



DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Support of pushdown automata in Automata Tutor

Sebastian Mair





DEPARTMENT OF INFORMATICS

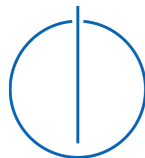
TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Support of pushdown automata in Automata Tutor

Unterstützung von Kellerautomaten in Automata Tutor

| | |
|------------------|------------------------------------|
| Author: | Sebastian Mair |
| Supervisor: | Jan Kretinsky |
| Advisor: | Julia Krämer, Maximilian Weininger |
| Submission Date: | October 15, 2018 |



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

Munich, October 15, 2018

Sebastian Mair

Acknowledgments

I would like to thank my advisor Maximilian Weininger for his continual active support for this Bachelor's thesis. I would also like to thank the Hanns-Seidel-Foundation for their financial and ideational support during this Bachelor.

Abstract

In this thesis the learning platform Automata-Tutor, which is used by the chair for theoretical computer science for the lecture "Introduction to theoretical computer science", was expanded with the functionality of pushdown automata (PDAs) to greatly simplify creating and evaluating exercises about PDAs. Now instructors can easily create problems of two different types concerning PDAs. Then the students can solve these problems and get an automatically generated rating and a feedback, if the submitted solution was not correct. This feedback is helpful for finding and fixing the mistake when doing further submissions. A special focus is on a comfortable and clear creation of pushdown automata, which shall evoke a deeper understanding on PDAs, and on an efficient implementation of the used grading algorithms.

Zusammenfassung

Im Rahmen dieser Bachelorarbeit wurde die Lernplattform Automata-Tutor, die vom Lehrstuhl für Theoretische Informatik für die Vorlesung "Einführung in die Theoretische Informatik" verwendet wird, um die Funktionalität der Kellerautomaten (PDAs) erweitert. So ist es nun Lehrenden mit wenig Aufwand möglich, zwei verschiedene Typen von Aufgaben zu PDAs zu erstellen, die dann von den Studierenden bearbeitet werden können und automatisch bewertet werden. Die Bewertung beinhaltet dabei im Falle einer fehlerhaften Abgabe ein Feedback, das den Studierenden für die folgenden Abgaberversuche beim Auffinden des Fehlers hilft. Besonderer Fokus liegt hierbei auf einer komfortablen und anschaulichen Erstellung von Kellerautomaten, die das Verständnis von deren Funktionsweise vertiefen soll, und auf einer effizienten Implementierung der Algorithmen, die zur Bewertung verwendet werden.

Contents

| | |
|--|------------|
| Acknowledgments | iii |
| Abstract | iv |
| Zusammenfassung | v |
| 1 Introduction | 1 |
| 1.1 Motivation | 1 |
| 1.2 Related work | 1 |
| 1.3 Outline | 2 |
| 2 Preliminaries | 3 |
| 2.1 Theoretical preliminaries | 3 |
| 2.2 Introduction to Automata Tutor | 6 |
| 2.2.1 Usage of Automata Tutor | 6 |
| 2.2.2 Structure of the project | 7 |
| 2.2.2.1 Frontend | 7 |
| 2.2.2.2 Backend | 8 |
| 3 Creation of the PDA functionality | 9 |
| 3.1 Exercise types | 9 |
| 3.2 Realization of the frontend | 10 |
| 3.2.1 GUI for building PDAs | 10 |
| 3.2.1.1 User functionality | 10 |
| 3.2.1.2 Technical view | 14 |
| 3.2.2 Creation of the new exercise types | 20 |
| 3.3 Realization of the backend | 21 |
| 3.3.1 Software architecture | 22 |
| 3.3.2 Word problem | 24 |
| 3.3.2.1 Naive approach | 24 |
| 3.3.2.2 Implemented approach | 25 |
| 3.3.2.3 Word problem for DPDAs | 29 |
| 3.3.3 Determination of equality | 29 |

Contents

| | | |
|----------|---|-----------|
| 3.3.4 | Simulation | 31 |
| 3.3.4.1 | Naive approach | 32 |
| 3.3.4.2 | Implemented approach | 34 |
| 3.3.4.3 | Simulation in a DPDA | 38 |
| 3.3.5 | Graders | 40 |
| 3.3.5.1 | English to PDA grader | 40 |
| 3.3.5.2 | Words of PDA grader | 42 |
| 4 | Student evaluation | 43 |
| 4.1 | Overview | 43 |
| 4.2 | Questionnaire and answers | 44 |
| 4.3 | Improvements after the evaluation | 46 |
| 5 | Conclusion and outlook | 48 |
| | List of Figures | 49 |
| | List of Tables | 50 |
| | Bibliography | 51 |

1 Introduction

1.1 Motivation

The topic of this bachelor's thesis is interesting in two respects. Pushdown Automata (PDAs) are a bit difficult to understand at the beginning compared with for example deterministic finite automata (DFAs). So from the didactical point of view it is an exciting task to clarify the way of working of pushdown automata (PDAs) to students. Therefore it is important and very helpful for students, that the learning platform Automata Tutor also provides the functionality of PDAs now. Due to a clear and simple creation of PDAs, the possibility to run a PDA simulation for a specific word and a helpful feedback when submitting an incorrect PDA, the understanding of PDAs can be greatly deepened. Furthermore, the automated feedback and rating of the students' solution attempts avoids extra work for the instructors, because checking the correctness of a given PDA manually is a relatively expensive task, especially when it has to be done multiple times.

On the other hand, the realization of the functionality of PDAs is also interesting in itself. Creating a model for PDAs that is as generic as possible as well as implementing and developing some algorithms related to PDAs were exciting tasks and lead to a deeper understanding of PDAs for the authors of this thesis, too.

1.2 Related work

The learning platform Automata Tutor already allows creating and solving many different types of exercises in the field of theoretical computer science, which range from creating DFAs for given languages over applying the product construction up to running the CYK-algorithm manually. Automata Tutor was built by D'Antoni, Loris et al. ([1]) and extended in the context of former bachelor theses, as in the current case [2, 3, 4, 5].

1.3 Outline

In the next section we introduce some important definitions and notations, that we use in this thesis. Then we give a short introduction to Automata Tutor, where we explain how to use Automata Tutor and how the project is structured into the frontend and backend part. The section after that is the main part of this thesis: we explain, which new types of exercises were added to the platform and how that was realized. This includes the implementation of the graphical user interface for building, presenting and simulating PDAs, the basic infrastructure added to the frontend as well as the realization of the backend part, including how to determine whether two given PDAs can be regarded as equal. The fifth part deals with feedback and suggestions of students. As part of this work, we conducted an evaluation, where students were asked to solve exercises about PDAs. In the subsequent questionnaire they were, inter alia, asked about the usability. The final section contains a summary and an outlook for the future of Automata Tutor and further steps concerning PDAs.

2 Preliminaries

2.1 Theoretical preliminaries

Now we introduce some relevant definitions and notations about PDAs and related terms.

PDA [6] A PDA is defined as 7-tuple $M = (Q, \Sigma, \Gamma, q_0, Z_0, \delta, F)$, where Q is a finite set of states, Σ is the finite input alphabet, Γ is the finite stack alphabet, $q_0 \in Q$ is the initial state and $Z_0 \in \Gamma$ is the initial stack symbol. $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \rightarrow P_\epsilon(Q \times \Gamma^*)$ is the transition function, where $P_\epsilon(X)$ stands for the set of all finite subsets of X . $F \subseteq Q$ is a set of final states. The states together with the transition function δ are interpreted as a state transition machine with a stack in the following way: let $(q', \alpha) \in \delta(q, a, Z)$ be a transition; if M is in state q , reads the input symbol a and Z is the top most symbol in the stack, then M can transit to state q' , pop Z from the stack and push α to the stack. If a is ϵ , then M can "enter" the transition without reading an input symbol. The left most stack symbol of α is thereby the new top most symbol of the stack. Apart from representing the transitions as a function, they can be notated as follows, which will be referred to as *arrow-notation* in the later chapters: if $(q', \beta) \in \delta(q, a, Z)$ where $a \in \Sigma \cup \{\epsilon\}$, then we write $q \xrightarrow{a, Z/\beta} q'$.

Figure 2.1 shows an example PDA, where the states and transitions are represented as a state machine. The double circle of the state 1 marks this state as final.

Configuration [6] A configuration of a PDA M is a triple (q, ω, α) with $q \in Q$, $\omega \in \Sigma^*$ and $\alpha \in \Gamma^*$. It is a snapshot of a PDA, where q is the current state, ω the remaining part of the input word and α the current stack (whose top most symbol is the left most in α). The configuration, that starts the PDA in figure 2.1 for the word ab at, is $(0, ab, Z)$

Relation on configurations [6] The binary relation \rightarrow_M is defined over the set of all configurations as follows:

$$(q, a\omega, Z\alpha) \rightarrow_M \begin{cases} (q', \omega, \beta\alpha) & \text{if } (q', \beta) \in \delta(q, a, Z) \text{ and } a \in \Sigma \\ (q', a\omega, \beta\alpha) & \text{if } (q', \beta) \in \delta(q, \epsilon, Z) \end{cases}$$

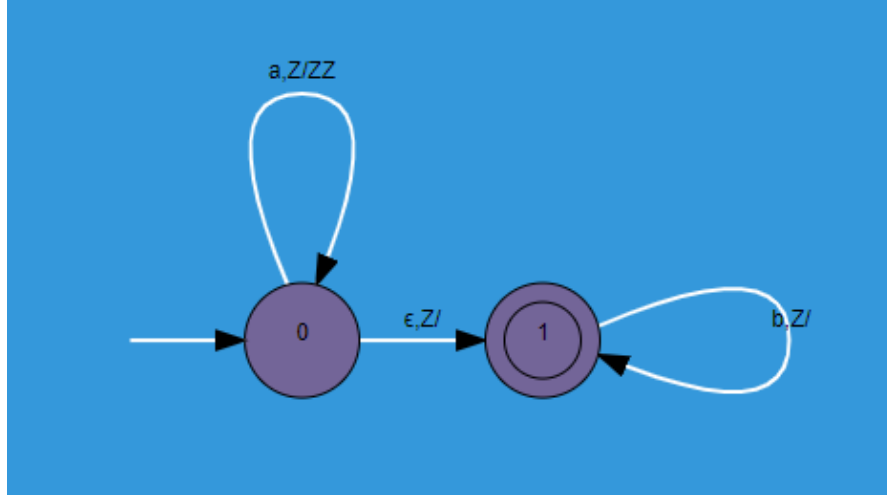


Figure 2.1: Example PDA with $\Sigma = \{a, b\}$ and $\Gamma = \{Z\}$

Therefore, \rightarrow_M indicates whether a configuration can go into another configuration with one transition. Because of the nondeterminism, a configuration can have several descendants regarding \rightarrow_M . The reflexive and transitive closure of \rightarrow_M , notated as \rightarrow_M^* , therefore specifies if a configuration can transit to another one with an arbitrary number of transitions. For the PDA in figure 2.1 for example holds the following: $(0, ab, ZZ) \rightarrow_M \{(0, b, ZZZ), (0, ab, Z)\}$

Language of a PDA [6] The language a PDA M accepts can be defined by two ways, either by the set F of final states or by the stack of the PDA only. The language accepted by final states is defined as $L_F(M) := \{\omega | \exists f \in F, \gamma \in \Gamma^*. (q_0, \omega, Z) \rightarrow_M^* (f, \epsilon, \gamma)\}$. The language accepted by empty stack is defined by $L_\epsilon(M) := \{\omega | \exists q \in Q. (q_0, \omega, Z_0) \rightarrow_M^* (q, \epsilon, \epsilon)\}$. Therefore, the set F in the definition of M has no meaning when considering L_ϵ .

The language, that the PDA in figure 2.1 accepts by final states, is exactly the set $\{a^n b^m | n \geq 0 \wedge m \leq n\}$, whereas the language accepted by empty stack is $\{a^n b^n | n \geq 0\}$. The acceptance by final states and by empty stack are equally powerful, which means that for every PDA M PDAs M_1 and M_2 can be created, so that $L_F(M) = L_\epsilon(M_1)$ and $L_\epsilon(M) = L_F(M_2)$. Finally, the languages that PDAs can accept are exactly the context free languages (CFL), so for every CFL a PDA can be created that accepts this language and the language any PDA accepts is context free.

When a PDA is said to accept a specific language by *final states* and *empty stack*, then this means that the PDA accepts this language by *final states* respectively by *empty stack*.

However, this does not mean, that for every word in the language the PDA must stop both in a final state and with an empty stack simultaneously.

Path of a word A path of a word ω is a tuple of configurations of a PDA and is defined as follows: if the PDA does not accept the word, it is the empty tuple $()$. If the PDA does, then a path is a tuple of configurations (c_0, \dots, c_n) with minimal length n , where $c_0 = (q_0, \omega, Z_0)$, so that $\forall 0 \leq i < n | c_i \rightarrow_M c_{i+1}$ and $c_n = (q', \epsilon, \beta)$ with $q' \in F$ or $\beta = \epsilon$, depending on the acceptance condition of the PDA. So, intuitively, a path of a word is a simulation of this word in the PDA. For the PDA in figure 2.1, the path $((0, ab, Z), (0, b, ZZ), (1, b, Z), (1, \epsilon, \epsilon))$ belongs to the word ab .

DPDA [6] Additionally, the so called deterministic PDAs (DPDAs) form a subset of the PDAs. A PDA is deterministic, if and only if $\forall q \in Q. \forall a \in \Sigma. \forall Z \in \Gamma. |\delta(q, a, Z)| + |\delta(q, \epsilon, Z)| \leq 1$. That implies, that any configuration of the PDA has at most one possible descendant regarding \rightarrow_M :

$$\forall q \in Q. \forall \omega \in \Sigma^*. \forall \alpha \in \Gamma^*. |\{(q', \rho, \beta) | (q, \omega, \alpha) \rightarrow_M (q', \rho, \beta)\}| \leq 1.$$

The PDA in figure 2.1 is not deterministic.

The languages that DPDAs accept by final states are the so called deterministic context free languages (DCFL), which are a proper subset of the CFLs. The languages that DPDAs accept by empty stack are a subset of the DCFLs, namely all DCFLs that fulfill the so called prefix condition. According to this condition the language must not contain two words, where the one is a proper prefix of the other one. That implies that the acceptance by final states and empty stack are not equally powerful for DPDAs.

CFG [6] A CFL can also be represented by a context free grammar (CFG). This is a 4-tuple $G = (V, \Sigma, P, S)$, where V is a finite set of nonterminals and Σ is the alphabet containing the terminals. Σ and V have to be disjoint. $P \subset V \times (V \cup \Sigma)^*$ is the finite set of productions and $S \in V$ is the start symbol. A production $(\alpha, \beta) \in P$ can be represented as $\alpha \rightarrow \beta$.

