The user is given a representation of an (ω)-regular language. The task is to give an instance of another kind of representation that accepts the same language as the given one. The attempt to do so is called a try in this context. In each try the user can propose a solution. The level is considered solved if the input is correct. Moreover, the stage is marked solved in the level choice menu and the user is asked if he wants to start a new game, review the game just solved, or quit the game. If not, the user is shown a distinguishing word and which representation contains it and which one doesn't. Furthermore, an entry in the game history is created. The user can always go back to previous tries and check

his solution proposed at that point. Combined with the game history this proves to be a convenient way to switch between past tries and check which word distinguishes which guess from the given representation. This offers a chance to include earlier attempts in the considerations for the next try.

3.1.3 Game logic

At the heart of the whole game lies the comparison of the given representation and the one entered by the user. At this point GOAL comes into play. The second generation of GOAL [Tsai 13] provides the functionality of comparing two finite word automata or two Büchi automata with each other as long as they have the same type of alphabet. Therefore, all representations are first converted to automata. GOAL can compare these and return a distinguishing word if the representations are not equivalent.

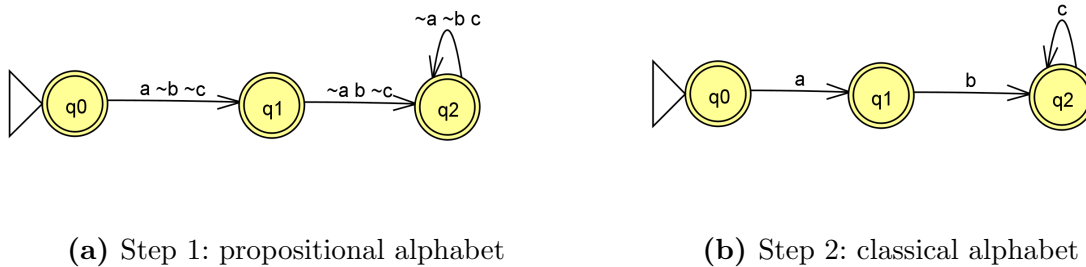
For converting automata between JFLAP and GOAL format the methods

- `ToGOALConverter#convert(automaton: FiniteStateAutomaton, alphabet: AlphabetType, isBuchi: boolean): FSA` and
- `FromGOALConverter#convert(automaton: FSA): FiniteStateAutomaton`

have been implemented. Comparing regular language representations poses no problem, nor does the comparison of Büchi automata with LTL formulas or ω -regular expressions. If, however, an ORE is to be compared to an LTL formula, it gets a little more complicated. An ORE can only be converted to a BA with classical alphabet, while LTL formulas can only be converted to BAs with propositional alphabet. Therefore, the LTL formula is extended by an appendix that makes sure, that each configuration satisfies exactly one atomic proposition. This will result in every transition fulfilling only one atomic proposition after conversion to a BA with propositional alphabet. For instance, if an LTL formula φ over the alphabet $\Sigma = \{a, b, c\}$ is to be compared to an ORE, the appendix $\rho = G((a \wedge \neg b \wedge \neg c) \vee (\neg a \wedge b \wedge \neg c) \vee (\neg a \wedge \neg b \wedge c))$ will be added by a conjunction and the LTL formula $\varphi \wedge \rho$ will be converted to a BA with propositional alphabet (see figure 3.2a). Afterwards the transitions are renamed with the only satisfied atomic proposition (for example, ' $a \wedge \neg b \wedge \neg c$ ' are renamed ' a '), thus rendering the alphabet classical (see figure 3.2b). Now it can be compared to the BA that is equivalent to the ORE.

It should also be noted that the alphabet of the given language is always determined automatically except for the case that an MSOL formula is given, where the alphabet is passed alongside the formula.

Figure 3.2: Difference between classical and propositional alphabet $a \wedge X b \wedge XXG c$



3.1.4 User interface design

In order to provide a uniform appearance, all types of conversion games have the same window structure. The top area shows a textual description of the game, e.g. „Guess the regular expression for the shown DFA“. The alphabet the language is constructed over is shown beneath this instruction if it doesn't emerge clearly from the given representation. One more line further down feedback appears after clicking the compare button. This can be either a success message, or a distinguishing word with a note which representations accepts the word and which not.

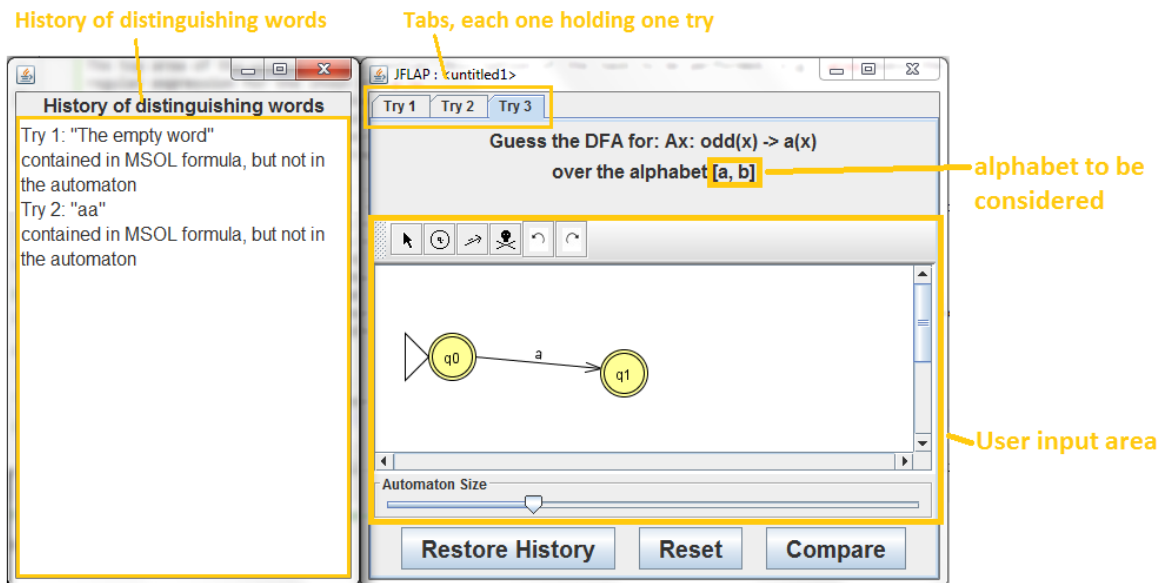


Figure 3.3: Structure of the conversion game UI

In the central area there always is an element that allows the user to enter his proposed solution. Should an automaton be the given representation, it is displayed in the central

area to the left, while the input field will be displayed on the right. In the bottom part of the window three buttons are placed. The „Compare“ button triggers the comparison of the given and the guessed representation. If the input proves syntactically incorrect (e.g. the user automaton has no initial state), the user is shown an error message and asked to resolve the problem. To restore the guess from the previous guess one can press the „Reset“ button. Note that this button is disabled for the first guess. Lastly, in the case of the conversion game history window having been closed or minimized, the „Restore History“ button can be clicked to make the conversion game history visible again.

