TU Berlin
Fakultät IV
Institut für Technische Informatik und Mikroeletronik
Fachgebiet Regelungstechnik

Controller Synthesis for Discrete Event Systems in the Setting of a Regular Plant and a Deterministic Context-Free Specification in Libfaudes

Master Thesis

Stefan Jacobi

Berlin, 02.05.2013

Supervisors: Sven Schneider, Anne-Kathrin Hess

1. Examiner: Prof. Jörg Raisch
2. Examiner: Prof. Uwe Nestmann

Contents

Abbrevations

λ	empty word, also called ε
$\overline{L}$	Prefix closure of L
L	Language
L_m	Marked language
DFA	Deterministic finite automaton
DPDA	Deterministic pushdown automaton
EDPDA	Extended deterministic pushdown automaton
SDPDA	Simple deterministic pushdown automaton

Kurzfassung

Ereignisdiskrete Regelungstechnik nach Ramadge und Wonham [4] erzeugt minimal-restriktive Regler für eine Strecke und eine Spezifikation, die als reguläre Sprache dargestellt werden können. Da reguläre Sprachen eine sehr einschränkende Sprachklasse sind, sind die Modellierungsmöglichkeiten für Strecke und Spezifikation beschränkt. Durch das Erweitern der Spezifikation auf deterministisch-kontextfreie Sprachen anstatt von regulären wird mehr Freiheit im Entwerfen von Spezifikationen gewonnen, während es immer noch möglich ist, einen minimal-restriktiven Regler zu entwerfen [5]. Weil deterministische endliche Automaten die deterministisch-kontextfreie Sprachen nicht ausreichend visualisieren können, werden deterministische Kellerautomaten benutzt. Die größte Herausforderung beim Berechnen des Regelkreises stellt das Lösen von Blockierungsproblemen dar. Im Gegensatz zur klassischen Herangehensweise nach Ramadge und Wonham werde Zustände eines Automaten dabei nicht nur gelöscht, sondern auch neue hinzugefügt. Dies geschieht, indem der Automat in eine deterministisch-kontextfreie Grammatik, dann in einen LR(1)-Parser und schließlich zurück in einen deterministischen Kellerautomanten transformiert wird. Der Algorithmus, der einen minimal-restriktiven Regler berechnet, wurde in dem Open-Source-Framework Libfaudes [1], einem Framework für ereignisdiskrete Systeme, realisiert. Ein neu entworfenes Plugin enthält alle Datenstrukturen, die der voll funktionsfähige Algorithmus benötigt. Für den Algorithmus besteht Optimierungspotenzial bezüglich der Laufzeit.

Abstract

Supervisory control theory as it was first introduced by Ramadge and Wonham [4] constructs minimally restrictive controllers from a plant and a specification that can be represented as regular languages. Because regular languages are a very restrictive language class, the scope of valid plant and specification models is limited. Allowing the specification to be a deterministic context-free language instead of a regular one allows for more freedom in design while still being able to construct a minimally restrictive controller [5]. Because deterministic finite automatons are not sufficient for visualizing deterministic context-free specifications, deterministic pushdown automatons are used. Solving the nonblocking problem while constructing the closed loop is the biggest challenge of the algorithm. Contrary to the classic approach by Ramadge and Wonham, states may not only need to be deleted from an automaton, but new ones may have to be added as well. This is done by converting the automaton to a deterministic context-free grammar, then to a LR(1) parser and then back to a deterministic pushdown automaton. The algorithm to construct the minimally restrictive controller has been realized in Libfaudes [1], an open source framework for discrete event systems. A plugin was designed to support all data structures required by the algorithm, which is fully functional. Potential for optimization remains regarding the algorithm's execution time.