As PDAs and CFGs both accept the CFLs, they are equally powerful and can be converted to each other.

Derivation [6] The so called derivation relation \rightarrow_G of a CFG G is defined over the set of strings of $V \cup \Sigma$ (that means over $(V \cup \Sigma)^*$) as follows: $\alpha \rightarrow_G \alpha'$ if and only if there is a production $(\beta \rightarrow \beta') \in P$ and strings $\alpha_1, \alpha_2 \in (V \cup \Sigma)^*$ so that $\alpha = \alpha_1 \beta \alpha_2$ and $\alpha' = \alpha_1 \beta' \alpha_2$. Then \rightarrow_G^* is defined as the reflexive transitive closure of \rightarrow_G . Furthermore, $\alpha_1 \rightarrow_G^* \alpha_n$ is called a derivation of α_n from α_1 . Now the language of the CFG is defined as $L(G) = \{\beta \in \Sigma^* | S \rightarrow_G^* \beta\}$.

Useful symbols [6] A symbol $X \in V \cup \Sigma$ is called producing, if there is a derivation $X \rightarrow_G^* \omega$ with $\omega \in \Sigma^*$, and it is said to be reachable, if there is a derivation $S \rightarrow_G^* \alpha X \beta$ with $\alpha, \beta \in (V \cup \Sigma)^*$. Furthermore, X is useful, if there is a derivation $S \rightarrow_G^* \omega$, that contains X (where $\omega \in \Sigma^*$ again). If at first all not producing symbols and all productions where they occur are removed from a CFG G , leading to the resulting CFG G' , and from G' all not reachable symbols are removed with the result G'' , then G'' only contains useful symbols and has the same language as G .

Chain and epsilon productions A production $A \rightarrow B$ is called a chain production. An epsilon production has the form $A \rightarrow \epsilon$.

2NF A CFG is said to be in 2 normal form (2NF), if for every production $A \rightarrow \alpha$ holds that $|\alpha| \leq 2$.

CNF A CFG is said to be in Chomsky-normal-form (CNF), if every production has the form $A \rightarrow a$ or $A \rightarrow BC$, where $A, B, C \in V$ and $a \in \Sigma$. Any CFG G can be converted into a CFG G' in CNF, so that $L(G) = L(G') \setminus \{\epsilon\}$.

2.2 Introduction to Automata Tutor

The tool Automata Tutor is a web application, which can be found under the URL automata.model.in.tum.de. In the following we describe the basic principle of the usage of Automata Tutor and give an overview over the structure of the project.

2.2.1 Usage of Automata Tutor

After signing up or logging in Automata Tutor offers different user interfaces depending on whether one is an instructor or a student (or both). Instructors can manage the list of all problems, that means he can create new ones, edit and delete existing problems. A problem is an unreleased exercise. Each problem has a specific type, e.g. "English to DFA", more precisely a problem is an instance of this problem type. During runtime of Automata Tutor the problem types are preconfigured, that means in order to add a problem the project code has to be extended. After creating all desired problems, the instructor can arrange them in a named problem set, where he can add any of the existing problems. For every problem added to the set, he can define the maximum grade and the number of attempts allowed for this problem.

Then an instructor can mark specific problem sets as practice, that means that any logged in student can solve the problems in the set under the menu entry "Practice

Problem Sets" as often as he wants. This feature is used for optional free training of the students. Furthermore, the instructor can pose problem sets to one of his courses, so that all logged in users, which have access to that course, can solve the problems in the set. When a student submits his solution, he gets an automated computed grade and in the case of a incorrect solution also a feedback, which should help to fix the mistake. Thereby they have only the defined number of attempts, and the final grade and last submission of each problem are saved. Among other things, the problems published in the courses are used to allow solving exercises online instead of written form.

2.2.2 Structure of the project

The Automata Tutor project is structured into two sub projects, the frontend and the backend. The frontend is for rendering the web page and managing the database with the problems, solution attempts etc.. The backend is for more complicated and expensive calculations, as for example computing the grade of a solution attempt of a student. The backend can be called by the frontend over AJAX whenever it is necessary. The communication between both uses XML as data exchange format. The description of their structures now is limited to the aspects that are relevant for this thesis.

2.2.2.1 Frontend

The frontend is written in Scala and uses Lift as web framework. The Scala-part consists of four packages, of which two concern the problems. The package `com.automatatutor.model` contains a Scala-class for every problem type. This class models a problem, which means it contains all necessary properties and methods, e.g. the correct solution. The model classes are used by the frontend to generate database schemata for storing the problems permanently, that are created by the instructors. This is comfortable, as only the model class must be defined in order to add a new problem type, and the database need not be manipulated directly. The file `SolutionAttempt` is also in the model-package and contains a class for every problem type. This is used to model and save the attempt of a user, so that the user can go on solving a problem at another time and does not need to restart completely. Therefore, also for the `SolutionAttempt`-class of a problem type a database schema is generated.

The other relevant package is `com.automatatutor.snippet`. For every problem it contains a class that has to extend `ProblemSnippet`, so it has to implement the methods `renderCreate`, `renderEdit`, `renderSolve` and `onDelete`. The first three methods are used to render html-content, which is embedded into the website when a problem is created, edited or solved. Thereby these methods also have to take the necessary steps when an instructor finishes creating or editing a problem or a user submits a solution. In

particular, the `renderSolve`-method has to call the corresponding grading-method in the backend and handle the returned XML result. That backend call is done using an object called `GraderConnection`, which contains a method for every problem. This method calls the corresponding backend-function with the correct parameters, e.g. the attempt of the user and the correct solution. Furthermore each of the methods `renderCreate`, `renderEdit` and `renderSolve` uses an html-template for creating the html-content, so that most of the html-code is placed there and only the specific values of a problem instance have to be put there. This is done using a mechanism provided by Lift. Of course the html-templates can use javascript-libraries and css-stylesheets.

2.2.2.2 Backend

The backend is written in C#. The project consists of several sub projects, which are partially used by specific problem graders only or for functionality like problem generation. The project `WebServicePDL` contains a class extending `System.Web.Services.WebService`. This class defines all the methods that the backend provides over its API and that are called by the frontend. So in particular every grading-method is stored here, accessing problem specific functionality of the other projects in the solution. Each of the methods in the `WebService`-class takes XML objects as arguments and returns an XML element as respond.

The other two projects that are relevant for this thesis are `AutomataPDL`, where we put all the PDA functionality, and `TestPDL`, which contains the tests for it. `AutomataPDL` already contained the namespace `CFG`, among other things. This namespace has some classes that allow the usage of CFGs and some typical algorithms concerning them [2].

3 Creation of the PDA functionality

3.1 Exercise types

We added the following two problem types to Automata Tutor.

English to PDA In exercises of this type students should create a PDA to a given context free language. When creating a problem of this type, an instructor has to specify the following parameters:

- **CFL:** the context free language which the students should create an automaton for. The instructor can define it in set notation or descriptively in English.
- **Correct PDA:** a correct PDA or DPDA and its acceptance condition, that accepts exactly the defined language. Optimally, the number of states of this PDA and the number of stack symbols is as little as possible for the language. The PDA the students should create must have the same acceptance condition and determinism property as this PDA created by the instructor.
- **Give stack alphabet:** option that specifies whether the stack alphabet defined by the instructor has to be used by the students, too. If *false*, the students can specify their own stack alphabet.
- **Allow simulation before submitting correct solution or failing:** option specifying whether the students should be allowed to run simulations in the PDA before they submit a correct solution or fail (that means, that they use all allowed attempts but do not submit a correct solution). This option should be deactivated especially for problems that are easy to solve, as running simulations simplifies creating a correct solution.

This problem type is the classical one related to PDAs and the most helpful one for thoroughly understanding PDAs. Moreover, exercises of this type can be relatively hard to solve.

Words of PDA When solving an exercise of this type, an immutable PDA is presented to the students, including its properties presented above (which are also immutable, of course). Then the student should enter a given number of words that are in the language accepted by the PDA and a given number of words that are not in this language.

Thus, to create a problem of this type, an instructor has to define the PDA or DPDA and its acceptance condition and the both numbers of words that *are* respectively *are not* in the language of the PDA. He also has to specify whether the students should be allowed to run simulations before submitting a correct solution or failing, like in the problem type before.

Problems of the type described in this paragraph are generally easier to solve than the ones of the type "English to PDA" (3.1). They facilitate the entry into PDAs, as a student only needs to understand and simulate a given PDA (manually) but does not have to build one on his own.

3.2 Realization of the frontend

3.2.1 GUI for building PDAs

For all problem types described above a tool for building and presenting a PDA is needed. This should be as simple and intuitive to use as possible, but also give a clear overview over the PDA, so that it helps students to deeply understand the way of working of PDAs. Therefore, the obvious way of simply defining all components of the 7-tuple in textual form is not sufficient, whereas a graphical presentation of a PDA is preferable.

At first we tried to extend the existing tool for creating DFAs and NFAs, which seemed to be too expensive, as the program was hard to understand and not designed for extendability. So we decided to create a completely new interface for building PDAs. At first we will describe all functions of the PDA construction and presenting tool (in the following referred to as "the GUI") from the user's point of view. After that we will give technical insights and mention other features of the tool, which are only relevant for programmers.

3.2.1.1 User functionality

Overview over sectors of the GUI The GUI is basically structured in two sectors. The first one contains the alphabet, the stack alphabet, the acceptance condition of the PDA and whether the PDA is deterministic. These four settings are referred to as *properties* of a PDA in the following (even if the first two are direct components of the 7-tuple

definition of a PDA and the last two contain additional information). The second sector contains the graphical representation of the PDA with its states and transitions.

Modes of the GUI In principle, the GUI can be in two different modes: in present or construct mode. The present mode, as its name suggests, is for showing an immutable PDA only, so neither the properties nor the states and transitions can be changed. This mode is for example used by the problem type "Words of PDA". The construct mode allows at least the edition the states and transitions of a PDA. For every property of the PDA can be decided separately by the programmer if it should be editable. This is necessary for the problem type "English to PDA", because students solving such exercises should potentially be able to edit the stack alphabet, but they should not be able to change the alphabet, as it is defined by the instructor.

Properties sector The first sector contains a bullet for each *property*. If a property is not editable, its value is simply displayed there (see 3.1). If a property is editable, then an input field is shown, that always contains the current value of the *property* (see 3.2). If the user edits one of the *properties*, then this changes are applied to the second sector with the states and transitions, without resetting them to their initial values.

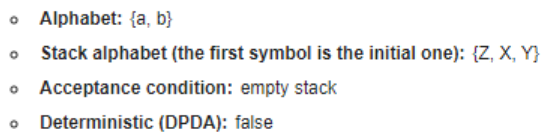
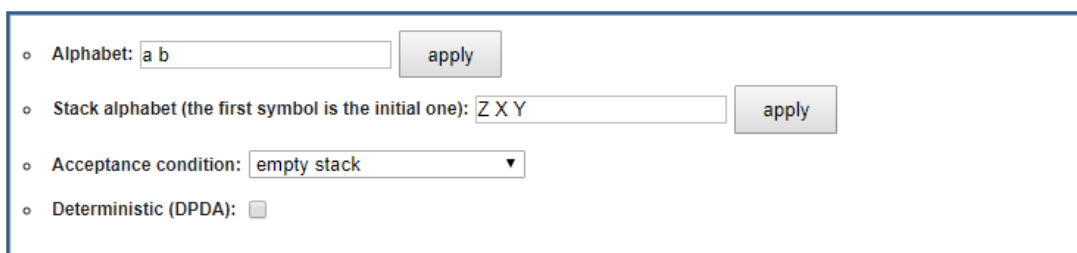
- 
- Alphabet: {a, b}
 - Stack alphabet (the first symbol is the initial one): {Z, X, Y}
 - Acceptance condition: empty stack
 - Deterministic (DPDA): false

Figure 3.1: Properties immutable



- Alphabet:
- Stack alphabet (the first symbol is the initial one):
- Acceptance condition:
- Deterministic (DPDA): ☐

Figure 3.2: Properties editable

States and transitions sector The second sector starts with a collapsible area with a brief explanations for how to construct the states and transitions. They are kept short and focus on the most important things. After that the field for editing the states and transitions is displayed. It has a height of 540 pixel and a width of 670 pixel in its default configuration and contains a state chart, that represents the states as small numbered circles and the transitions as labeled arrows.

Constructing states The initial state with number 0 is always shown and cannot be removed. Apart from its number 0, the initial state is indicated by a small unlabeled arrow to it. When clicking somewhere on the area, a new state is created. The states are consecutively numbered (in detail, the lowest available number is used for a new state). A state can be removed using the context menu, which is displayed on right clicking. Also by the context menu or by double clicking on a state, it is marked as final, which is signaled by a small inner circle within the pre-existing outer circle. States can only be marked as final, if the acceptance condition is *final state* or *final state and empty stack*. If the acceptance condition is changed to *empty stack*, all final states are unmarked. States remember if they were final, that means if then switching the acceptance condition to *final state* again, they mark their selves as final again.

Constructing transitions If a user hovers over a state A , then a bigger outer circle appears. In order to create transitions from A to state B , he must start to drag from that outer circle and leave over B . Then a labeled arrow is drawn from A to B . This arrow, in the following called *link*, groups all transitions between the two states. By double clicking on the label of the arrow the user can edit the transitions between A and B . The label then is replaced by a multi line text input field, where the user can insert one transition per line in the *arrow-notation* (see 2.1): $a, Z/\alpha$, where a is the input letter or ϵ , Z the popped stack symbol and α the pushed stack symbol sequence (see figure 3.3). As ϵ is impractical as input key, E is used instead, so that E cannot be used as regular alphabet letter. If nothing should be pushed onto the stack, then α must simply be the empty string. The user can finish editing transitions by clicking at any other point, then the lines of the input field are parsed and displayed in the arrow label. All lines that have not the required format are ignored.

A *link* can be removed using the context menu. It can also be dragged to change the start or end state: if the user drags an arrow on a point that is, for example, closer to the start point of the arrow, then he can change the start state of the *link* to C by dragging the arrow to C and leaving it there.

Furthermore, self *links* or mutual *links* (that are *links* between two states A and B , where also a *link* between B and A exists) are drawn different from the other ones, that are

simply straight lines. A self *link* is shown as a loop and mutual *links* are drawn curvy to avoid overlapping each other. A self *link* is positioned at a specific position at the edge of the circle, so that it does not overlap with other *links* of that state, if possible. The size of the curve of a self *link* and of mutual *links* is determined by the size of the label of the *link*. Thereby, for a self *link* must be ensured that the label does not overlap with the state. For mutual *links* must be guaranteed that their labels do not overlap each other.