3.2 Modifying the existing conversion game modes

There are several ways of modifying the existing game modes. The two most important are described in more detail below.

3.2.1 (De-)Activating and adding MSOL macros

It requires little effort to deactivate macros that have been added for convenience. Within the `parse(String formula)` method every macro has an according section where the matcher for the specific pattern is built. To deactivate a certain macro, that section has to be commented out alongside the following handling of a possibly found match. For example, if one wants to deactivate the 'second(x)' macro, the following code in the `MSOLParser` class has to be commented out:

```
matcher = SecondPattern.matcher(formula);
if(matcher.find()){
    MSOLVariable variable =
        new MSOLVariable(matcher.group(2));
    return new MSOLSecond(variable);
}
```

In order to reactivate a deactivated macro, just remove the comment symbols. On the other hand, for adding a new macro one has to implement the according logic first. A new class should be created within the `conversion_game.MSOL.macros` package. If the macro is an operator on subformula(s) (just like the negation or the alternation), the new macro has to be implemented as a subclass of `MSOLNode`. If, however, the new macro is an operator on variables and/or sets (like $x \in X$ or $first(x)$), it must be implemented as a subclass of `MSOLLeaf`. Particularly, the `toString()` and `toFSA(Set<String> alphabet)`

methods have to be overridden. Moreover, it is crucial to add any free variables or sets that might appear in the new macro to the corresponding `freeVariables` or `freeSets` set, respectively, in the constructor.

To make the parser recognize the macro in the user input, within the `MSOLParser` class a pattern has to be compiled from a proper regular expression:

```
private static Pattern newMacroPattern =
    Pattern.compile(".*+/*Regular expression for macro*/");
}
```

Additionally, some code has to be added to the `parse(String formula)` method:

```
matcher = newMacroPattern.matcher(formula);
if(matcher.find()){
    // build the MSOL formula from the found match
    // return the parsed MSOL formula
}
```

Be careful to place this code according to the precedence of the operator. The higher the precedence, the later the recognition code must appear. For example, the alternation (`'|'`) recognition has been placed after the existential quantifier (`'E'`) recognition but before the negation (`'~'`) recognition.

3.2.2 Changing available levels

One can also change the preset levels for each game mode. To do so, the corresponding files in the data folder need to be edited. For games whose given representations are MSOL formulas (ω)-regular expressions, or LTL formulas, the preset levels can directly be edited by changing the corresponding „expressions.txt“ or „formulas.txt“ text file in the `data/guessXXXForXXX` folder. Be cautious to enter expressions and formulas correctly: Different levels are separated by semicolons. For MSOL formulas a dot separates the expression or formula from the alphabet within one level description. Commas separate the individual symbols within the alphabet. If we consider *formula* an MSOL formula and *nonMSOL* either an (ω)-regular expression, or an LTL formula, the text files holding the preset levels must be written according to *levels* as in figure 3.4.

Naturally, one can also change the preset levels of games that have automata as their given representation. To change the automaton given in level X, it is necessary to open the automaton saved in the file „levelX.jff“ in the folder `data/guessXXXForXXX/automata`.

Figure 3.4: Grammar for the text file which stores (ω) -regular expression or LTL/MSOL formulas
$$\begin{aligned}
 \textit{alphabet} &:= \textit{symbol} (\textit{;} \textit{;} \textit{symbol})^* \\
 \textit{MSOL} &:= \textit{formula} \textit{;} \textit{;} \textit{alphabet} \\
 \textit{levels} &:= \textit{MSOL} (\textit{;} \textit{;} \textit{MSOL})^* \mid \textit{nonMSOL} (\textit{;} \textit{;} \textit{nonMSOL})^*
 \end{aligned}$$

When the editing is done, the automaton can be saved back to the same file. Of course, it is possible to remove automata from the folder as well as adding new automata. One must only make sure that the automata file names are successively named „level0.jff“, „level1.jff“, and so on.

In any case of editing preset levels or changing available macros it is necessary to edit the „version.txt“ text file. If the user has previously downloaded an older version of the conversion game, a higher version number will result in updating the preset levels and resetting the progress.

3.3 Adding new functionalities to the conversion game

The modular design of the implementation allows easy adding of new kinds of representations and types of conversion games. The implementation structure of the game can be examined in the class diagram in Appendix A. An API documentation for GOAL is available at [GOAL].

Furthermore, a logger has been implemented. To use it, set up a static field in the class that references it:

```

private final static Logger LOGGER =
    Logger.getLogger(Logger.GLOBAL_LOGGER_NAME);
}

```

By default the logging is written to the Logging.txt file. Detailed MSOL conversion infos can be obtained by setting the logging level to `Level.FINER`.

3.3.1 Adding new representations of (ω) -regular languages

It proves easy to add new kinds representation of (ω) -regular languages. Obviously, one must first implement the logic of the desired representation form. In the context of the

conversion game, the most important task is to implement a working editor for that representation. For easy integration in the existing conversion game window structure it should be implemented as a subclass of `JComponent`.

Comparisons between two (ω)-regular languages are so far always made by comparing their DFA, or BA respectively, representations. Therefore, it is essential to provide a method that transforms an instance of the newly added representation form to a finite state automaton, be it an NFA in the regular case or a BA in the ω -regular case. Note that in the regular case it suffices to transform to an NFA, as the intern comparison between two NFAs or DFAs first converts both automata to DFAs if necessary.

Furthermore, a method that retrieves the alphabet of the language represented by an instance of the new representation form must be implemented. This proves especially important when comparing to MSOL or LTL formulas.

Lastly, a dialog for entering a self created instances of the new representation form should be implemented in order to allow the user to play conversion games aside from the preset ones.

3.3.2 Adding new game modes

This section describes how to add a new game type to increase the number of available games for the user. We assume that the user is supposed to work out an XXX representation for a given YYY representation of an (ω)-regular language.

Subclass of `GameFileHandler`

First of all, a subclass of `GameFileHandler` is necessary. This class will handle loading the given levels as well as saving and loading the level progress state file. A constructor without parameters needs to set up the correct path to the text file that saves the progress for this type of conversion game:

```
public XXXForYYYGameFileHandler() {
    STATES_PATH = "/data/guessXXXForYYY/levelStates.txt";
}
```

The string in this example is the proposed path for storing the progress states in, as it will be appended to the standard JFLAP conversion game data path, which can be found in `GameFileHandler#STANDARD_JFLAP_PATH`. One must also store the path to

the given representations as a constant in a private static field. The main part of the implementation is writing the loading method for the given representations. A method, named `getYYYRepresentations(): YYY[]` or similarly, has the task of retrieving the given representations from the proper folder within the data folder and returning them as an array. The only abstract superclass method can then be easily overridden in the following way:

```
public Object [] getObjects(){
    return getYYYRepresentations();
}
```