Chapter 1

Summary

Classic supervisory control theory (see Ramadge and Wonham [4]) employs regular languages to modell plant and specification behavior. This approach is limited in terms of its expressiveness, because regular languages are the least powerful language in the Chomsky hierarchy. Extending the specification behavior to deterministic context-free languages (see Schneider and Hess [5]) while still being able to calculate a minimally restrictive controller greatly improves the freedom in modelling desired behaviors.
Realizing a language can be done in a number of different ways. The most common in control theory is by way of a deterministic finite automaton (DFA), which is an automaton that represents regular languages. A DFA can also be represented by a regular grammar, which is equivalent to the automaton representation. Context-free languages are represented by pushdown automatons. A pushdown automaton is a DFA that has been extended with a stack and stack operations along its transitions. For this thesis, only deterministic pushdown automatons (DPDAs) are relevant. As with DFAs, DPDAs can be converted into grammars, which are called deterministic context-free grammars (CFGs). These grammars can in turn be converted into LR(1) parsers. A parser can decide for an input string whether it belong to a language or not.
All these language representations are needed for the supervisor algorithm presented in this thesis. The classic algorithm by Ramadge and Wonham requires the product of plant and specification and iterative trimming of the supervisor candidate to remove blocking and controllability problems. The supervisor algorithm that takes deterministic context-free specifications into account does basically the same, although it requires more caution in removing blocking and controllability problems. Because DPDAs are used instead of DFAs, these problems are not solely dependent on states, but on combinations of states and stack top symbols. In order to get the minimally restrictive controller, only the right state/stack top combination must be removed, as just removing a state will often result in a supervisor that is too restrictive.
The controllability problem can be solved by splitting the DPDA into states that can be uniquely assigned to a single stack symbol. The nonblock problem is solved in a more roundabout way. The potentially blocking DPDA is converted into a CFG, then converted into a parser, which is then converted back to a DPDA. The blockingness is removed in the grammar step and the parser step is only required for DPDA conversion. This process does not preserve the structure of the original blocking DPDA, as is the case with the classic supervisor algorithm. In fact, DPDAs made nonblocking in this way are often get bigger rather than smaller. Removing nonblocking and controllabilty problems is done iteratively until all problems are removed. At this point, the minimally restrictive supervisor is found.
This new supervisor algorithm is integrated into the supervisory control framework Libfaudes [1]. Libfaudes is an open source framework and supports plugins to extend its functionality. Libfaudes offers only support for DFAs with various extensions like timing and controllability. A new plugin called Pushdown has been written to facilitate DPDAs and the related grammar and parser structures. Because DPDAs are based on Libfaudes DFA types, certain restrictions had be worked around to support λ transitions and multiple different stack operations on one transition. The supervisor algorithm has been split up thematically into different files to accomodate its large size. Because of function dependencies, implementation has been done bottom-up. Functions have been thoroughly unit tested before advancing to the next higher-level function. In the case of errors, the test suite provides a good starting point in the

search for the cause.
The implementation of the algorithm as a whole has been tested and is fully functional. However, it has shortcomings in terms of execution time and memory consumption. The biggest problem is posed by the function converting DPDAs into CFGs, called Sp2Lr(). The memory problem of Sp2Lr() could be solved by slightly rewriting the function (as compared to [5]). Execution time could be significantly decreased by truncating the input size of said function in the number of states and stack symbols as well as transitions. Still, execution times for examples a little larger than non-trivial can be unacceptably high.
Potential for further optimization remains, although the scaling of Sp2Lr() in the square of its number of input states cannot be changed and the number of total calls of Sp2Lr() is unknown. The implementation is currently single-threaded and could be changed to multi-threaded, which would be of great benefit to Sp2Lr(). Other potential improvements include providing support for Libfaudes' graphical user interface DESTool [2].

Chapter 2

Introduction

2.1 Motivation

In classic supervisory control theory, as developed by Ramadge and Wonham [4], plant and specification behaviors are modelled using regular languages. Regular languages have well-defined behavior, like closure under intersection, but are very restrictive in what behavior they can modell. Calculating supervisors is done by established algorithms. However, these algorithms cannot be applied if the modell is more complex than what a regular language can modell.
Consider the following example of a regular languages failing to modell real-world circumstances. A catering company produces sandwiches and water bottles and delivers them to their customers. Producing arbitrary numbers of goods can be modelled by a regular plant without problems (an $a^n b^m$ language). Employees pack sandwiches and bottles according to the ordered amounts and send them off to the customer. This requires no supervisor, because everything is done manually. For efficiency reasons, the company's management decides to automate packaging. Since nearly all customers order the same amount of sandwiches and water bottles, a supervisor is needed to match the number of sandwiches to the number of bottles. The automation plans fail, because the supervisor requires matching arbitrarily large numbers (an $a^n b^n$ language), which is beyond the scope of regular languages.
However, number matching, among other things, can be easily done with context-free languages. Creating a supervisor algorithm that works on context-free specifications expands the application range of supervisory control theory significantly.