A user may add transitions that are invalid regarding to different conditions. Firstly, he can add a transition using alphabet letters or stack symbols, that are not in the alphabet respectively in the stack alphabet of the PDA. Such transitions are marked red. Secondly, if the user creates a DPDA, then transitions may violate the determinism condition. These transitions are marked green (see figure 3.3). In general, every *link* containing invalid transitions has a red arrow.

Layout The layout of the state chart is left to the user, that means he can drag the states around depending on how he wants the PDA to be presented. This is more practical than a force based layout or something similar, because so the states can be organized into specific groups. The positions of the states are also saved with the problem, therefore the desired layout is kept.

Simulation Finally, a user can run a PDA-simulation of a specific word, that means he can step through all configurations of a path of a word and display each configuration. The simulation runs as follows: after entering a word α to the text field and clicking the button to start the simulation, the backend calculates a path of the word in the pda. If the path is empty, then a message is displayed that the word is not accepted and the simulation is stopped. Otherwise, the actual simulation is started. Under the field with the state chart the simulation control is shown (see figure 3.4). It consists of four buttons for moving to the begin, to the end, one step back and one step forward in the simulation and one button for ending the simulation. In each step the current configuration is displayed like this: the state of the configuration is marked in a special color, the remaining word is presented in the bottom of the state chart and the current stack is shown next to the state chart, where the latest pushed stack symbols are marked. It should be mentioned, that the button for stepping forward changes the configuration actually in two sub-steps: with the first click on the button the transition leading to the next configuration is marked, as well as the top most symbol on the stack and the first letter of the remaining word, if the transition is no ϵ -transition (see 3.4). Then only after the next click the new configuration is displayed as described. During the simulation it is not possible to edit the pda, neither its properties nor its

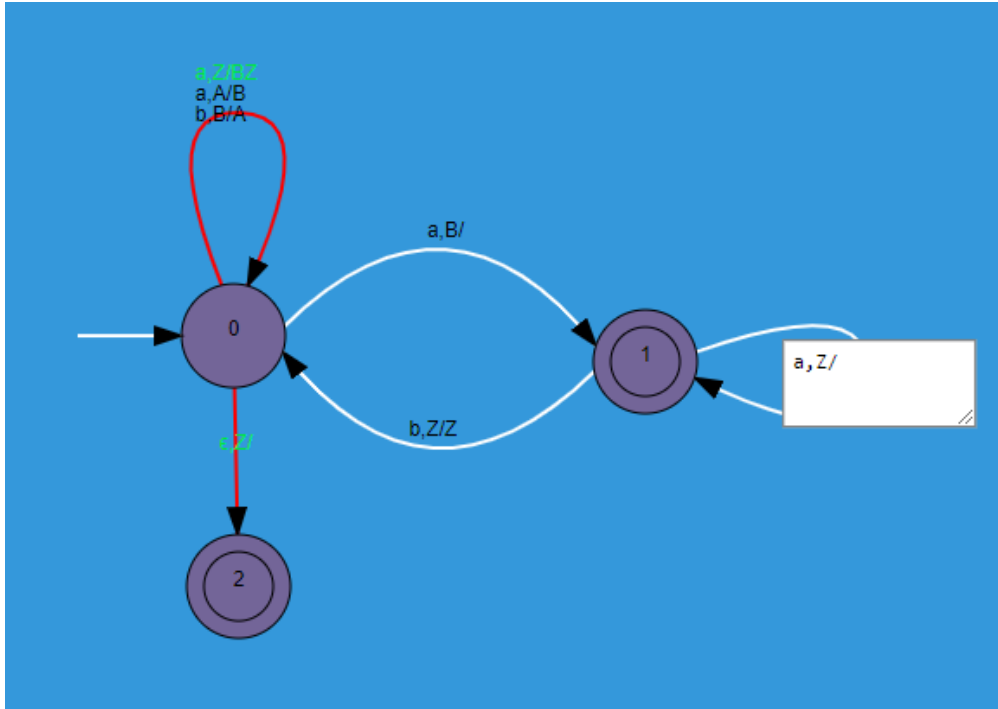


Figure 3.3: The states and transitions sector shows a DPDA with a determinism warning in green. The state 0 is the initial state, the states 1 and 2 are final. The self transition of 1 is edited at the moment, when clicking somewhere else the transition is parsed and displayed in the label: it reads a , pops Z from the stack and pushes nothing back.

states and transitions.

As explained above (see 3.1), running a simulation is not always allowed: an instructor can always run a simulation during creating or editing a problem, but a student can only do that either when the instructor explicitly allowed it, when he submitted a correct solution or when he used all possible attempts but did not solve the exercise.

3.2.1.2 Technical view

Now we want to explain how we developed the GUI described in the last section.

Language and technology For developing the GUI we used Javascript, version ES7. As the tool should easily be embedded in a website and has to interact with a graphical user interface, Javascript was appropriate [7]. For creating such graphical interfaces the

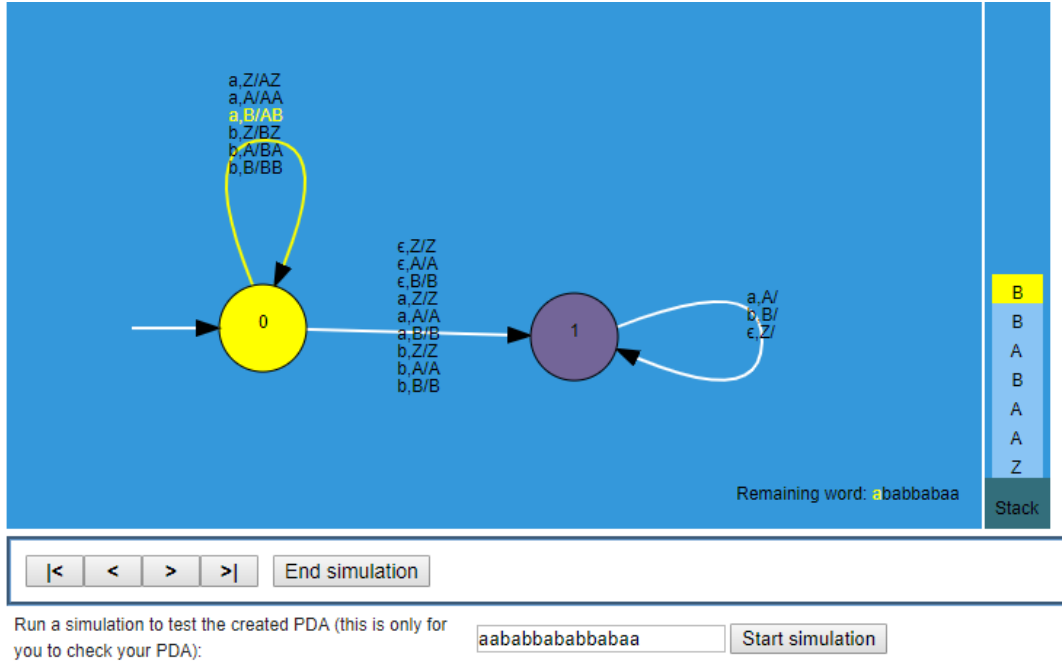


Figure 3.4: Snapshot of a simulation: on the right the stack of current configuration is displayed, in the bottom right the remaining word is shown. The state of the current configuration is 0. At the moment a transition is entered, which is marked yellow. As this transition reads the letter a and the stack symbol B , these both are also marked yellow.

vector graphic language svg is the best choice in a web environment, as it is easy to create and can be rendered very fast by most common browsers. We also used "normal" html, especially for the section with the PDA properties, and css for styling the html and svg elements. In the Javascript program the svg elements of the interface are often accessed and manipulated. Therefore, we used the Javascript-library d3.js besides the browser build in techniques, as d3.js is very useful for visualizing collections of data [8].

We would like to mention briefly, that ES6 and ES7 have some significant advantages over older versions. The introduction of classes is a great gain and was intensively used for the implementation of this GUI. Furthermore, through the possibility of importing Javascript-functionality of other files and exporting own elements, the code can be structured into logical components much better. These features are very helpful for creating a clear and understandable architecture.

Finally, the Javascript code of the different files need to be bundled into one file, so that

this file can be loaded by the browser. For this purpose we used webpack, which cannot only build all Javascript files into one, considering the import- and export-statements, but also includes imported css-files, which are applied in the browser by the bundled file [9].

Separation of model and view One major goal of the software design was a strict separation of the model and view of a PDA. This was achieved by several View-classes, of which the respective model-classes contain objects (see figure 3.5). Whenever a model object changes a property, that is displayed to the user, it updates its view.

Observer pattern Furthermore, the observer pattern was often used in the software design at different points, not only for handling user interaction. The observer pattern allows an object to precisely notify other objects about changes of own properties, which they depend on. Compared to global update routines this pattern can increase the reactivity and performance of a GUI, as only "interested" objects are informed about changes immediately. The pattern can also help to clarify dependencies between classes by defining uniform interfaces for them, especially when the participated classes have no "natural relationship", as for example a class and its view do. Finally, the observer pattern leads to a reduction of the dependencies to the essential ones, as the access of arbitrary other objects is prevented.

The observer pattern was instantiated in the GUI as follows (see also figure 3.5):

- **Creating an observer class:** An observable class *C* has to define its so call `ListenerInterface`, which is simply a Javascript object with all event names, that can arise on *C*. Now *C* creates a property of the class `ListenersSet`, which takes the `ListenerInterface` as argument, and is used for storing the listeners of *C*.
- **Creating an observable class:** An observer class *D* of *C* has to create an instance of the class `Listener`, that needs the `ListenerInterface` of *C* as parameter as well. Then for each event defined in the `ListenerInterface`, *D* can add a function to the `Listener`-object. In this process only those events need be considered, that *D* wants to listen for.
- **Registering an observer:** *D* or a controller class must add the `Listener` of *D* to the `ListenersSet` of *C*. The `ListenersSet` checks if the `Listener` has the correct `ListenerInterface`.
- **Notifying observers:** when one of the events arises in *C*, *C* can just call the method `callForAll` on the `ListenersSet` with the event name and any other

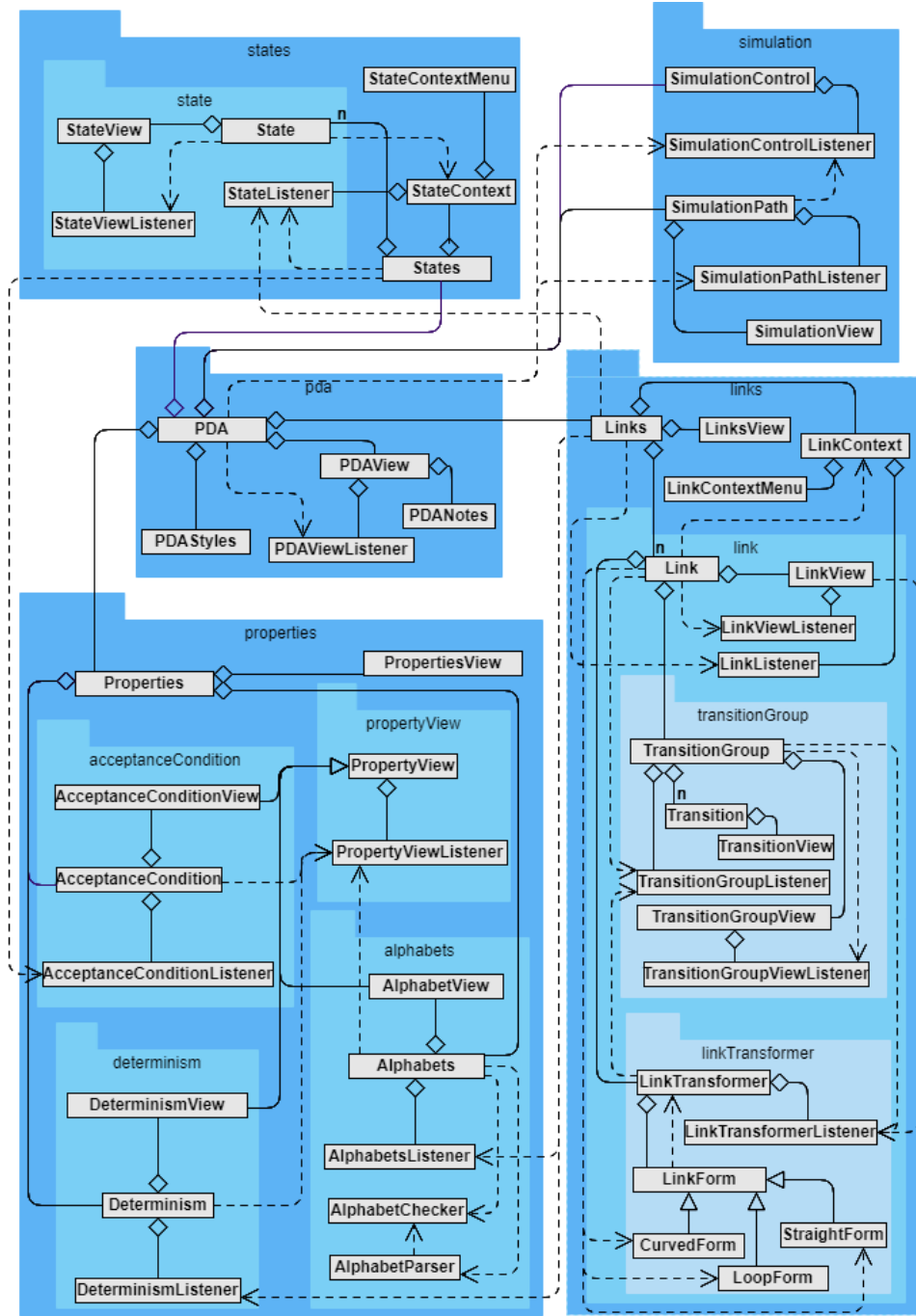


Figure 3.5: Class diagram of the most important components in the GUI. The packages stand for directories and the classes are realized as ES6-classes or some of them only as own files. All aggregation dependencies without a defined multiplicity have 1 to 1

relevant values as arguments. This method then invokes the corresponding function with the additional arguments for each of the `Listeners` in the set.

A lot of the classes in the diagram have corresponding listener-components, which they have an aggregation dependency to and which makes them observable. The listener-components are no classes, but files with the `ListenerInterface`-instance and a function to create the `ListenersSet`. All observers of a certain class have simple dependencies (represented as dashed arrows) to the listener-component of that class, more precisely to the `ListenerInterface`.