Subclass of ConversionGame

A new conversion game type is bound to be a subclass of the `ConversionGame` class. A private field for the given representation has to be created. A constructor with the following three parameters has to be set up: the given representation (LTL, `FiniteStateAutomaton`, ...), the alphabet/atomic propositions (`String[]`) of the represented language and the level that is to be played (`int`). Firstly, the latter two arguments have to be passed to the superclass constructor `super(level: int, alphabet: String[])`. The remaining one is used to set up the field that stores the given representation.

```
public GuessBAForLTLGame(YYY yyy), String[] alphabet, int level){
    super(level, alphabet);
    this.given = yyy;
}
```

As a subclass of `ConversionGame`, the following abstract methods must be overridden:

isCompareActionApplicable(): boolean Checks if the user entered input is a valid representation. If so, this method must return true. Otherwise, false must be returned and the user should be shown an error message. For games, where the user must guess an NFA for some other representation, the input automaton must be checked for invalid labels (`AutomatonChecker#hasInvalidLabels(FiniteStateAutomaton fsa)`), for determinism (`AutomatonChcker#isDFA(Automaton automaton)`), and for existence of an initial state (`FiniteStateAutomaton#getInitialState()`). If the user is supposed to enter an (ω)-regular expression, an MSOL formula, or an LTL formula, the user input should be parsed. If an error/exception occurs, an according feedback needs to be shown to the user.

createFileHandler() Every game type has its own subclass of a `GameFileHandler`. This method creates a new instance of the proper `GameFileHandler` and assigns it to the according field of the superclass.

newStep(): ConversionGameStep This method is responsible for creating a new conversion game step. The implementation should take the user input from the previous guess and set it as initial input for the newly created step. For the first step an empty input has to be set.

getFeedback(distinguisher: String): String This method is used in the superclass method `checkEquivalence()` after a possible distinguisher has been found. Using the `Equivalence.Result` this method must find out which representation accepts the distinguishing words and must then return an according feedback.

getAutomata(): Pair<FiniteStateAutomaton,FiniteStateAutomaton> Every type of game has two certain representations of languages: the given one and the user entered one. This method converts both (if necessary) to finite state automata and returns them.

getProperGOALFSAs(automata: Pair < FiniteStateAutomaton, FiniteStateAutomaton>): Pair<FSA, FSA> As the comparison of finite state automata is handled by GOAL, the JFLAP format automata have to be converted to GOAL format. To do so, the method `ToGOALConverter#convert` must be used. Depending on the type of automata, the arguments vary. For Büchi automata the last argument must be `true`, whereas for finite word automata it must be `false`. In case of Büchi automata, the second argument indicates if the automaton has an `AlphabetType.CLASSICAL` or `AlphabetType.PROPOSITIONAL` alphabet.

Subclass of `ConversionGameStep`

Consequently, an according subclass of `ConversionGameStep` must be implemented. Therein, a constructor with the two parameters, one for the game the new step belongs to and one for the initial user input, has to be created. The constructor must call the superclass constructor `super(game: ConversionGame)` and has to create a `Component` that contains a proper editor for the user input and, if necessary, a display for the given representation. The area that has to be built can be seen in figure 3.5. As each instance of a subclass of `ConversionGameStep` is initialized with a `BorderLayout` layout manager the built component needs to be added with the `BorderLayout.CENTER` constraint:

```

public GuessXXXForYYYStep(
    GuessXXXForYYYGame game, XXX xxx){
    super(game);
    // build your component here
    this.add(comp, BorderLayout.CENTER);
    instruction.setText(
        "Guess the XXX for: "+game.getYYY.toString());
}

```

Furthermore, the abstract superclass methods `setUserInput(Object object)` and `getUserInput(): Serializable` have to be overridden. The former type checks the passed argument and sets it as current user input. When setting an automaton as user input, the existing automaton editor has to be removed and a new one has to be created and added. Make sure to call `validate()` at the end of the method body. The latter simply returns the user input object.

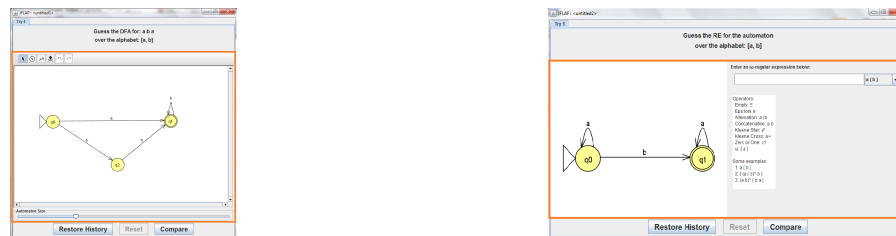


Figure 3.5: The marked area has to be constructed individually for each subclass of `ConversiongameStep`

Subclass of `ChooseNewGame`

In order to create the according level choosing menu, the developer has to create a subclass of `ChooseNewGame`. A constructor without parameters (see listing 3.1) must first call the superclass constructor `super()`. Then a new instance of the proper `GameFileHandler` has to be instantiated and stored in the `fileHandler` field. After that the superclass method `initialize()` has to be invoked. Secondly, as `ChooseNewGame` implements the interface `ActionListener`, but doesn't override the `actionPerformed(e: ActionEvent)`, the subclass must do this. A click on a level button must result in the according level being loaded. The source of the `ActionEvent` argument will always be a `LevelJButton`, which has been implemented to enrich the standard `JButton` with an extra field for the level it ought to lead to. Note that `-1` indicates that the user wants to create a level himself. After having loaded the chosen level or having accepted the user input, the alphabet (or the set

of atomic propositions) has to be determined. So far, for MSOL formulas the alphabet has to be given explicitly by adding it in the text file (see figure 3.4), or letting the user enter it. For all other kinds of representation that are supported so far, methods for automatic recognition of the alphabet or set of atomic propositions exist (see GOAL and JFLAP documentation). Passing the representation loaded or created by the user, the retrieved alphabet/atomic propositions, and the level obtained from the button, an instance of the newly created subclass of `ConversionGame` may be instantiated. It can then be passed to `FrameFactory#createFrame(object: Serializable)`. Lastly, the menu window must be disposed by invoking `dispose()`.

Listing 3.1: Constructor in subclass of `ChooseNewGame`

```
public NewGuessXXXForYYYGame () {
    super ();
    fileHandler = new XXXForYYYGameFileHandler ();
    initialize ();
}
```

Code modifications in existing classes

To bring all together, a few lines have to be added to existing classes. Firstly, in the `ConversionGameEnvironment#actionPerformed(e: ActionEvent)` method the following line has to be added;

```
if (game instanceof GuessXXXForYYYGame) newGame =
    new ChooseXXXForYYYGame ();
```

The proper place is marked in the source code. Secondly, within the `ChooseGameType` class a constant describing the game type has to be created:

```
private static final String
    GUESS_XXX_FOR_YYY = "Guess XXX for YXY";
```

Moreover, in `ChooseGameType#actionPerformed(e: ActionEvent)` the line

```
if (command.equals(ChooseGameType.GUESS_XXX_FOR_YYY))
    newConvGame = new ChooseXXXForYYYGame ();
```

is necessary in the place marked in the source code.