2.2 Discrete Event Systems and Grammars

Discrete event systems are often visualized as automatons. A common kind of automaton to use for this purpose is the deterministic finite automaton (DFA). If the DFA is too restrictive in terms of expressiveness, the more powerful, but also more complex determinisitc pushdown automaton (DPDA) can be used.
Grammars, which are commonly used in computer science, are closely related to automatons. They can be divided into different classes and an equivalence between certain grammars and certain types of automatons can be shown. Grammars and automatons and their languages as well as parsers for languages are explained in the following sections to the extent necessary for this thesis.

2.2.1 DFAs and Regular Grammars

The most basic automaton is the deterministic finite automaton (DFA). It is a 5-tuple A:

$$A = (Q, \Sigma, \delta, q_0, Q_m) \tag{2.1}$$

consisting of the elements

- Q - the set of states,

- Σ - the set of events (the alphabet),
- δ - the set of transitions,
- q_0 - the initial states, and
- Q_m - the set of marked states.

A transition is a 3-tuple consisting of elements (p, a, q) with $p, q \in Q$ and $a \in \Sigma$. The set of events can be divided into controllable and uncontrollable events, $\Sigma = \Sigma_{uc} \cup \Sigma_c$. The transitions can also be written as function $\delta : Q \times \Sigma \rightarrow Q$.

A DFA A recognizes or marks a language $L_m(A)$ that is equal the set of event strings that can be generated when traversing the automaton from the initial state to a marked state. For example, the DFA shown in Figure 2.1 recognizes the marked language $a^n b$.
The language $L(A)$ is called the language generated by A. It is equal the the set of event that strings that can be generated when traversing the automaton from the initial state to any other state. The language L generated by the DFA from Figure 2.1 is $a^n b \cup \{\lambda\}$.
The language $\overline{L(A)}$ is called the the prefix closure of $L(A)$. It contains the set of substrings $s \in \Sigma^*$ if $\exists t \in \Sigma^*$ with $st \in L(A)$. In the case of the example automaton, $\overline{L(A)} = L(A)$, but generally $\overline{L(A))} \subseteq L(A)$.
Languages that can be generated by DFAs are called regular languages and every regular language can be defined by a DFA.

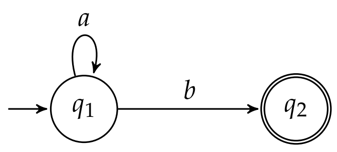


Figure 2.1: DFA with the $L_m = a^n b$, $L = a^n + a^n b$

Another way of defining regular languages is by means of a left (or right) regular grammar. While DFAs are sufficient to define a regular language, grammars may be more manageable or concise, depending on the situation.
A regular grammar is a 4-tuple G:

$$G = (N, \Sigma, P, q_0) \tag{2.2}$$

consisting of the elements

- N - the set of nonterminals,
- Σ - the set of terminals,
- P - the set of productions, and
- q_0 - the start symbol.

A production of a left regular grammar is a 2-tuple (w_1, w_2) (also written as $w_1 \rightarrow w_2$) with $w_1 \in N$ and $w_2 \in \{\lambda\} \cup \Sigma \cup \Sigma N$. Right regular grammars define the production's w_2 as $w_2 \in \{\lambda\} \cup \Sigma \cup N\Sigma$.
The language $a^n b$ defined as a (left) regular grammar with $\Sigma = \{a, b\}$, $N = \{S, A\}$ and $q_0 = S$ has the following productions:

$$\begin{aligned} S &\rightarrow A \\ A &\rightarrow aA \\ A &\rightarrow b \end{aligned} \tag{2.3}$$

Regular grammars and DFAs are equal and can be transformed into each other. However, regular languages are very restrictive language. As shown in Figure 2.2, regular languages are the most restrictive of all language classes in the Chomsky hierarchy. Languages like $a^n b^n$, where counting the occurences of a and matching it with an equal number of b is required, are not in the class of regular languages. This shortcoming is remedied by the next class of languages, the context-free languages.