The components in detail The project is structured into five directories, which are shown in figure 3.5 and explained in the following.

Pda This directory contains the class `PDA` representing the whole PDA. `PDA` provides methods to export the PDA to XML and to create one out of a given XML-string. Apart from the constructor and the XML-import there are several static methods for creating PDAs in specific use cases, as for example build a new PDA with default properties or build an immutable PDA out of an XML-string (used for solving a problem of the type "Words of PDA"). Furthermore, the PDA has methods for switching between the present and construct mode. The method `startSimulation` takes a path of a word (see 2.1) in XML and starts the simulation.

Properties This directory contains a sub directory for each property, with the property-class itself, its listener-file and its view for displaying the property or enabling to edit it. The views of the properties have a common super class and use the same `ListenerInterface`, as the only event arising in a property-view is `onChanged`, fired simply when the user changes the property. Furthermore there is the class `Properties` with its view. This class stores all properties at one place and is accessed by the PDA.

States This folder includes the class `States`, which is used by the PDA. This class contains all States of the PDA and a `StateContext` object, that contains common used objects and values of all states, as for example the radius of a circle or the `StateContextMenu`. So every instance of `State` in `States` has access to the `StateContext` in `States`. The `StateContext` contains also the `ListenersSet` of the `States`, as all `States` share their listeners. Each `State` has its own view for drawing the circle and allowing the user to interact with it, as for example dragging it. The class `States` listens for the `AcceptanceCondition`, as it determines whether a `State` can be marked as final.

Links The transitions of the PDA are managed by the class `Links`. It includes a `LinkContext`, which has the same function as the `StateContext`, and a collection of `Links`. All `Links` have the same listeners, so that they are also stored in the `LinkContext`. A `Link` has a `LinkView` for displaying the arrow and handling user interaction with it. Moreover, a `Link` contains a `TransitionGroup`, that sums up all `Transitions`. It has an own view for displaying the label on the arrow-line or making them editable on double click. To be able to change its color when one of the `Transitions` is invalid (see 3.2.1.1), a `Link` listens to its `TransitionGroup`, which fires an event on changing its `Transitions`. For drawing the arrow line in the correct form, a `Link` stores a `LinkTransformer`. This class has a property `path`, which defines the `svg-path` of the line of the `LinkView`. The `svg-path` is a special `svg-attribute` specifying the exact form of a `svg-line`. So the `LinkView` listens to the `LinkTransformer` to be notified when the `path` property changes. The `LinkTransformer` contains a `LinkForm`, which can be either a `StraightForm`, `CurvedForm` or a `LoopForm`. The `LinkForms` are the actual creators of the `svg-path`, which they hand over to the `LinkTransformer`. The `LinkForm` of the `LinkTransformer` can be changed by the `Link`, which is done when for example the `Link` is dragged by the user to a self link. For calculating the new `svg-path`, they must consider the positions of the start and end state of the `Link`. The `LoopForm` and `CurvedForm` must also consider the size of the label on the arrow (see 3.2.1.1). That is the reason for the `LinkTransformer` listening to the `TransitionGroup`.

Simulation This directory contains everything which is necessary for running a simulation. The `SimulationControl` creates the control buttons beneath the PDA, the `SimulationPath` parses the given path of the word in XML, controls the whole simulation and uses the `SimulationView` to show the stack and the remaining word of the current configuration. It listens for the button clicks of the `SimulationControl`. The PDA contains the `SimulationControl` and creates a `SimulationPath` when a simulation is started. Furthermore it listens to the `SimulationControl` and `SimulationPath`, to be informed when the simulation is ended and when the current configuration changes in order to mark the state of the current configuration.

Realization of the context menus The `StateContextMenu` and the `LinkContextMenu` are actually no own classes, but instances of the class `ContextMenu`, which is omitted in the diagram. A `ContextMenu` instance can be created out of given items. Thereby, an item is a pair of a key and the string to display in the context menu. The `ContextMenu` then creates a `ListenerInterface` with the keys of the items as events. When for example a `State` wants to display the `StateContextMenu`, it has to call the method `show` on the `ContextMenu`, which takes a `Listener` with the `ListenerInterface` of the

ContextMenu as argument. Then the context menu is drawn at the click position. When the user clicks at an item of the ContextMenu, it notifies the given Listener and hides itself.

3.2.2 Creation of the new exercise types

To install the new problem types (see 3.1) into the frontend, we created the necessary scala-classes (see 2.2.2.1).

Model classes At first, we added the model-class for each problem type to the package `com.automatatutor.model`. The PDA, that is needed in every problem type, is stored there in XML. Then we added the `SolutionAttempt`-class for every problem type.

Snippet classes The `ProblemSnippet`-classes in the package `com.automatatutor.snippet` for rendering the html-content of the web pages were bit more of work. For the methods `renderCreate`, `renderEdit` and `renderSolve` of each snippet we created corresponding html template files, which the methods use for rendering. This is done by inserting specific values into the pages, as for example in the case of `renderEdit` the parameters of the existing problem and the XML string of the PDA, of which the PDA-GUI builds the PDA (see 3.2.1.2). The html contents generated by the `renderCreate` and `renderEdit` methods include the script of the PDA-GUI and have form input fields for all other parameters of the problem type. The html content of `renderSolve` contains the PDA-GUI script, too, and appropriate form input fields for the solution of the students. Therefore, the three pages of each problem type have html forms, which are submitted when the user finishes creating, editing or solving the problem. Then the corresponding method in the scala-snippet has to handle the submitted values, in order to create a new problem, change an existing one or to call the backend for getting the grade and feedback for a student's solution. Thereby the XML-export of the PDA-GUI is used for storing the created PDA respectively handing the created PDA over to the backend in the case of solving a problem.

Simulation On every of the three pages also a text input field and a button is displayed beneath the PDA-GUI, which is used for entering a word to run a simulation of it in the PDA (see figure 3.4). On the solve-pages the button is disabled, if the student is not permitted to run a simulation (see 3.1). For creating these both html elements we created the scala class `PDASimulationRunner`, which is used by the snippets. When the user clicks on the start-button, this class checks again if the user is permitted to run a simulation and then calls the backend for calculating a path of the word. After that, the

startSimulation-method of the PDA of the GUI is invoked with the XML result of the backend call (see 3.2.1.2).

3.3 Realization of the backend

The `WebService`-class in the sub project `WebServicePDL` contains all the methods of the public API of the backend, that take XML objects as input and also return an XML object (see 2.2.2.2). So we placed there the three necessary methods for the PDA functionality. The class `System.Xml.Linq.XElement` of the .NET framework is used for working with XML documents.

Listing 3.1: Methods added to the *WebService*-class

```
public XElement ComputeFeedbackPDAPWordProblem(XElement xmlPda, XElement
    xmlWordsInLanguage, XElement xmlWordsNotInLanguage, XElement
    xmlMaxGrade)
{
    return WordProblemGrader.GradeWordProblem(xmlPda, xmlWordsInLanguage,
        xmlWordsNotInLanguage, xmlMaxGrade);
}

public XElement ComputeFeedbackPDAConstruction(XElement xmlPdaCorrect,
    XElement xmlPdaAttempt, XElement xmlGiveStackAlphabet, XElement
    xmlMaxGrade)
{
    return ConstructionProblemGrader.GradeConstructionProblem(
        xmlPdaCorrect, xmlPdaAttempt, xmlGiveStackAlphabet, xmlMaxGrade);
}

public XElement SimulateWordInPDA(XElement xmlPda, XElement xmlWord)
{
    return SimulationAdapter.RunSimulation(xmlPda, xmlWord);
}
```

The first two methods compute the grade and the feedback for a given solution attempt of the problem type "Words of PDA" respectively "English to PDA". The third method computes a path for the given word in the PDA and is used for the simulation. These three methods are the ones, that are called by the scala-classes we added to the frontend, and whose XML results are handled there (see 3.2.2). As one will have

noticed, the method do not contain the actual functionality, but only delegate to the corresponding grader-classes respectively to the [SimulationAdapter](#) in the last case.

3.3.1 Software architecture

The PDA functionality was added to the directory [PDA](#) in the sub project AutomataPDL. Furthermore, we created some tests in the sub project TestPDL.

The most relevant classes are presented in figure 3.6. Static classes have a dashed border, and the interfaces of a class are noted directly above it. We created the following namespaces with the described classes:

AutomataPDL.PDA.Graders This namespace contains the both static grader classes [ConstructionProblemGrader](#) and [WordProblemGrader](#), whose methods are called in the code snippet above (see 3.1). These classes contain the grading algorithm and use the other classes for calculating the grade and for generating the feedback.

AutomataPDL.PDA.PDA The namespace AutomataPDL.[PDA](#).[PDA](#) includes classes for modeling a PDA. A lot of the classes handling a PDA expect two generic type definitions, of which [A](#) determines the data type of the alphabet letters and [S](#) the type of the stack symbols.

The class [PDA](#) stands for the whole PDA, including an [AcceptanceCondition](#) and a property specifying if the PDA is deterministic. A [PDA](#) contains [States](#), which have a unique id. The [Transitions](#) of the PDA are directly stored at the [States](#), more precisely a [State](#) contains all [Transitions](#) having this [State](#) as start state. A [Transition](#) includes the property `SymbolIn` of the generic type [Symbol](#), which stands for the input letter or ϵ . The class [PDA](#) has the static method `FromXml`, which creates a [PDA](#) instance out of a given XML element. The actual parsing of the XML element is delegated to the static class [PDAXmlParser](#). The class [PDAAEqualityResult](#) is used for running an approximatively equality check of two PDAs and for storing the result of the check.

The [InconsistentPDAException](#) is thrown at several points in the code, whenever the situation occurs that a [PDA](#) has acceptance condition *final state and empty stack*, but accepts a specific word with final state and not with empty stack, or the other way around.

AutomataPDL.PDA.PDARunner The [IPDARunner](#) defines the interface of a class, that solves the word problem for a given word in a PDA (see 3.3.2). The classes [PDARunnerWithCFG](#) and [DPDARunner](#) implement this interface. The first one works for any PDA and combines the classes [PDATransformer](#), [PDAToCFGConverter](#) and

3 Creation of the PDA functionality

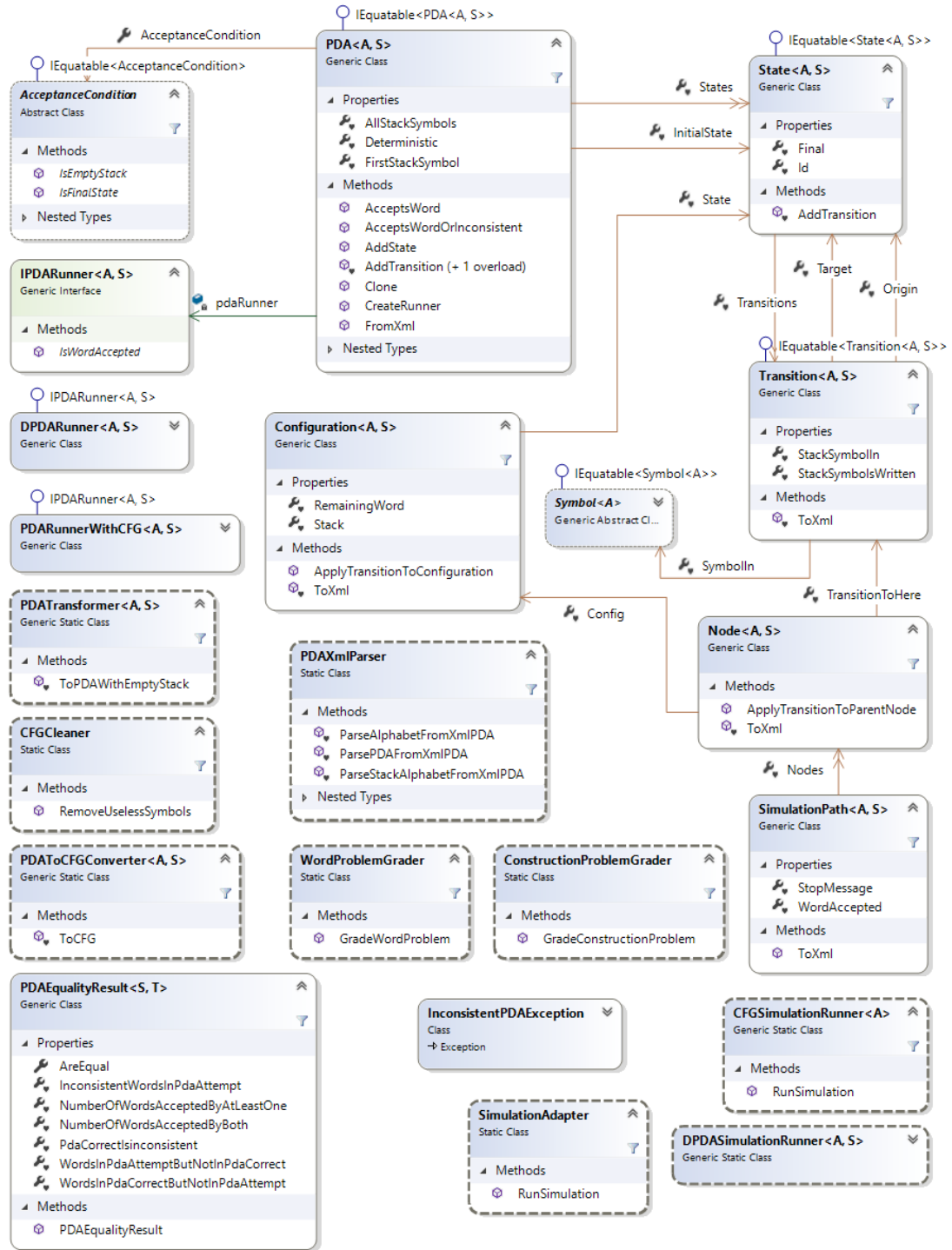


Figure 3.6: Class diagram of the PDA-backend

`CFGCleaner`, which contain important algorithms. The second one works for a DPDA only. A `PDA`, in turn, uses the `PDARunnerWithCFG` and `DPDARunner` for its method `AcceptsWord` and `AcceptsWordOrInconsistent`.

AutomataPDL.PDA.Simulation This namespace contains the classes `SimulationAdapter`, `CFGSimulationRunner`, `DPDASimulationRunner`, `SimulationPath` and `Node`, which are used for computing a path for a word in a PDA (see 2.1).