Chapter 4

Implementation of the Monadic Second Order Logic

Neither GOAL, nor JFLAP provide a functionality for dealing with MSOL formulas. As the initial requirements include comparing MSOL formulas against other kinds of representation, the need of implementing MSOL formulas as a whole new kind of representation within JFLAP emerged. In doing so, the following features have been implemented:

- data structure for MSOL formulas
- a parser that constructs MSOL formulas from strings
- a method that converts an MSOL formula to a finite state automaton
- an MSOL help panel, editor, and input dialog

The following describes these features in more detail. For a description of how to add and (de-)activate MSOL macros please refer to section 3.2.1.

4.1 Data structure

To get an overview of the implemented classes, please look at the UML class diagram in figure A.2 within the appendix. To make adding new MSOL macros as easy as possible, a common abstract superclass `MSOLFormula` has been implemented. Hierarchically below that there are two more abstract subclasses `MSOLNode` and `MSOLLeaf`. While new macros that handle subformulas (like the alternation or the negation) must be implemented as

subclasses of the former, whereas ones that contain only variables and sets (like $x < y$ or $first(x)$) have to be made subclasses of the latter.

4.2 Parser

The `MSOLParser` class implements the functionality of parsing MSOL formulas from Strings. The implementation is merely a makeshift solution as the available time did not allow to engage deeply with the art of creating an excellent parser. Assuming that a is a letter of the considered alphabet and φ_1, φ_2 represent correct MSOL formulas, it recognizes the formulas in table 4.1. The parser essentially compares the input string against regular expression patterns for each operator. This is done in an order according to the precedence of the operators. The lower the precedence, the earlier the input string is compared against the according pattern. This ensures that the logic implicitly expressed by the order and the parentheses is maintained.

Note that most of the recognized expressions are macros for easier input of

Table 4.1: Strings recognized by the MSOL parser

description	notation in lecture notes	representation in conversion game
atomic predicate	$Q_a(x)$	$a(x)$
membership	$x \in X$	$x \text{ in } X$
smaller than	$x < y$	$x < y$
negation	$\neg \varphi_1$	$\sim \varphi_1$
alternation	$\varphi_1 \vee \varphi_2$	$\varphi_1 \mid \varphi_1$
existential quantifier (variables)	$\exists x : \varphi_1$	$Ex : \varphi_1$
existential quantifier (sets)	$\exists X : \varphi_1$	$EX : \varphi_1$
universal quantifier	$\forall x : \varphi_1$	$'Ax : \varphi_1$
bigger than	$x > y$	$'x > y'$
equals	$x = y$	$'x = y'$
n bigger than	$x = y + n$	$'x = y + n'$
implication	$\varphi_1 \rightarrow \varphi_2$	$\varphi_1 '=>' \varphi_2$
equivalence	$\varphi_1 \leftrightarrow \varphi_2$	$\varphi_1 '<=>' \varphi_2$
conjunction	$\varphi_1 \wedge \varphi_2$	$\varphi_1 '&' \varphi_2$
even number	$even(x)$	$'even(x)'$
odd number	$odd(x)$	$'even(x)'$
even set	$Even(X)$	$'Even(X)'$
odd set	$Odd(X)$	$'Odd(X)'$
first	$first(x)$	$'first(x)'$
second	$second(x)$	$'second(x)'$
last	$last(x)$	$'last(x)'$

MSOL formulas. As described in section 2.1, the atomic predicate, the membership

predicate, the „smaller than“ predicate, the negation, the alternation, and the existential quantifiers would suffice to represent all regular languages.

A special exception `MSOLParseException` has been implemented to indicate an error occurring while parsing a string for an MSOL formula. The parser recognizes lower case letters as variables and upper case letters except A and E (as they represent the existential and universal quantifier) as sets. The parser additionally checks if a any character is used for a symbol as well as for a variable. If that is the case, a `MSOLParseException` is thrown. The parser does *not* check whether the formula contains free variables. To test a MSOL formula for free variables one must call the `containsFreeVariables()` method on it. Checking for free variables is already included in the `MSOLEditor#getFormula()` method.

How to make the parser recognize new macros and how to (de-)activate the recognition of existing ones is described in section 3.2.1.

4.3 Conversion to FSA

The main functionality of the implemented system is comparing various kinds of representations of regular languages against each other. As distinguishing the according FSAs is the easiest way of comparing two regular languages, a method for converting an MSOL formula to an equivalent FSA is crucial.

The functionality is implemented in the `MSOLFormula#toFSA()` method. In this case, only the symbols appearing in the formula are considered the alphabet. For conversion under an additional alphabet, the methods `MSOLFormula#toFSA(alphabet: String[])` and `MSOLFormula#toFSA(alphabet: Set<String>)` may be used. Figure 4.1 makes the difference clear. When converting the MSOL formula $(\exists x: first(x) \wedge \neg b(x)) \wedge$

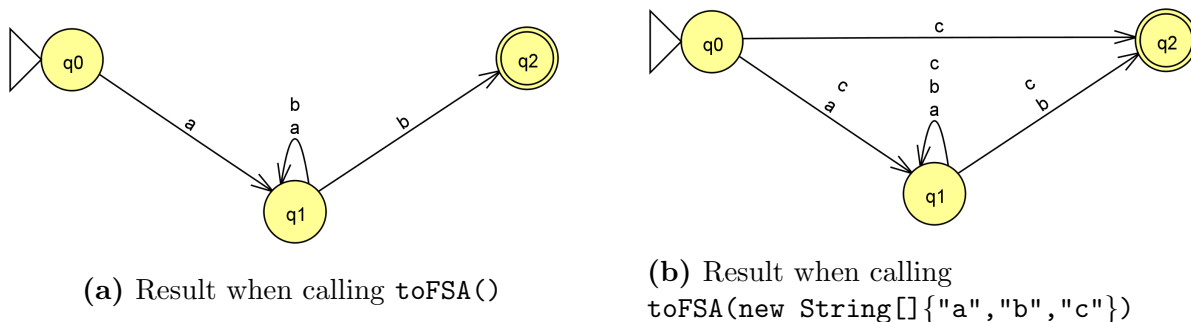


Figure 4.1: Conversion of $(\exists x: first(x) \wedge \neg b(x)) \wedge (\exists x: last(x) \wedge \neg a(x))$

$(\exists x: last(x) \wedge \neg a(x))$ (representing the language that contains all words that don't start with a b and don't end with an a), figure 4.1a shows the result of a call of `toFSA()` on the formula, whereas figure 4.1b shows the result of a call of `toFSA(new String[]{"a","b","c"})` on the formula.

The implementation follows exactly the same pattern as the proof in [Espa 12] that for each regular language L there is a MSOL formula that accepts exactly L .

4.4 Help, editor, and input dialog

A simple help for entering MSOL formula is provided in the `JScrollPane` subclass `MSOLHelp`. When included in the GUI design, it shows the user the notations and available macros for MSOL formulas. The help text can easily be edited by changing the static `String` field `MSOL_HELP_TEXT` within the class.

In order to deal properly with `String` representations of MSOL formulas, an editor has been implemented as a subclass of `JPanel`. Instances of `MSOLEditor` contain a text area for the user input and the help text mentioned above. The method `getStringInput()` returns the literal `String` entered by the user, whereas `getFormula()` immediately tries to parse the input `String` and throws an exception if there are free variables in the input or some other error occurs while parsing.

The `MSOLInputDialog` is basically an `MSOLEditor` wrapped in a modal dialog. If the user enters a syntactically incorrect formula, the user is shown an according error message and prompted to try again. Figure 4.2 shows the individual parts.

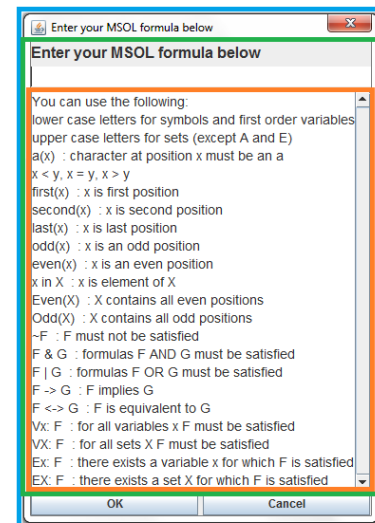


Figure 4.2:
orange: `MSOLHelp`;
green: `MSOLEditor`;
blue: `MSOLInputDialog`

Chapter 5

Conclusion

In the progress of this bachelor's project a system for checking equivalence of various representations of regular and ω -regular languages has been developed. The ordinary checking mode is already included in the JFLAP project. The whole exercise mode has been implemented relying on JFLAP[Rodg 06] and GOAL[Tsai 13]. It offers a gamified dialog where the user is shown a representation of an (ω -)regular language and is supposed to give a representation of another kind for the same language. Immediate feedback on the correctness is shown upon input and saved in a game history which allows the user to easily include his previous guesses in his considerations for next one. (Non-)deterministic finite automata, regular expressions, and monadic second order logic formulas are supported for regular languages. ω -regular languages may be represented by Büchi automata, ω -regular expressions, and linear temporal logic formulas.

For the sake of gamifying the system, game design elements have been implemented. Examples include the existence of levels, immediate feedback upon input, and interactive activities. A user manual has been created allowing the students to quickly start playing. The exercise mode is organized in subsequent levels. By default, levels are only accessible if all previous levels have already been solved before.

As discussed in chapter 1, this systems can be used for gamifying an automata theory course. As this game only covers a small part of the lecture and is intended to be used as a supplement, the danger of undermining the intrinsic motivation of the students is minimal. It merely brings variety to the course contents.

It is recommended that the value of the conversion game in terms of learning outcome and student satisfaction be evaluated in future use. Further work on the conversion game may

include developing more macros for the MSOL representation. The MSOL representation would also benefit from a more solid parser.

Programmers are welcome to modify and reuse the code written for this project, as long as it is not sold for commercial purposes. Please be sure to acknowledge the author. The original JFLAP code is copyrighted as in appendix C.

List of Figures

2.1	Example DFA and NFA	9
2.2	Example Büchi automata	13
3.1	Menu for choosing a game level	15
3.2	Difference between classical and propositional alphabet	17
3.3	Structure of the conversion game UI	17
3.4	Grammar for storing levels in txt files	20
3.5	Individual area of conversion game steps	24
4.1	Differences when using <code>toFSA()</code> with or without an argument	28
4.2	MSOL UI components	29
A.1	Class diagram of game structure	36
A.2	Class diagram of MSOL structure	37
B.2	Menu for choosing a game level	39
B.3	The after game dialog	39
B.4	Structure of the conversion game UI	40
B.6	Example Büchi automata	44

List of Tables

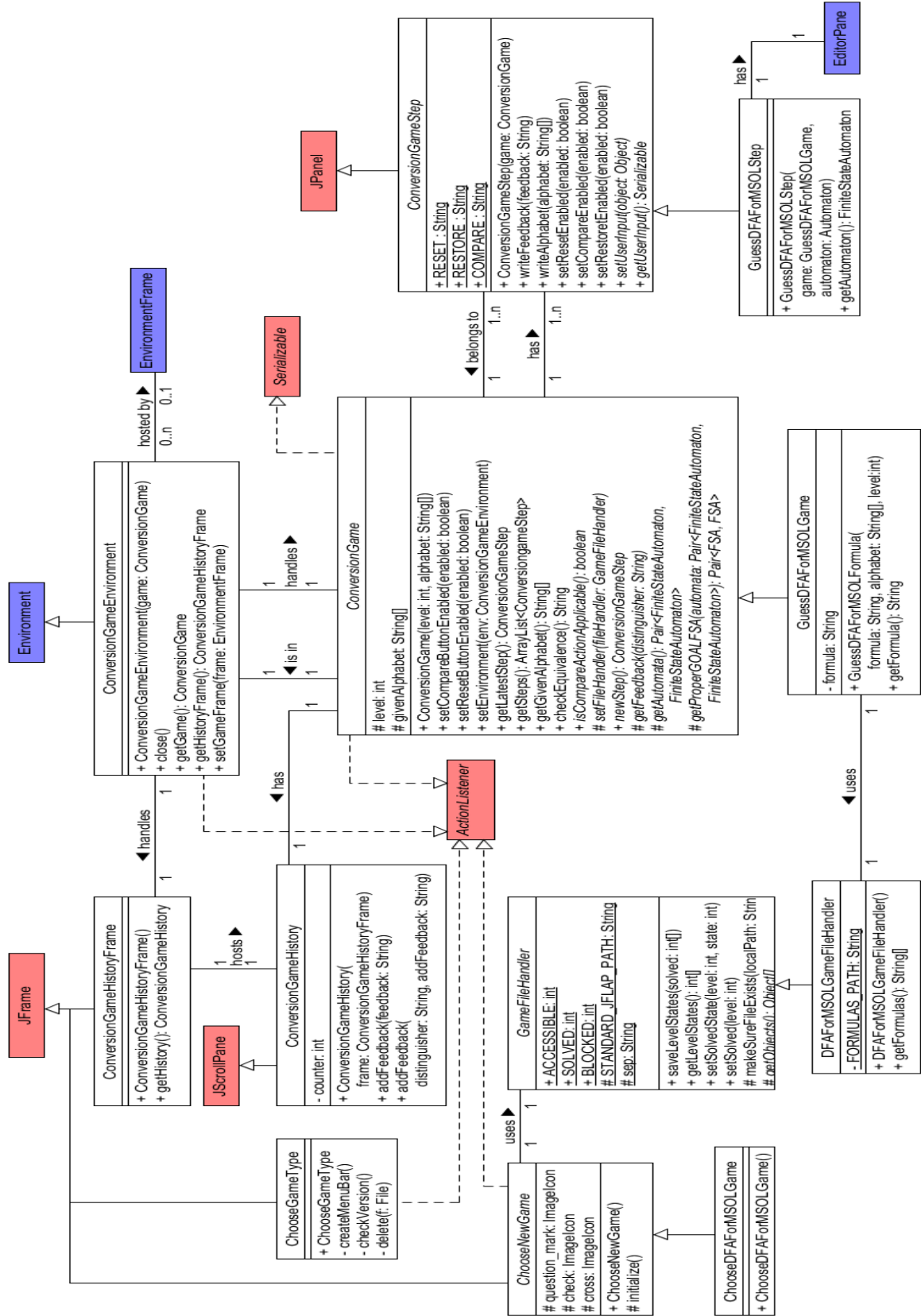
3.1	Supported conversion games	15
4.1	Strings recognized by the MSOL parser	27
B.1	Available conversion game types	39

Appendix A

UML class diagrams

All diagrams obey the rules of UML 2.0. To learn more about UML, please refer to www.uml.org. The class diagram in A.1 shows the structure of the general game process. Class diagram A.2 shows the data structure of the MSOL implementation. In order to make the borders of the own implementation clearer, standard Java classes as well as classes from JFLAP have been colored. JFLAP classes bear the color red, whereas the standard Java classes are painted blue.