3.3.2 Word problem

Both graders need to solve the word problem for PDAs. This means to decide whether a word is in the language a PDA accepts or not. Solving it is not as trivial as it seems: one could guess that a simple search algorithm on the PDA is sufficient, starting with the initial state q_0 and walking along all accessible transitions at every step, until the acceptance condition is reached or no transition can be entered. This could be done using a breadth first search (BFS) or a depth first search (DFS). Actually, the word problem is semi-decidable with this approach, that means if a word belongs to the language of the PDA, the algorithm stops with the correct answer, but if the word does not, the program may not terminate. This comes from possible endless loops during the search algorithm, which we show with a simple example. Assume a PDA M with $\Gamma = \{Z\}$ and arbitrary alphabet Σ . Its state and transition are as displayed in figure 3.7. The language $L_\epsilon(M)$ is empty. For any word, a BFS or DFS in the PDA does not terminate, as the only transition can be entered in any step again.

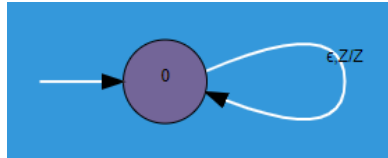


Figure 3.7: PDA with possible endless loops

The `PDA` provides the method `AcceptsWord`, which checks if a word is in the language of the PDA. If the `PDA` is deterministic, this is done using the `DPDARunner` (see 3.3.2.3), otherwise the `PDARunnerWithCFG` (see 3.3.2.2) is used.

3.3.2.1 Naive approach

At first we tried to implement a breadth first search on the PDA, that notices when an endless loop is reached and stops then there. However, in spite of testing various strategies, that seemed not possible, in particular not within an appropriate runtime. In

one of our approaches we tried to recognize specific schemata of configurations and visited states, which did not lead to a working algorithm. After that, we introduced a global counter for all branches of the BFS-tree, so that a branch was not pursued when this counter was zero. The counter was always reset in all branches to the total number of transitions in the PDA plus one, when in any branch a transition was entered for its first time or when a new remaining word with minimum length so far was reached. The counter was decremented in every branch step, where an ϵ -transition was entered, that pushed at least one symbol to the stack. Furthermore, if in a certain branch a new remaining word with minimum length in that branch was reached, the counter was reset there. Even if the algorithm has a too high complexity for a practical usage, it seemed to work in our tests, but we did not make it to prove the algorithm. It is easy to see that the algorithm always terminates, but that it does not reject a word in the language of the PDA, is not obvious. Especially because of the too high complexity (exponential in the maximum number of passable transitions of a state) we did not continue with this approach.

3.3.2.2 Implemented approach

Overview The classical way of deciding the word problem for a PDA $M = (Q, \Sigma, \Gamma, q_0, Z_0, \delta, F)$ is described now [6]. For a CFG in Chomsky-normal-form (CNF) the Cocke-Younger-Kasami (CYK) algorithm can solve the word problem. Therefore, in order to be able to apply this algorithm, M must be transformed into a CFG G with the same language. However, the algorithm for converting a PDA into a CFG works only for the acceptance condition *empty stack*, so that a transformation of M to a PDA N with $L_F(M) = L_\epsilon(N)$ is necessary, if $L_F(M)$ is examined. As this algorithm often produces a lot of non useful symbols, they should be removed for increasing the performance. Then G is converted into an equivalent CFG G' in CNF, on which the CYK algorithm can be executed. Summarizing, the following steps have to be done for deciding the word problem for the word α in M :

1. If $L_F(M)$ is considered, convert M into a language-equivalent PDA N with acceptance condition *empty stack*; otherwise set $N := M$
2. Create a CFG G from N with the same language
3. Reduce G to useful symbols only
4. Create a language-equivalent CFG G' in CNF from G
5. Run the CYK-algorithm on G' with α

These necessary five steps are executed in the class `PDARunnerWithCFG`, with usage of the other classes mentioned in the next paragraph. When instantiating a `PDARunnerWithCFG` for a certain PDA, the steps 1-4 are run immediately in the constructor and the resulting CFG G' in CNF is stored as a property, to not have to repeat this for every word. When calling the method `AcceptanceResult<S> PDARunnerWithCFG<A, S>.IsWordAccepted(string word)`, the CYK algorithm (step 5) is run for the word. If the M has the acceptance condition *final state and empty stack*, then the `PDARunnerWithCFG` creates two CFGs G_ϵ and G_F and their equivalent in CNF, one out of M itself and one out of N , which arises from the conversion to the *empty stack*-acceptance condition. The `IsWordAccepted`-method runs for the given word the CYK algorithm with both CFGs, and each result is stored separately in the returned `AcceptanceResult`-object. So it can be checked if both match.

The 5 steps in detail

Step 1 As mentioned above, this algorithm is implemented in the method `PDA<A, S> PDATransformer<A, S>.ToPDWithEmptyStack(PDA<A, S> pda)`. Assume, that $L_F(M)$ is considered. The method creates the new PDA $N = (Q \cup \{q'\}, \Sigma, \Gamma \cup \{Z'_0\}, q_0, Z'_0, \delta', F)$, so that $Z'_0 \notin \Gamma$ and $q' \notin Q$. The new stack symbol Z'_0 is the initial stack symbol of N and prevents the stack from getting empty before a final state is reached. The added state q' is used for emptying the stack when N reaches a final state. These both changes ensure, that N can reach an empty stack with a certain word if and only if M can reach a final state with this word. To implement the described changes, δ has to be extended to δ' with the following transitions: Firstly, $q_0 \xrightarrow{\epsilon, Z'_0 / Z_0 Z'_0} q_0$ has to be added, to simulate the initial configuration of M . Secondly, for every final state q_f the transitions $\{q_f \xrightarrow{\epsilon, X /} q' \mid X \in \Gamma \cup \{Z'_0\}\}$ are added, so that the PDA can transit from a final state to the new state, independent of the current top most stack symbol. Thirdly, for every stack symbol $X \in \Gamma \cup \{Z'_0\}$ the transition $q' \xrightarrow{\epsilon, X /} q'$ is created, which lead to an empty stack when reaching q' .

As mentioned above, $N := M$ if $L_\epsilon(M)$ should be considered.

Step 2 The conversion of N to G with the same language is realized in the method `ContextFreeGrammar PDAToCFGConverter<A, S>.ToCFG(PDA<A, S> pda)`. The class `ContextFreeGrammar` already existed in the namespace `AutomataPDL.CFG`. The algorithm works as follows: The CFG is constructed as $G = (V, \Sigma, P, S)$, where $V = Q \times \Gamma \times Q \cup \{S\}$. Such a nonterminal is written as $[q, X, q']$. For creating the productions, these steps are executed for every state $q \in Q$:

- Add the production $S \rightarrow [q_0, Z_0, q]$
- for every transition $q \xrightarrow{a, Z/\alpha} q'$ (see 2.1), do the following:
 - if $|\alpha| = 0$, then add the production $[q, Z, q'] \rightarrow a$
 - if $|\alpha| > 0$, let $\alpha = Z_1 \dots Z_k$ and add the following set of productions:

$$\{ [q, Z, r_k] \rightarrow a[q', Z_1, r_1][r_1, Z_2, r_2] \dots [r_{k-1}, Z_k, r_k] \mid (r_1, \dots, r_k) \in Q^k \}$$

The arising CFG G accepts the same language as the N with empty stack: $L(G) = L_\epsilon(M)$. The basic principle of the algorithm is the following fact: $[q, Z, r_k] \rightarrow_G \dots \rightarrow_G \alpha$ if and only if $(q, \alpha, Z) \rightarrow_M^* (r_k, \epsilon, \epsilon)$. For a full proof of the correctness of this algorithm, see [6].

Step 3 Reducing G to useful symbols only, is done in the static class `CFGCleaner` in the method `RemoveUselessSymbols`. The not producing symbols have to be removed first, and after that the not reachable ones are deleted (see 2.1). The set T of producing symbols is determined like this: all $a \in \Sigma$ are added initially to T , as they are producing. After that, all left-hand-side symbols of those productions are added iteratively to T , whose right hand side symbols are all already in T . Then G is restricted to these producing symbols by removing all others together with all productions where they occur. The algorithm for now getting all reachable symbols is very similar, it just starts with $\{S\}$ as initial set and works top down.

Step 4 and 5 As the conversion of G into another CFG G' in CNF with the same language was already implemented in the class `GrammarUtilities` in the namespace `AutomataPDL.CFG`, so we needed not do this ourselves. The same holds for the application of the CYK algorithm, so these both steps are not explained in detail here.

Complexity of the algorithm Now we want to determine the complexity of deciding the word problem for the word α in the PDA M .

At first, we introduce two definitions, that are used below:

- The length of the longest stack sequence pushed by a transition in PDA M :

$$s(M) := \max \left\{ |\beta| \mid (q, \beta) \in \bigcup_{p \in Q, a \in \Sigma \cup \{\epsilon\}, X \in \Gamma} \delta(p, a, X) \right\}$$

- the maximum number of transitions of a state of PDA M

$$t(M) := \max \{ |\delta(p, a, X)| \mid p \in Q, a \in \Sigma \cup \{\epsilon\}, X \in \Gamma \}$$

As the steps 1-4 are only run once and not for every word, it makes sense to split the examination of the complexity into two parts: step 1-4 and step 5. The steps 1-4 produce a result each, whose complexity directly determines the one of step 5, so they have to be specified, too.

1. Adding the new transitions is the only non constant part of this algorithm: its complexity is $\mathcal{O}(|F| * |\Gamma \cup \{Z_0\}| + |\Gamma \cup \{Z_0\}|)$, as for every final state and stack symbol a transition is added, as well as for every stack symbol a self transition is added to q' . That is simply $\mathcal{O}(|F| * |\Gamma|)$ (provided that M has at least one final state). The size of the arising PDA with empty stack is omitted here for convenience and because the change is negligible, so that for the complexity of the following steps M is considered instead of N .
2. Creating V of the CFG G is obviously in $\mathcal{O}(|Q| * |\Gamma| * |Q|)$. Transforming the transitions into productions has complexity $\mathcal{O}(|Q| * t(M) * |Q|^{s(M)}) = \mathcal{O}(|Q|^{s(M)+1} * t(M))$, because for every state ($|Q|$) for every transition from there (at most $t(M)$) at most $|Q|^{s(M)}$ productions are created. Therefore, if $s(M)$ is big and M has more than one state, the computation of G can become a very long process and G itself very big, too: $|V| = |Q| * |\Gamma| * |Q| = |Q|^2 * |\Gamma|$ and $|P|$ is in $\mathcal{O}(|Q|^{s(M)+1} * t(M))$. This is the reason we introduced the next step of reducing G , as running the CYK-algorithm is often greatly simplified and much faster so.
3. Determining all producing symbols is in $\mathcal{O}(|P|^2 * s(M))$, as for every production the right hand side symbols (which are at most $s(M)$) have to be checked, and this procedure has to be repeated at most $|P|$ times, because after that no new producing symbol can be reached. For the reachable symbols, $\mathcal{O}(|P| * s(M))$ is necessary, because when starting with S every production has to be visited only once. So in total we get $\mathcal{O}(|P|^2 * s(M))$. The check whether a symbol is included in a set is done in $\mathcal{O}(1)$ in both algorithms, as we used a hash set in the implementation.
4. Transforming G into G' in CNF is in $\mathcal{O}(|P| * s(M) + |P|^3)$, where the $|P| * s(M)$ -part is for restricting all productions to at most two symbols on the right hand side. The $|P|^3$ summand is for eliminating ϵ -productions and chain productions. The complexity of $G' = (V', \Sigma, P', S)$ is then as follows: $|V'| = \mathcal{O}(|V| + |P| * s(M))$, as for every production at most $s(M)$ nonterminals are added (because the right hand side of a production in G has a length of at most $s(M)$). With the same argument $|P'| = \mathcal{O}(|P| * s(M))$.
5. For the CFG $G' = (V', \Sigma, P', S)$ in CNF the CYK algorithm has complexity $\mathcal{O}(|\alpha|^3 * |P'| * |V'|^2)$ for the word α , when implemented inefficient ([6]). However, the

existing implementation has a complexity of $\mathcal{O}(|\alpha|^3 * |V'|^2)$, as for getting all productions with two certain symbols on the right hand side a hash map is used. When the complexity of the component V' from above is inserted there, then we get a complexity of $\mathcal{O}(|\alpha|^3 * (|V| + |P| * s(M))^2)$. Replacing $|P|$ and $|V|$ by the corresponding terms of the PDA, $\mathcal{O}(|\alpha|^3 * (|Q|^2 * |\Gamma| * s(M) * |Q|^{s(M)+1} * t(M))^2)$ is reached for deciding the word problem for α in the PDA M . Therefore, the complexity of the word problem is cubical in $|\alpha|$, quadratical in $t(M)$ and exponential in $s(M)$ to the base $|Q|$. So in sum, the maximum number of pushed stack symbols of a transition can have a huge impact on the performance of the decision procedure.

Overall, the maximum number of pushed stack symbols of any transition in M is relevant for the runtime of the whole algorithm, what our tests directly reproduced.

3.3.2.3 Word problem for DPDAs

For DPDAs, the complicated steps of the general algorithm (see 3.3.2.2) are not necessary. The word problem can be solved using the `DPDASimulationRunner` (see 3.3.4.3). The `DPDARunner` simply runs a simulation for the given word and return true, if the simulation is finished successfully. When the acceptance condition is *final state and empty stack*, simulations for both are necessary.

3.3.3 Determination of equality

For CFGs in general, equivalence is not decidable, therefore the same holds for PDAs, of course [6]. But the `ConstructionProblemGrader` needs to find out, if the solution attempt of the student is approximatively equal to the PDA created by the instructor. Therefore we implemented in the class `PDAEqualityResult` a comparison of two PDAs (see figure 3.6), which simply generates words out of the alphabet and tests, if both PDAs matches concerning their acceptance of these words. When creating a new `PDAEqualityResult`, one has to specify the `pdaCorrect`, `pdaAttempt`, the *alphabet*, the `maximumWordLengthTested` as well as `maximumNumberOfWords` and `maximumMilliSeconds`. If the comparison lasts longer than specified in this parameter, it is terminated and the result so far is returned, so that the server where the backend runs at is not overloaded by too complex PDAs. The generation of words out of the alphabet is delegated to another class, where all words are created that have at most length `maximumWordLengthTested`. The `PDAEqualityResult` then takes the first `maximumNumberOfWords`. Such a restriction is important for keeping the computation expense realistic.

Then for all these words a function is invoked, using the LINQ-feature `AsParallel()`, which automatically parallelizes the execution of these functions considering the number of processors of the machine. The function then checks for a word, if it is in the `pdaCorrect` and `pdaAttempt`, using the `AcceptsWord`-method of the PDAs. If the word is in only one of the languages, it is added to the collection `WordsInPdaCorrectButNotInPdaAttempt` respectively `WordsInPdaAttemptButNotInPdaCorrect`. As the functions are executed in parallel, we used the class `System.Collections.ConcurrentBag` for the collections. Furthermore, the total number of words accepted by both PDAs and by at least one PDA is stored in the property `NumberOfWordsAcceptedByBoth` respectively `NumberOfWordsAcceptedByAtLeastOne`. For the `pdaAttempt` all words with inconsistent acceptance condition are also collected in the property `InconsistentWordsInPdaAttempt`. This is not necessary for the `pdaCorrect`, as it is assumed to be well defined. If it is not, this is notated in the property `PdaCorrectIsInconsistent`. If the both collections with the not matching words are empty, then `AreEqual` returns true.

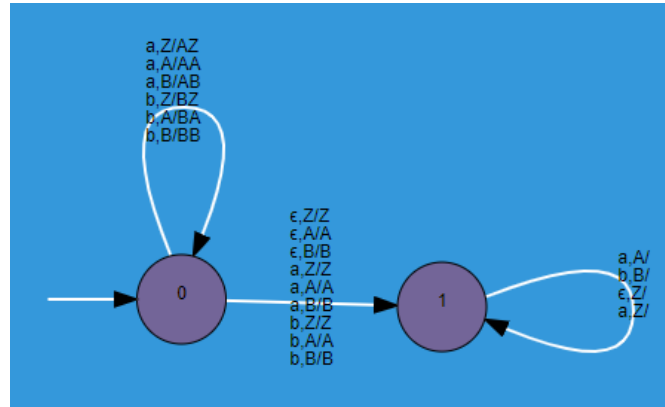


Figure 3.8: PDA for the performance test

We run a performance test for the `PDAEqualityResult` on our machine (CPU i5 6200U 2.3 GHz, RAM 8 GB). As this class uses the `AcceptsWord`-method and therefore the `PDARunnerWithCFG`, its performance depends greatly on this. We tested the following PDA N with the alphabet $\{a, b\}$ on equality with the PDA M , which is basically the same without the transition $1 \xrightarrow{a, Z/} 1$. M contains all symmetrical words and N additionally the symmetrical words followed by an a .

The result is presented in figure 3.9. We run the whole test ten times and calculated the averages to flatten temporary variations. For the maximum length n exist $2^{n+1} - 1$ words, so with an increase of the maximum word length by one in each step, the number of words is almost doubled. Furthermore, as the CYK-algorithm is cubical in

the word length (see 5) and with every step longer words are added, the runtime of the comparison should more than double in every step, which can be seen in the figure.

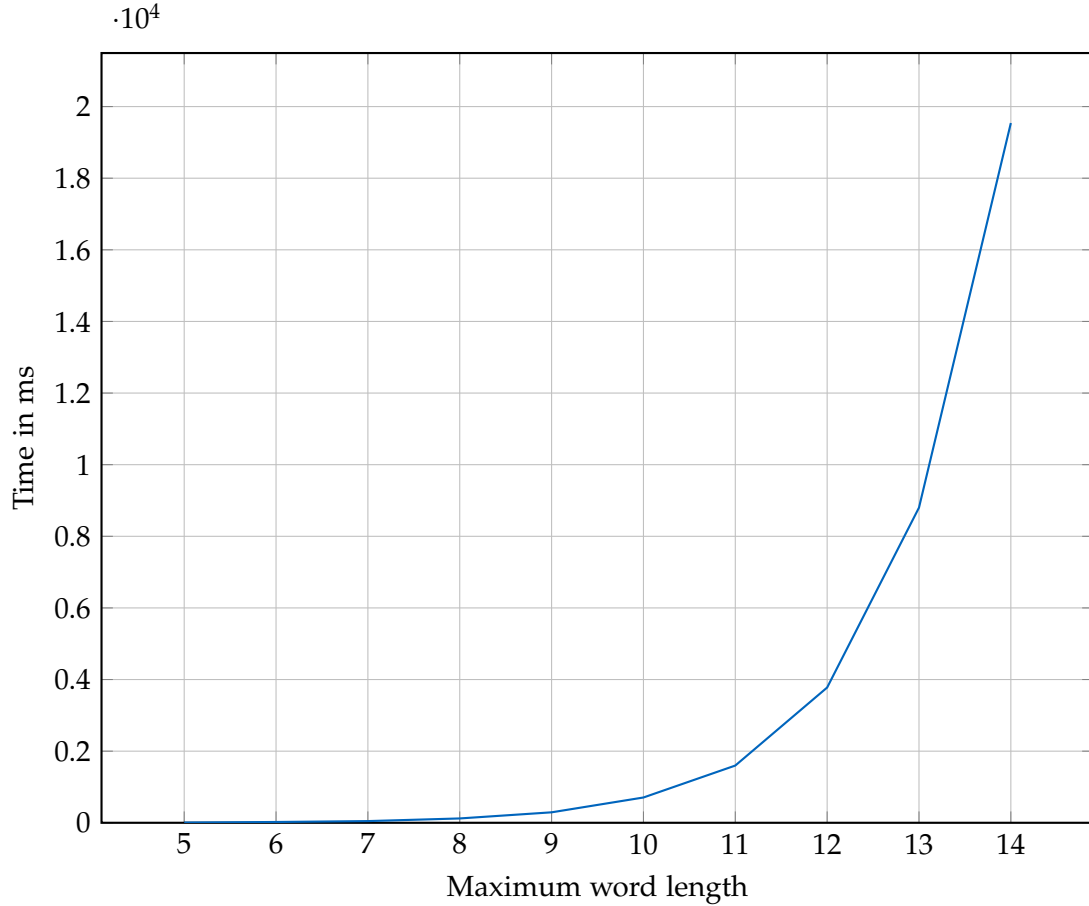


Figure 3.9: Result of the performance test

3.3.4 Simulation

The classes for running a simulation, that means for calculating a path of a word in a PDA, are in the namespace `AutomataPDL.PDA.Simulation` (see figure 3.6). We implemented three different proceeds for that. The first one is the naive approach, which theoretically works for a general PDA, but is actually not used because of a too bad runtime. The second one can be used for a general PDA, too. It is realized in the class `CFGSimulationRunner`. The third approach only works for DPDAs and it implemented in the class `DPDASimulationRunner`. The last two procedures are the ones

that are used by the public API of the backend for the frontend.

The class, that is actually accessed by the public API method, is the `SimulationAdapter` (see 3.1). It tests whether the given PDA is deterministic, and calls depending on that the `DPDASimulationRunner` or the `CFGSimulationRunner`. The methods of these classes return a `SimulationPath`, which includes a sequence of `Node`. Each `Node` contains a `Transition` and a `Configuration`: Even if a path is only defined as a sequence of configurations, the transition, which lead to a certain configuration, is also determined. This is used by the frontend to display the simulation in detail. The `SimulationPath` is then converted to XML by the `SimulationAdapter`, which is sent back to the frontend by the API method.

For a nondeterministic PDA, a path can in general only be computed for words the PDA accepts, as otherwise endless loops may occur in the PDA (see figures 3.7). There are also words, of course, that are not in the language of the PDA, but for which a path could be calculated: This path would simply stop, if for example no further transition can be entered or the stack is empty. But as this only holds in special cases, we allow a simulation only for words the PDA accepts and return an empty path otherwise. For DPDAs, however, a path can always be computed: There endless loops can be determined, so we can always find a end of a path, even if a word is not accepted by the DPDA. For explaining to the user, why the simulation of a word, that is not accepted, stops at a certain steps, the property `StopMessage` of the `SimulationPath` is used. It is considered when the path is exported to XML.

3.3.4.1 Naive approach

At first we tried to directly run a simulation for words the PDA accepts, that means we implemented a breath first search on the PDA in the class `DirectSimulationRunner`. As this is only done for a word the PDA accepts, this algorithm always terminates.

Listing 3.2: Naive approach of computing a simulation path

```
var initialNode = SimulationNode<A, S>.InitialNode(  
    new Configuration<A, S>(pda.InitialState, new Word<A>(word),  
    CurrentStack<S>.WithSingleSymbol(pda.FirstStackSymbol)),  
    pda.AcceptanceCondition);  
  
var frontChain = new List<SimulationNode<A, S>> { initialNode };  
  
while (frontChain.Count() > 0)  
{
```



```

var nodesAcceptedWord = frontChain.Where(node => node.HasAcceptedWord).
    ToList();
if (nodesAcceptedWord.Count() > 0)
{
    return SimulationPathFromFinalNode(nodesAcceptedWord.First());
}

foreach (var node in frontChain)
{
    node.DoStep();
}

frontChain = frontChain.SelectMany(node => node.Children).ToList();
}

```

The class `SimulationNode` represents a certain step in the simulation and contains a `Configuration`, the property `HasAcceptedWord` and its `Children`. The `DoStep()`-method of a `SimulationNode` `n` creates the `Children`-property: this includes all `SimulationNodes`, that arise from applying all transitions, that can be entered in the configuration `c` of `n`, to `c`.

However, this approach does not work sufficiently. Assume a PDA $M = (Q, \Sigma, \Gamma, q_0, Z_0, \delta, F)$ and a word $\alpha \in L_\epsilon(M)$, whose path length is n (see 2.1). The complexity of computing a path is then in $\mathcal{O}(|Q|^n)$, because in every step of exploring new nodes at most $|Q|$ new states can be entered. This holds for the runtime as well as for the size of the resulting data structure. Therefore, for specific PDAs running a simulation like this is not possible for longer words. Of course, there are a lot of PDAs, where this algorithms can be applied.

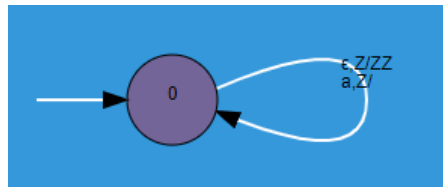


Figure 3.10: PDA with exponential runtime relative to the word length

The PDA in figure 3.10 is defined over the alphabet $\{a\}$ and has *empty stack* as acceptance condition. It accepts all words $\alpha \in \Sigma^*$. In every step for most nodes in the `frontChain` two transitions can be entered. Running a simulation for the word `aaaaaaaaaaaa` lead to a `System.OutOfMemoryException` on our machine. This shows

that the naive approach is not sufficient for computing a path for a word, as it even does not work properly for such a "simple" PDA.

3.3.4.2 Implemented approach

Overview Therefore we developed another algorithm for general PDAs, which is implemented in the `CFGSimulationRunner`-class. It uses a CFG instead of directly the PDA. Assume a PDA M and the word α to simulate. The algorithm works as follows:

1. If M has *final state* acceptance condition, convert M to the equivalent PDA N with *empty stack* as acceptance condition; otherwise, set $N := M$.
2. Create the language equivalent CFG G from N
3. Remove useless symbols from G
4. Convert G to the equivalent CFG G' in 2NF (see 2.1)
5. Calculate a CYK table α in G' and read a derivation d for α out of the CYK-table, if α is accepted. That can be read out of the CYK table. Otherwise stop and return the empty path.
6. Convert the derivation back to a path p in N
7. If M has *final state* as acceptance condition, convert p to the corresponding path in M .

Our first idea was to read a path of a word out of the CYK-table, that arises from applying the CYK-algorithm to the CFG in CNF instead of 2NF, as the conversion to CNF as well as the CYK-algorithm for a CNF already was implemented in the backend. However, this seemed to be too difficult, as the elimination of epsilon and chain productions in CNF makes it hard or maybe even impossible to get a path in the PDA from a derivation there.

The whole procedure is defined in the class `CFGSimulationRunner`, from where the single steps are delegated to other classes. The namespace `AutomataPDL.PDA.CFGUtils` contains further classes, that are relevant for the simulation. They are displayed in figure 3.11.

The steps in detail

Step 1-3 These steps are the same as for solving the word problem (see 3.3.2). So simply the mentioned classes are used for doing these tasks.

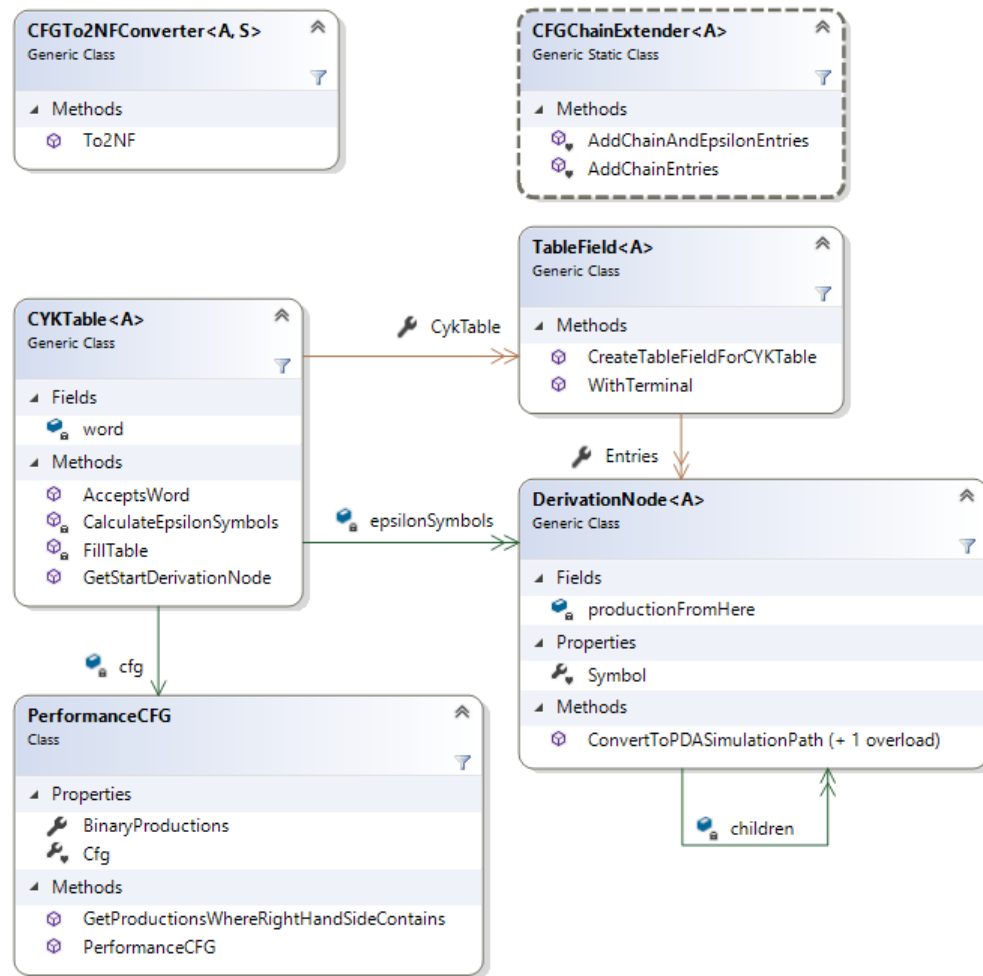


Figure 3.11: Class diagram of the CFGUtils-namespace

Step 4 Converting G to G' in 2NF is implemented in `CFGTo2NFConverter`. To restrict a production $p = A \rightarrow \alpha$ with $|\alpha| > 2$ to at most two symbols on the right hand side, p is removed and simply replaced by the following productions: $A \rightarrow Z_1P_1, P_1 \rightarrow Z_2P_2, \dots, P_{n-2} \rightarrow Z_{n-1}Z_n$ with $\alpha = Z_1\dots Z_n$. This is a sub step of converting a CFG to CNF and produces obviously a CFG with the same language and the desired property.