Figure A.1: Class diagram of game structure



Appendix B

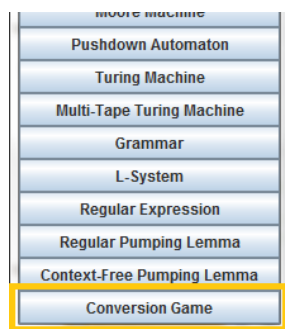
User Manual

This guide aims to explain the practical use of the conversion game for (ω -)regular languages. It was designed for supplementary use for the lecture „Automata Theory“. If you get lost with some notations, you can look them up in the according script [Espa 12].

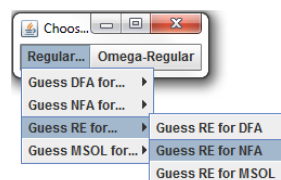
Starting a game

Firstly, a Java 8 runtime environment is required. You can get it at <https://www.java.com/de/download/>. You can download the enhanced JFLAP version from the website of the TUM Chair for Foundations of Software Reliability and Theoretical Computer Science (<https://www7.in.tum.de/tools/index.php?id=tools&arg=>).

To start a new conversion game, select the bottom button of the menu, namely „Conversion Game“ (see figure B.1a). After that you can choose which type of game you want to play (see figure B.1b). In table B.1 you can see the available conversion games.



(a) Click the marked button to start a new conversion game



(b) Choose your game type from the drop out menu

	DFA	NFA	RE	MSOL		BA	ORE	LTL
DFA	✗	✓	✓	✓	BA	✗	✓	✓
NFA	✗	✗	✓	✓	ORE	✓	✗	✓
RE	✓	✓	✗	✓	LTL	✓	✓	✗
MSOL	✓	✓	✓	✗				

Table B.1: Available conversion game types

Once you’ve selected your desired game type, you can choose which of the preset levels you want to play (see figure B.2). Buttons bearing a question mark lead to accessible but not yet solved levels. Already solved levels are signaled by a check mark. Disabled buttons with a cross mark indicate levels that are still blocked. Usually you can activate these levels by solving the precedent ones. You also have the freedom to create your own conversion game by clicking the „Self created“ button. A dialog for entering your own representation of an (ω -)regular language will then appear.

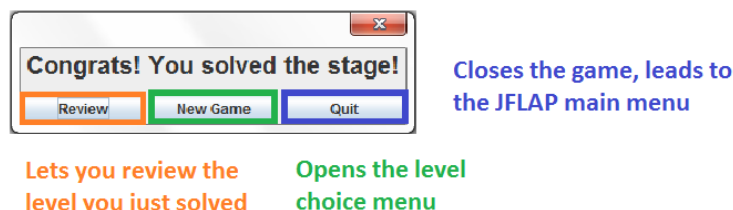


Figure B.2: Menu for choosing a game level

After having chosen the game level or having entered your own representation, the game starts. All games have the same process and basic frame structure in common.

You are given a representation of an (ω -)regular language. The task is to enter an instance of another kind of representation that accepts the same language as the given one. The attempt to do so is called a **try** in this context. In each try you can propose a solution. The level is considered solved if the input is

Figure B.3: The after game dialog



correct. A dialog will ask if you want to start a new game, review the game just solved, or quit the game(see figure B.3). If not, you will get feedback in the form of a distinguishing word and a note about which representation contains it and which one doesn’t. Then

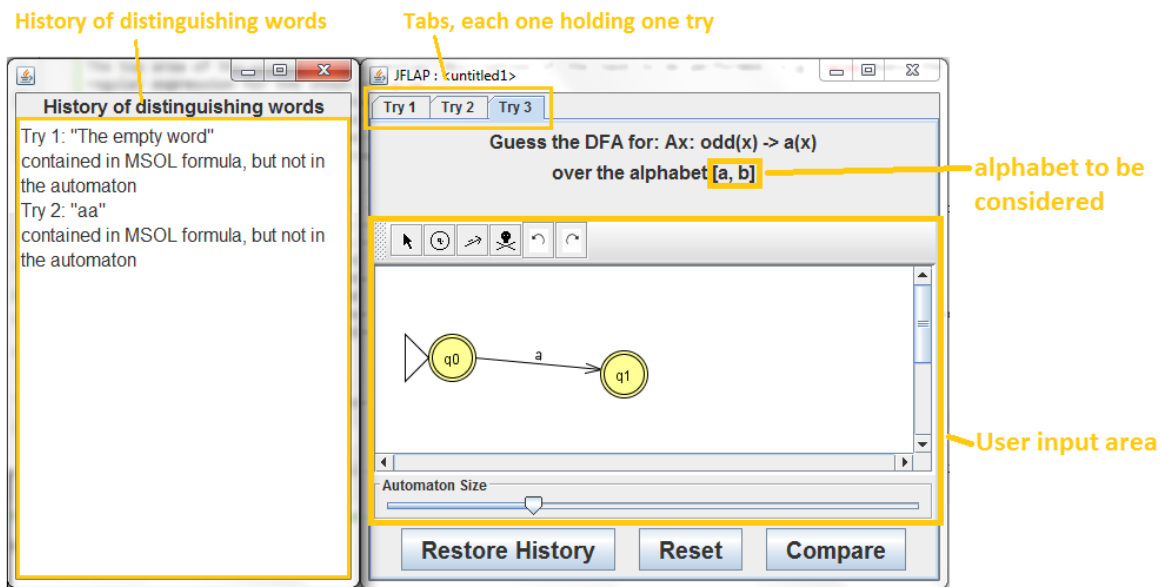


Figure B.4: Structure of the conversion game UI

another tab will be created where you can enter a fresh guess. Furthermore, an entry in the game history is created. You can always go back to previous tries by clicking on the tabs and check your solution proposed at that point. This offers a chance to include earlier attempts in the considerations for the next try.

All games have the same basic window structure (see figure B.4). The top area shows a textual description of the game, the alphabet (if necessary), and textual feedback after the „Compare“ button has been clicked. In the central area there always is an element that allows you to enter your proposed solution. Should an automaton be the given representation, it is displayed in the central area to the left, while the input field will be displayed on the right. In the bottom part of the window three buttons are placed. The „Compare“ button triggers the comparison of the the given and the guessed representation. To restore the input from the previous guess you can press the „Reset“ button. Note that this button is disabled for the first guess. The conversion game history is placed on the left. It shows you the words that distinguish your previous guesses from the given representation. Should you at any time close or minimize the conversion game history window, you can hit the „Restore History“ button to make the conversion game history visible again.

Kinds of representation

Regular expressions

In this conversion game regular expressions are formed by the grammar

$$\begin{aligned}
 \langle \textit{symbol} \rangle &:= a \mid \dots \mid d \mid f \mid \dots \mid z \\
 \langle \textit{expression} \rangle &:= E \mid e \mid \langle \textit{symbol} \rangle \mid \langle \textit{expression} \rangle _ \langle \textit{expression} \rangle \\
 &\quad \mid (\langle \textit{expression} \rangle)^* \mid (\langle \textit{expression} \rangle)^+ \mid \langle \textit{expression} \rangle^? \\
 &\quad \mid (\langle \textit{expression} \rangle \mid \langle \textit{expression} \rangle)
 \end{aligned}$$

Note that for a concatenation of the symbols a and b a space in between is necessary: a_b instead of ab . In this context e stands for the empty word (ϵ), E for the empty set (\emptyset), and the vertical bar (\mid) represents the alternation ($+$) from the lecture (see page 15 in [Espa 12]). All other operators coincide with the ones from the lecture. The question mark ($?$) is an abbreviation for "zero or one times", i.e. if r is a regular expression, $r^?$ is an abbreviation for $r \mid e$.

Monadic second order logic formulas

The MSOL parser was constructed to accept MSOL sentences as described in the lecture notes starting from page 163. If we assume that φ_1 and φ_2 are correct MSOL formulas, the parser recognizes input after the following fashion:

description	notation in lecture notes	representation in conversion game
atomic predicate	$Q_a(x)$	$a(x)$
membership	$x \in X$	$x \text{ in } X$
smaller than	$x < y$	$x < y$
negation	$\neg \varphi_1$	$\sim \varphi_1$
alternation	$\varphi_1 \vee \varphi_2$	$\varphi_1 \mid \varphi_2$
existential quantifier (variables)	$\exists x : \varphi_1$	$Ex : \varphi_1$
existential quantifier (sets)	$\exists X : \varphi_1$	$EX : \varphi_1$

where a variable is a lower case letter and a set is an upper case letter except A and E . Additionally, there must be no free variables and no symbol of the alphabet may appear as a first order variable. In order to ease constructing MSOL formulas, several macros are available. Among those are

description	notation in lecture notes	pattern (without ' ')
universal quantifier	$\forall x : \varphi_1$	'Ax:' φ_1
bigger than	$x > y$	'x>y'
equals	$x = y$	'x = y'
n bigger than	$x = y + n$	'x = y + n'
implication	$\varphi_1 \rightarrow \varphi_2$	φ_1 '=>' φ_2
equivalence	$\varphi_1 \leftrightarrow \varphi_2$	φ_1 '<=>' φ_2
conjunction	$\varphi_1 \wedge \varphi_2$	φ_1 '&' φ_2
even number	$even(x)$	'even(x)'
odd number	$odd(x)$	'even(x)'
even set	$Even(X)$	'Even(X)'
odd set	$Odd(X)$	'Odd(X)'
first	$first(x)$	'first(x)'
second	$second(x)$	'second(x)'
last	$last(x)$	'last(x)'

where φ_1 and φ_2 are correct MSOL formulas.

DFAs, NFAs, and BAs

DFAs, NFAs, and BAs are drawn with the JFLAP automaton drawer. More information about the usage of this tool can be found at <http://www.jflap.org/tutorial/>. Note that the undo and redo tools are not supported. When drawing a BA with propositional alphabet use the tilde „~“ for negation. Please mind the following hints for common mistakes:

- You can draw an ϵ -transition by creating a standard transition and leaving the label empty. It will be displayed as λ .
- Instead of creating a transition with the label "ab", create two transitions labeled "a" and "b" with a state in between
- If you want several symbols to lead from one state to another, draw one transition for each of those symbols. Labels like "a|b" are not supported.
- When drawing a BA with propositional alphabet, you can use "true" as a label.

ω -regular expressions

ω -regular expressions are formed almost the same way as regular expressions. Following the lecture notes (see page 195), they are constructed following the grammar

$$ore := \{'re'\} \mid re\ ore \mid ore\ ' \mid ore$$

where re is a regular expression as described above. The $'|'$ represents the $+$, while the curly brackets represent the exalted ω from the lecture notes.

Linear Temporal Logic formulas

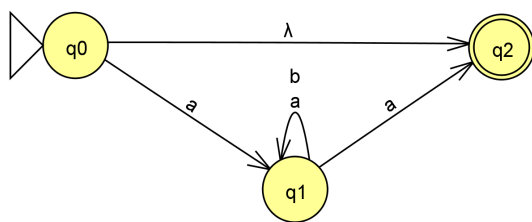
The LTL parser was imported from the GOAL project. To understand the notation you can check the lecture notes starting from page 263. Assuming that φ_1 and φ_2 are correct LTL formulas, it recognizes input after the following fashion:

description	notation in lecture notes	pattern
true	$true$	true
atomic proposition	p	p
negation	$\neg\varphi_1$	$\sim\varphi_1$
alternation	$\varphi_1 \vee \varphi_2$	$\varphi_1 \varphi_2$
conjunction	$\varphi_1 \wedge \varphi_2$	$\varphi_1 \&\& \varphi_2$
next	$\mathbf{X}\varphi_1$	X φ_1
until	$\varphi_1 \mathbf{U}\varphi_2$	$\varphi_1 \text{ U } \varphi_2$
eventually	$\mathbf{F}\varphi_1$	F φ_1
globally	$\mathbf{G}\varphi_1$	G φ_1

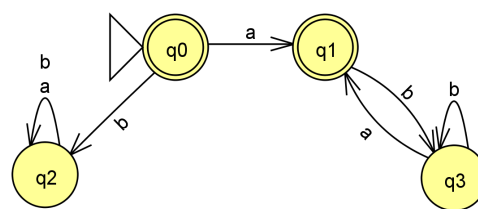
Note that there must be no free variables. To ease the user input when comparing to an ORE, the atomic propositions can only be fulfilled exclusively. That means the the input a is short for $a \&\& \sim b \&\& \sim c$ when considering the set of atomic propositions $\{a, b, c\}$.

Examples

To make the use of the different kinds of representations clearer, please look at these two examples. In the regular case we consider the regular language $L := \{w \in \Sigma^* \mid w \text{ starts and ends with an } a \text{ or is the empty word}\}$ where we assume the alphabet $\Sigma = \{a, b\}$. Below you can see an NFA and a DFA representing L .