Step 5 As mentioned above, the CYK-algorithm already implemented in the back-end could not be used at this step, as it only works for CFGs in CNF. So we implemented the CYK-algorithm for a CFG in 2NF in the class `CYKTable`, which is similar to the CYK-algorithm for CNFs (for detailed information about the CYK-algorithm for CNFs, see [6]). The actual CYK-table in this class is a two dimensional array of `TableFields`, which represent a single cell in the table. The cells in the array are indicated with $(row, column)$. The table contains the terminals of a word α in row 0, so that the start symbol is reached in the cell $(|\alpha| - 1, 0)$ (of course only if the word is in the language). A `TableField` contains the property `Entries`. This is a hash map from `int` to `DerivationNode`. A `DerivationNode` does not only contain the corresponding production, but also the property `children` with those `DerivationNodes`, that are reached by the production, that means the descendants in the derivation. Therefore, if the CYK-table is filled completely and the word is accepted, a derivation is also created. As key property for the hash map `Entries` the unique identifier of a grammar symbol is used. So a `TableField` contains for a specific grammar symbol only one `DerivationNode`, that has this grammar symbol as left hand side of its production. That means that not all possible derivations for a word are calculated. This is also not possible in general, as there there can be potentially endless derivations for a word, which is shown for example by the grammar with the productions $\{S \rightarrow A, A \rightarrow A|a\}$.

At first the `CYKTable` computes all so called epsilon symbols, that means all symbols that have a derivation to ϵ . They are stored in a hash map `epsilonSymbols` of `int` to `DerivationNode`: the `int` defines the unique identifier of the symbol and the `DerivationNode` leads to a derivation to ϵ . The calculation of `epsilonSymbols` starts with all epsilon productions (respectively their left hand side symbol) and adds for every production $A \rightarrow \alpha$ the entry $(A.uniqueId, N)$ to `epsilonSymbols`, if α only contains symbols that are already in `epsilonSymbols`. N stands for the created `DerivationNode`, whose children are the `DerivationNodes` of the symbols in α . Furthermore, the entry is only added, if no entry for A exists in the hash map. The hash map `epsilonSymbols` can then be used by the actual CYK-algorithm.

In the CYK-algorithm for a CFG in CNF, the table cells (i, j) have to be filled successively with the corresponding set $p_{i,j}$ of productions, starting with row 0 (see [6]). The same is done in the CYK-algorithm for CFGs in 2NF. But as a CFG in 2NF may contain

epsilon and chain productions, this is not sufficient: for every productions $A \rightarrow \alpha$ an entry $(A.uniqueId, N)$ has to be added iteratively to $p_{i,j}$, where α contains exactly one symbol out of $p_{i,j}$ and either no other symbol or an epsilon symbol. Here the field `epsilonSymbols` of the `CYKTable` is used. N stands for the `DerivationNode` again, that contains as children the corresponding `DerivationNodes` of the symbols in α . As in the calculation of the epsilon symbols, the new entry is only added, if no entry for A exists yet.

Adding the symbols that are reached through chain and epsilon productions to every cell is actually delegated to the class `CFGChainExtender`, which is also used when calculating the epsilon symbols.

Furthermore, the `CYKTable` uses the class `PerformanceCFG`. This provides a method for getting all productions, whose right hand side contains a certain symbol, in a fast way, which is reached by precomputing a hash map. It increases greatly the performance of the CYK-algorithm, as this method is often used when adding the further symbols to each cell.

Finally, the method `GetStartDerivationNode()` returns the `DerivationNode` of the start symbol of the CFG, after filling the table. The method `AcceptsWord` can be used to test if the word is accepted, and to return the empty path, if not.

Step 6 Converting the derivation back to a path in N is done in the method `ConvertToPDASimulation` in the `DerivationNode`. Here, a `DerivationNode` has to replace recursively each nonterminal in its production, that was added through the conversion of the CFG to 2NF, by the right hand side of the productions of the `DerivationNode` of that nonterminal, so that the 2NF is reverted for the own production. The PDA transition, that corresponds to that production, is determined then. That is done by basically reverting the algorithm for creating a CFG out of a PDA (see 3.3.2.2). The resulting transition is then added to the `SimulationPath` so far, together with the `Configuration` bundled in a `Node` (see 3.3.4). After that, the method `ConvertToPDASimulation` is recursively called for each `DerivationNode` in the children-property. Of course, for the first node in the derivation the procedure is a bit different. In sum, this results to a `SimulationPath`.

Step 7 Converting the `SimulationPath` back to a path in the PDA with *final state* is done in the `CFGSimulationRunner` and follows directly from the algorithm for converting a PDA from *final state* to *empty stack* (see 3.3.2.2).

Complexity of the algorithm The complexity of the algorithm is similar to one for solving the word problem. Step 4 of this algorithm is actually faster than step 4 of the

other one, as the elimination of the epsilon and chain productions need not be done. The CYK-algorithm for a CNF, that is implemented in the backend, is in $\mathcal{O}(|\alpha|^3 * |V'|^2)$ (see 5) and a bit faster than for a 2NF. But through a bit different implementation and the intensive usage of hash maps, we reach a complexity of $\mathcal{O}(|\alpha|^2 * (|\alpha| * |P'| + |V'| * |P'|))$ for filling the CYK table: For filling a new cell, the CYK for CNFs iterates over the symbols of the two corresponding other cells and uses a hash map for getting the productions leading to two certain symbols. The factor $|V'|^2$ comes from this. Our CYK-algorithm iterates over the productions instead and checks if the symbols on the right hand side of a production are in the corresponding cells, for which we use a hash map. Adding the symbols, that reach one of the symbols that are already in the set, through chain and epsilon productions, is for every cell in $\mathcal{O}(|V'| * |P'|)$: a cell contains at most $|V'|$ symbols, and for each symbol not more than $|P'|$ productions have to be examined. Getting all productions, where a specific symbols occurs on the right hand side, is in $\mathcal{O}(1)$ because of the usage of the PerformanceCFG.

In sum, the CYK-algorithm for a CFG in 2NF is cubical in the word length and exponential to $s(M)$ with the base $|Q|$, which can be seen through replacing V' and P' by their terms from the steps before, like it is done for the CYK for a CNF (see 5). Compared to the naive approach with its exponential complexity to the word length, this is a great gain. However, a huge $s(M)$ can lead to a long runtime again, so that there can be PDAs, for which the naive approach is faster provided that the word belongs to the language of the PDA. But as this has to be determined in the naive approach at first using a CFG, the second approach has in each case the better complexity.

3.3.4.3 Simulation in a DPDA

For a DPDA, a variant of the naive approach (see 3.3.4.1) achieves a better complexity than the algorithm for a general PDA in 3.3.4.2. It is implemented in the class [DPDASimulationRunner](#).

Algorithm The algorithm works as follows:

Basically, a list of [Nodes](#) is created, where successively new nodes are added. Initially, the list only contains a [Node](#) with no transition and the initial configuration of the DPDA (that means, the configuration with the initial state, the whole word and the initial stack symbol). Let n always be the last node in this list. Then the following has to be done:

1. Test, if n accepts the word. This holds, if $n.\text{Config.RemainingWord}$ is empty and when $n.\text{Config.State}$ is final or $n.\text{Config.Stack}$ is empty (depending

on the acceptance condition of the DPDA). If n accepts the word, return the **SimulationPath**, which is created from the list of **Nodes**.

2. Get the passable transition t . If there is no, then return the current **SimulationPath** with a **StopMessage**, that no transition can be entered.
3. Check if an endless loop was reached. An endless loop means, that the remaining path will be endless and only consisting of ϵ -transitions. An endless loop was reached if a so called repeating circle was passed. A repeating circle is a partial path p with the configuration sequence (c_1, \dots, c_m) , so that $c_i = (q_i, \beta, \gamma_i)$ and the following holds:
 - a) $m \geq 2$
 - b) $q_1 = q_m$
 - c) $|\gamma_m| \geq |\gamma_1|$
 - d) If $k := |\gamma_1| - \min\{|\gamma_i|\} + 1$, then the top most k stack symbols of γ_1 and γ_m are the same. k specifies the number of stack symbols of γ_1 , which are popped during the circle.

As every c_i in the configuration sequence has the same remaining word β , all transitions in the circle are ϵ -transitions.

So if the **list** contains a sub sequence of **Nodes** with configurations, that fulfill the conditions of a repeating circle, and $n.\text{Config}$ is the end of this sub sequence, then a repeating circle was passed. In this case, the **SimulationPath** so far is returned with a corresponding **StopMessage**.

4. Apply the passable transition t to $n.\text{Config}$ and add the resulting **Configuration** together with t as **Node** to the **list**. Go on with step 1 then.

Now we want to prove that an endless loop is reached if and only if such a repeating circle with the definition of 3. was passed.

- In an endless loop a repeating circle is passed (" \rightarrow "): In an endless loop there is at least one state p , which occurs infinite times, as the set of states is final. Furthermore, there is a configuration g in the path, so that after g only ϵ -transitions are entered and for every subsequent configuration $c_1 = (p, \beta, \gamma_1)$ of g the following holds: a configuration $c_2 = (p, \beta, \gamma_2)$ after c_1 in the path exists, so that $|\gamma_2| \geq |\gamma_1|$, because otherwise the stack would become empty at some point. Let $d_1 = (p, \beta, \gamma)$ be a configuration in the path after g . As there is only a finite set of stack sequences with length $|\gamma|$ but an infinite number of configurations with p , d_1 and a configuration d_2 somewhere after d_1 in the path can be chosen, so

that $d_2 = (p, \beta, \gamma\delta)$. As the PDA is deterministic, starting from d_2 exactly the same sequence of transitions is entered as between d_1 and d_2 . From this fact, the condition d) follows directly, as for the circle between d_1 and d_2 , $k \leq |\gamma|$.

- A repeating circle leads to an endless loop (" \leftarrow "): After passing the circle, the topmost k stack symbols are the same again as when entering the circle. Therefore, whenever the circle is passed, the circle will be entered and passed again, as k defines the number of stack symbols popped during the circle with regard to γ_1 . So an endless loop is reached.

Complexity The complexity of the algorithm for a word in a DPDA M α is in $\mathcal{O}(|\alpha| * |Q| * |\Gamma| * t(M))$, as for every letter in α at most every state with every stack symbol on the top of the stack can be visited a bounded number of times. For every determination of the passable transition, at most $t(M)$ (see 3.3.2.2) have to be checked. So, compared to the complexity of the approach with the CFG, the complexity is not exponential in a parameter. Therefore, this algorithm is preferable for DPDAs.

3.3.5 Graders

Now we explain, how the graders for the two problem types calculate the grade and generate the feedback. Each of the grading methods takes the parameters wrapped as XML elements ([XElements](#)) (see 2.2.2). So the methods have to parse the corresponding values from the XML elements. If the XML describes a PDA, a [PDA](#)-instance is created using the static method [PDA.FromXML](#) (see figure 3.6). Thereby, the generic types [A](#) and [S](#) for the alphabet respectively for the stack alphabet (see 3.3.1) are assumed to be [char](#) both. The alphabet and stack alphabet can be read out of the XML description of a PDA using the methods [ParseAlphabetFromXmlPDA](#) and [ParseStackAlphabetFromXmlPDA](#) of the static class [PDAXmlParser](#) (see figure 3.6).

3.3.5.1 English to PDA grader

The computation of the grade and the feedback is run in the class [AutomataPDL.PDA.Graders.ConstructionProblemGrader](#) (see figure 3.6) in the method [GradeConstructionProblem](#) (see 3.1). It takes the correct PDA, the attempt PDA by the student, the option whether the stack alphabet was predefined for the students and the maximum possible grade to reach ([maxGrade](#)). The the two PDAs are compared using the [PDAEqualityResult](#) with maximum word length of 10, a maximum number of tested words of 400000 and a maximum duration of 15s (see 3.3.3).

If the PDA created by the instructor is inconsistent concerning its acceptance condition (see 3.3.1), the students gets the full grade and a feedback with a message, that explains

this fact. Then this is immediately returned, as a further execution does not make sense if the correct PDA is invalid.

Otherwise, the grade is calculated by the formular

Listing 3.3: Calculation of the grading

```
int joinSize = pdaEquality.NumberOfWordsAcceptedByAtLeastOne;  
double proportion = joinSize == 0 ? 1 :  
(double)pdaEquality.NumberOfWordsAcceptedByBoth / joinSize;  
int grade = (int) (maxGrade * proportion);
```

where `pdaEqualityResult` is the `PDAEqualityResult`-instance by the comparison of the PDAs above (for its properties, see 3.3.3).

In order to explain, why this formular makes sense, assume that C is the set of words accepted by the correct PDA and A is the set of words accepted by the attempt PDA. Thus, if $C \cup A$ is empty, the proportion in the formular is 1, which makes sense, as both PDAs accept the empty language and are equal, hence (of course only regarding the set of tested words). If $C \cup A$ is not empty, the the proportion can be expressed as $\frac{|C \cap A|}{|C \cup A|}$. So, for a fixed $C \cup A$, the grade is directly proportional to the size of the cut set of A and B , which obviously makes sense.

Additionally, if the attempt PDA uses more states or stack alphabet symbols, the grade is reduced by 1 (but never smaller than 0). The same is done if the attempt PDA is inconsistent concerning its acceptance condition.

For generating a feedback, a list of strings is created, where the following hints are added in the described case:

- If the words accepted by the attempt PDA are a super set or a sub set of the words accepted by the correct PDA, a corresponding hint is added.
- If `pdaEqualityResult.WordsInPdaCorrectButNotInPdaAttempt` (see 3.3.3) is not empty, the shortest word of this list is added together with a corresponding hint.
- If `pdaEqualityResult.WordsInPdaAttemptButNotInPdaCorrect` (see 3.3.3) is not empty, the shortest word of this list is added together with a corresponding hint.
- If the attempt PDA is inconsistent concerning its acceptance condition, the shortest word with different acceptance is added together with a corresponding hint.
- If the attempt PDA uses more states and stack alphabet symbols, a corresponding hint is added.
- If the student has reached the maximum grade, a hint that the PDA is completely correct is added.

- If the attempt PDA is equal to the correct PDA (approximatively only, of course), but the full grade is not reached because of too many stack symbols or states, a hint that the PDA is almost correct is added.

The grade and the list of feedbacks is then returned, after wrapping them up in XML.

3.3.5.2 Words of PDA grader

The method `GradeWordProblem` (see 3.1) in the class `AutomataPDL.PDA.Graders.WordProblemGrader` (see figure 3.6) computes the grade and feedback for problems of the type "Words of PDA". The parameters of the method are the PDA, the words in the language of the PDA (referred to as *wordsIn* in the following), the words not in the language of the PDA (referred to as *wordsNotIn* in the following) and the maximum possible grade. For the feedback a list of strings is initialized again. Then for every word in *wordsIn* and every word in *wordsNotIn* is tested, if it is correct. A word is correct, if and only if all following conditions hold:

- The word contains only letters of the alphabet. In particular, this is important for the words of *wordsNotIn*, as the students could try to simply enter words with letters outside the alphabet.
- The word is the first occurrence in its list. Of course, the students should enter distinct words.
- If the word is in *wordsIn*, it is accepted by the PDA, if it is in *wordsNotIn*, it is not accepted by the PDA.

For every incorrect word, a hint is added to the feedback list that explains the violated condition above.

If for any word the `InconsistentPDAException` is thrown, the students gets the full grade again and feedback that explains this fact, and this is returned immediately.

Otherwise, assuming that n is the total number of correct words, the grade is calculated like this:

Listing 3.4: Calculation of the grading

```
int totalNumberOfWords = wordsIn.Count() + wordsNotIn.Count();
int grade = (int) ((double) numberOfCorrectWords * maxGrade /
    totalNumberOfWords);
```

where `numberOfCorrectWords` is the total number of correct words. So for every incorrect word, the grade is simply reduced by $\frac{1}{totalNumberOfWords}$ of the maximum grade. If the maximum grade was reached, a corresponding hint is added to the feedback. Finally, the grade and the list of feedbacks is returned, wrapped in XML.

4 Student evaluation

4.1 Overview

We carried out an evaluation with students, who participated at the Bachelor-lecture "Introduction to the theoretical computer science" at the Technical University of Munich. All students of this lecture were asked to solve two certain exercises in Automata Tutor and to fill in a questionnaire after that.

By this, we wanted to find out whether the created problem types to PDAs are really helpful for students and which things should be improved.

The first exercise (in the following referred to as exercise 1) had the type "Words of PDA" and the title "Nondeterministic counter", where the students should enter five words that are in the language of the given PDA and five words that are not. The PDA was defined over the alphabet $\{a, b\}$ and stack alphabet $\{Z, X, Y\}$ (of which Z is the initial stack symbol) and had the acceptance condition *empty stack*. Its states and transitions are displayed in figure 4.1. This PDA is probably relatively difficult to

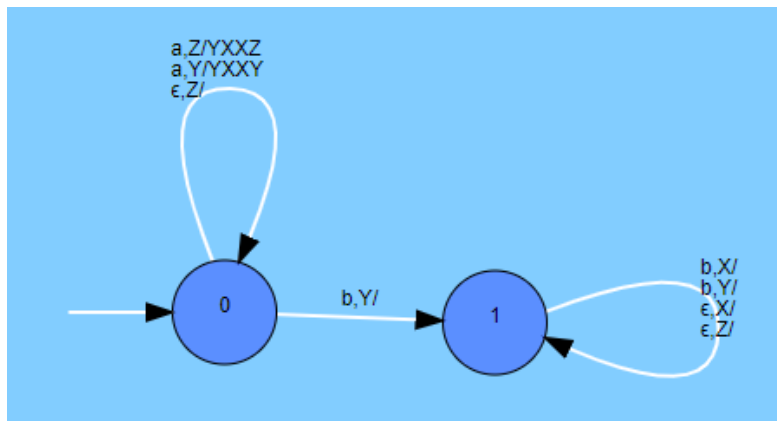


Figure 4.1: States and transitions of the PDA for the exercise 1 of the evaluation

understand for one who is new to PDAs. It accepts the language $\{a^n b b^m \mid n \leq m \leq 2n\}$, so that the title of the exercise is a small hint for how the PDA works.

The second exercise (in the following referred to as exercise 2) had the problem type "English to PDA" and was titled with "Mirrored words". Here the students should

create a DPDA with acceptance condition *final state and empty stack* for the language $\{wxw^R | w \in \{a, b\}^*\}$, defined over the alphabet $\{a, b, x\}$. This exercise had medium difficulty, only the restrictions to the determinism and to both acceptance conditions make it a bit harder.

The maximum reachable grade was ten for both exercises.

At the time we carried out the evaluation the feature to run simulations was not available yet, so it was not considered in the questionnaire. Moreover, the design was different and some features were added later (see also Section 4.3).

4.2 Questionnaire and answers

Questions The questionnaire consists of five question blocks. In the first block the students were asked about the results of solving exercise 1, as for example their final grade or how difficult they found the exercise in a range between 1 (too easy) and 5 (too hard). The second one included an evaluation of the user interface of exercise 1. At this block the students could give an evaluation to different aspects of the user interface in a range between 1 (excellent) and 5 (insufficient). The third and fourth block had a similar content as the first two block, but for exercise 2. The last block consisted of an evaluation of the user interface in general and of a field for giving individual feedback.

Answers We got a total of 17 answers for the questionnaire. They are summed up in the following.

Statistical evaluation The table 4.1 contains the first block and the third block, that means the results of solving the exercises.

Table 4.1: Averages of the answers about the results of solving the exercises

| Exercise type | Number of Attempts | Final grade | Time in min | Difficulty | Helpful |
|----------------|--------------------|-------------|-------------|------------|------------------------|
| Words of PDA | 1.8 | 10 | 4.9 | 2.6 | yes (88%), no (12%) |
| English to PDA | 3.4 | 9.2 | 7.6 | 2.8 | yes (100%) |

As table 4.1 shows, most students solved both exercises properly with an appropriate number of attempts in a realistic time. The difficulty was estimated between easy and appropriate. Both problem types were seen as helpful for getting a better understanding of PDAs.

The tables 4.2 and 4.3 display the results of the second and fourth question block, that means the evaluations of the user interfaces of both exercises.

Table 4.2: Averages of the answers about the user interface at exercise 1

| Representation of the PDA | Grading and feedback | Overall |
|---------------------------|----------------------|---------|
| 2 | 2 | 2.1 |

Table 4.3: Averages of the answers about the user interface at exercise 2

| Language description | Hints for PDA construction | PDA construction | Grading and feedback | Overall |
|----------------------|----------------------------|------------------|----------------------|---------|
| 1.8 | 2.1 | 2.5 | 2.6 | 2.4 |

Table 4.4: Averages of the answers about the user interface in general

| Problem description | Description of the PDA properties | Overall |
|---------------------|-----------------------------------|---------|
| 2 | 2.2 | 2.5 |

Individual comments Apart from that statistical evaluation we got also a lot of helpful individual feedback. Among positive feedback, as for example for the text input for transitions, the points of criticism were the following:

1. Editing the transitions of a link is finished by clicking somewhere else (see figure 3.3 and 3.2.1.1). When clicking somewhere on the area, apart from finishing editing the transitions also a new state was created (see 3.2.1.1), which is unwanted in most cases. The same held for closing the context menu.
2. It seemed to be unclear to a lot of students that a transition, that only pops from the stack, must have an empty string for the pushed stack symbol sequence (thus, the form $a, X/$) (see 3.2.1.1). Some thought that "E" should be used instead (as it is for ϵ -transitions).
3. Some students criticized that transitions of the wrong format were removed silently instead of showing an error message.
4. The hints seemed to have contained too much text.

5. The button for changing the stack alphabet was below the input field, so that it was not directly evident that the button belongs to the stack alphabet input.
6. The acceptance condition *final state and empty stack* was not understood correctly by all students. Some thought that the PDA has to accept every word in the language by final state and empty stack at the same time.
7. The input field for the transitions did not automatically scale when more lines were entered than the default height allowed. In Google Chrome the arrow keys could be used for moving between the lines, but only three lines were shown at once. In Mozilla Firefox even that was not possible.
8. As we did not test the PDA construction tool in Mozilla Firefox properly, there was a bug concerning the context menu, which led to more than one shown context menu at a time.

4.3 Improvements after the evaluation

Firstly, the grade and the feedback for the problem type "English to PDA" only got a grade of 2.6. A reason for that was not named in the individual feedback. We guessed that the only medium rating comes from the following fact: An attempt PDA, that is almost correct, may possibly get only a little grade in certain circumstances, if for example the acceptance condition is *final state* and the student forgot to mark any state as final. This follows from the grading algorithm, that does not compare the two PDAs structurally, but only concerning the words they accept. However, an improvement is hard to reach here, as a good and helpful structural comparison is boundless and very difficult to realize properly.

The improvements after the evaluation were done according to the numerical grade of the single components (see 4.2, 4.3, 4.4), to the individual feedback (see the 9 points above) and to some known improvable aspects.

With respect to the individual comments, the following things were improved:

1. Now when clicking somewhere on the area, a new state is only created if no context menu is opened and when no transitions are edited at the moment.
2. We changed the hints so that it should be clear, how to create a transition that only pops from the stack.
3. We adapted the parsing process of the transitions in the text input field. Now only transitions with a completely wrong form are ignored: If the transition has at least the format $a, X/\alpha$ but maybe with any symbol that is not part of the alphabet

respectively the stack alphabet, then this transition and the corresponding link is marked red to show its faultiness.

4. We tried to show the hints in a clearer form and made them collapsible, so that they do not disturb someone who already knows how to use the GUI.
5. We placed the buttons for applying the new alphabets directly next to the input fields.
6. If the acceptance condition *final state and empty stack* is selected, then an extra hint is displayed next to it. This hint clarifies what this acceptance condition means exactly.
7. Now the text input field for transitions scales automatically with changing line number, until a certain maximum height. Then a scroll bar is shown. To make the text input field work properly in Firefox, we replaced the *svg-foreignObject* by a normal html text area, which is drawn over the svg. The *foreignObject* does not work in all browsers soundly, so that this is the better solution in general.
8. We fixed that bug.

Furthermore, we made the alphabets changeable "on the fly": before the whole automaton was reset when changing one of the alphabets. That was a bit annoying especially for the stack alphabet.

The layout of the sector with the properties was improved in general, so that it can be captured better at a glance.

5 Conclusion and outlook

Summarizing, we added much functionality of PDAs to Automata Tutor. That included expanding the frontend by a graphical user interface for creating and presenting a PDA in a clear and comfortable way and for running simulations of words. We added two different problem types about PDAs, which use this GUI. An instructor can create problems of these types, so that students can try to solve these, with getting an automatically generated grade and feedback.

For generating these two things for a student's solution attempt, we also extended the backend of Automata Tutor. There we implemented several PDA algorithms, which are necessary for the graders and the simulation, e.g. efficient ways of solving the word problem, procedures for deciding whether two PDAs can be regarded as equal as well as algorithms for computing a simulation path of a word.

The student evaluation helped us to improve the functionality, especially the GUI, and confirmed, that the created PDA problem types are really helpful for understanding this topic better.

We already plan some further extensions of the PDA functionality. The simulation of a word, where a path is pre-calculated so far, should be also available in a free form, so that the user can choose one of the passable transitions to enter this one next. Furthermore, the equivalence of DPDAs should be tested exactly. From the constructive proof (see [10]) of the decidability of the equivalence of two DPDAs an algorithm of doing that can be implemented. We have already begun with that, the first steps can be found in the backend project.

List of Figures

| | | |
|------|--|----|
| 2.1 | Example PDA | 4 |
| 3.1 | Properties immutable in the GUI | 11 |
| 3.2 | Properties editable in the GUI | 11 |
| 3.3 | States and transitions sector of the GUI | 14 |
| 3.4 | Simulation snapshot | 15 |
| 3.5 | Class diagram of the GUI | 17 |
| 3.6 | Class diagram of the backend | 23 |
| 3.7 | PDA with endless loops | 24 |
| 3.8 | PDA for performance test | 30 |
| 3.9 | Performance test | 31 |
| 3.10 | PDA with exponential runtime | 33 |
| 3.11 | Class diagram of the CFGUtils | 35 |
| 4.1 | PDA of exercise 1 | 43 |

List of Tables

| | | |
|-----|---|----|
| 4.1 | Answers about solving the exercises | 44 |
| 4.2 | Answers about the user interface of the first exercise | 45 |
| 4.3 | Answers about the user interface of the second exercise | 45 |
| 4.4 | Answers about the user interface in general | 45 |

Bibliography

- [1] e. a. D'Antoni Loris. *Automata Tutor and what we learned from building an online teaching tool*. 2015. URL: <http://eatcs.org/beatcs/index.php/beatcs/article/viewFile/365/347>.
- [2] M. Helfrich. *Kontextfreie Grammatiken in AutomataTutor. Bachelor's thesis*. 2017.
- [3] J. Wagener. *Lehren von Algorithmen für reguläre Sprachen mithilfe von AutomataTutor. Bachelor's thesis*. 2017.
- [4] T. E. Ajdari. *Extending the tool AutomataTutor with Turing machines. Bachelor's thesis*. 2018.
- [5] C. Backs. *Automatic generation of exercises on Turing machines. Bachelor's thesis*. 2018.
- [6] U. Schöning. *Theoretische Informatik - kurz gefasst*. Spektrum Akademischer Verlag Heidelberg, 2009.
- [7] *JavaScript Versions*. URL: https://www.w3schools.com/js/js_versions.asp (visited on 10/15/2018).
- [8] M. Bostock. *Data-Driven Documents*. 2017. URL: <https://d3js.org/> (visited on 10/09/2018).
- [9] T. K. et al. *webpack*. 2018. URL: <https://webpack.js.org/> (visited on 10/09/2018).
- [10] C. Stirling. *Deciding DPDA Equivalence is Primitive Recursive*. 2002.