(a) NFA for L

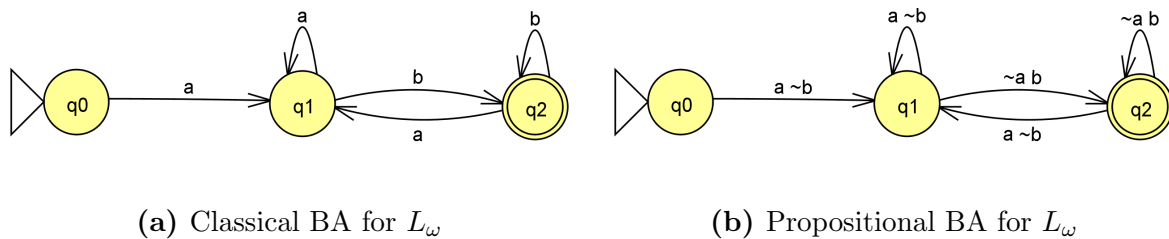


(b) DFA for L

A corresponding regular expression within the conversion game would be written as $e \mid a \mid a (a|b)^* a$. The equivalent MSOL sentence is given by $(\sim \text{Ex: first}(x)) \mid (\text{Ex: first}(x) \ \& \ a(x)) \ \& \ (\text{Ex: last}(x) \ \& \ a(x))$.

In the ω -regular case we consider the language $L_\omega := \{w \in \Sigma^* \mid w \text{ starts with an } a \text{ and contains infinitely many } b\text{'s}\}$. We still assume $\Sigma = \{a, b\}$. Two according BAs with classical and propositional alphabet are shown below:

Figure B.6: Büchi automata for the example language L_ω



The equivalent ω -regular expression in the conversion game is given by $a \{a^* b\}$. A correct LTL sentence for the same language is $a \ \&\& \ \mathbf{G} \ \mathbf{F} \ b$. When comparing BAs with OREs, the BA will have a classical alphabet as in figure B.6a. On the other hand, when comparing BAs against LTL formulas, they will have a propositional alphabet as in figure B.6b.

Appendix C

JFLAP Copyright

JFLAP 7.0 LICENSE

Susan H. Rodger
Computer Science Department
Duke University
August 27, 2009

Duke University students contributing to JFLAP source include: Thomas Finley, Ryan Cavalcante, Stephen Reading, Bart Bressler, Jinghui Lim, Chris Morgan, Kyung Min (Jason) Lee, Jonathan Su and Henry Qin.

Copyright (c) 2002-2009
All rights reserved.

- I) You are allowed distribute unmodified copies of JFLAP under the following two conditions:
- 1) You must include a copy of this license text.
 - 2) You cannot charge a fee for any product that includes any part of JFLAP, in source or binary form.
- II) You are allowed to distribute modified copies of JFLAP under the following conditions:
- 1) You must include a copy of this license text.
 - 2) You cannot charge a fee for any product that includes any part of your modified JFLAP, in source or binary form.

- 3) If you made the changes yourself,
you must clearly describe how to contact you.
When the maintainer asks you (in any way) for a copy of the
modified JFLAP you distributed, you must make your changes,
including source code, available to the maintainer without fee.
The maintainer reserves the right to include your changes in the
official version of JFLAP.
The current maintainer is Susan Rodger.
If this changes, it will be announced at www.jflap.org.

The name of the author may not be used to
endorse or promote products derived from this software without
specific prior written permission.

THIS SOFTWARE IS PROVIDED ‘‘AS IS’’ AND WITHOUT ANY EXPRESS OR
IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Bibliography

- [Dete 11] S. Deterding, D. Dixon, R. Khaled, and L. Nacke. “From Game Design Elements to Gamefulness: Defining ”Gamification”. In: *Proceedings of the 15th International Academic MindTrek Conference: Envisioning Future Media Environments*, pp. 9–15, ACM, New York, NY, USA, 2011.
- [Espa 12] J. Esparza. “Automata theory: An algorithmic approach”. 2012. An algorithmic approach.
- [GOAL] “GOAL API Document: <http://goal.im.ntu.edu.tw/api/latest/>”.
- [Gram 99] E. Gramond and S. H. Rodger. “Using JFLAP to Interact with Theorems in Automata Theory”. *SIGCSE Bull*, Vol. 31, No. 1, pp. 336–340, 1999.
- [Hama 14] J. Hamari, J. Koivisto, and H. Sarsa. “Does Gamification Work? – A Literature Review of Empirical Studies on Gamification”. In: *2014 47th Hawaii International Conference on System Sciences (HICSS)*, pp. 3025–3034, 2014.
- [Hanu 15] M. D. Hanus and J. Fox. “Assessing the effects of gamification in the classroom: A longitudinal study on intrinsic motivation, social comparison, satisfaction, effort, and academic performance”. *Computers & Education*, Vol. 80, pp. 152–161, 2015.
- [Kapp 12] K. M. Kapp. *The gamification of learning and instruction: game-based methods and strategies for training and education*. John Wiley & Sons, 2012.
- [Koiv 14] J. Koivisto and J. Hamari. “Demographic differences in perceived benefits from gamification”. *Computers in Human Behavior*, Vol. 35, pp. 179–188, 2014.
- [Lawr 94] A. W. Lawrence, A. M. Badre, and J. T. Stasko. “Empirically evaluating the use of animations to teach algorithms”. In: *1994 IEEE Symposium on Visual Languages*, pp. 48–54, 4-7 Oct. 1994.
- [McGr 13] N. McGrath and L. Bayerlein. “Engaging online students through the gamification of learning materials: The present and the future”. *Australasian*

Society for Computers in Learning in Tertiary Education (ASCILITE), Sydney, Australia, 2013.

- [Nort 09] D. A. Norton. “JFLAP Finite Automata: Feedback/Grading Tool”. 2009.
- [Rodg 06] S. H. Rodger and T. W. Finley. *JFLAP: an interactive formal languages and automata package*. Jones & Bartlett Learning, 2006.
- [Rodg 09] S. H. Rodger, E. Wiebe, K. M. Lee, C. Morgan, K. Omar, and J. Su. “Increasing engagement in automata theory with JFLAP”. *ACM SIGCSE Bulletin*, Vol. 41, No. 1, pp. 403–407, 2009.
- [Ryan 00a] R. M. Ryan and E. L. Deci. “Intrinsic and extrinsic motivations: Classic definitions and new directions”. *Contemporary educational psychology*, Vol. 25, No. 1, pp. 54–67, 2000.
- [Ryan 00b] R. M. Ryan and E. L. Deci. “Self-determination theory and the facilitation of intrinsic motivation, social development, and well-being”. *American psychologist*, Vol. 55, No. 1, p. 68, 2000.
- [Tsai 13] M.-H. Tsai, Y.-K. Tsay, and Y.-S. Hwang. “GOAL for Games, Omega-Automata, and Logics”. In: N. Sharygina and H. Veith, Eds., *Computer Aided Verification*, pp. 883–889, Springer Berlin Heidelberg, 2013.
- [Viei 04] L. F. M. Vieira, M. A. M. Vieira, and N. J. Vieira. “Language Emulator, a Helpful Toolkit in the Learning Process of Computer Theory”. *SIGCSE Bull*, Vol. 36, No. 1, pp. 135–139, 2004.