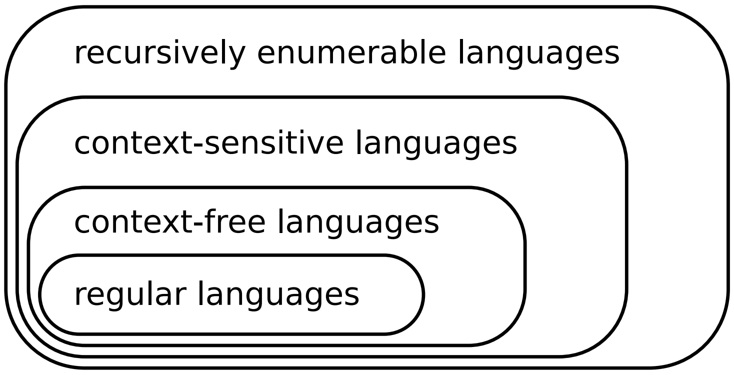


Figure 2.2: Chomsky hierarchy of languages

2.2.2 DPDAs and Context-Free Grammars

The class of context-free languages is easily defined by context-free grammars. It is the same 4-tuple as defined in def. 2.2, except that the productions are defined slightly differently.
A production of a context-free grammar is a 2-tuple $w_1 \rightarrow w_2$ with $w_1 \in N$ and $w_2 \in (N \cup \Sigma)^*$.
The language $a^n b^n$ can then be defined by the following productions and with λ as the empty word:

$$\begin{aligned} S &\rightarrow A \\ A &\rightarrow aAb \\ A &\rightarrow \lambda \end{aligned} \tag{2.4}$$

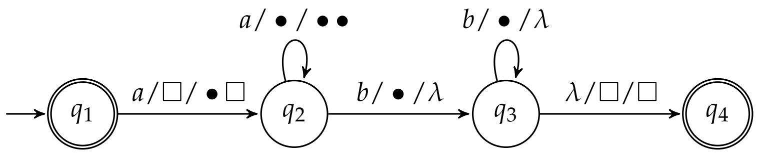
Like the regular grammars have an automaton equivalent in the form of DFAs, the context-free grammars have an automaton equivalent in the form of pushdown automatons. Pushdown automatons that are deterministic are called deterministic pushdown automaton (DPDA) and represent a subset of these grammars, called the deterministic context-free grammars. A DPDA is a DFA that has been extended with a stack and the stack operations *pop* and *push*. Using this stack, a DPDA has the ability to memorize past events, for example counting the occurences of a.
The formal definition of a DPDA is as follows:

$$A = (Q, \Sigma, \delta, q_0, Q_m, \Gamma, \square) \tag{2.5}$$

consisting of the elements

- Q - the set of states,
- Σ - the set of events (the alphabet),
- δ - the set of transitions,
- q_0 - the initial states,
- Q_m - the set of marked states,
- Γ - the set stack symbols, and
- $\square$ - the stack bottom

A transition is a 5-tuple consisting of elements (p, a, u, v, q) with $p, q \in Q$, $a \in \Sigma$, $u \in \Gamma$ and $v \in \Gamma^*$. u is the stack pop part of a transition and v is the stack push part of a transition. A DPDA that represents the $a^n b^n$ language is shown in Figure 2.3. Table 2.1 shows how the word $aaabbb$ is consumed by this DPDA. Every a will push one $\bullet$ onto the stack and every b pops one $\bullet$ from the stack. Because the marked state q_4 can only be reached when all $\bullet$ symbols are popped from the stack and $\square$ is the top stack symbol, there must be one b for every a.



$a/\bullet/\bullet\bullet$
$b/\bullet/\lambda$
q_1
$a/\square/\bullet\square$
q_2
$b/\bullet/\lambda$
q_3
$\lambda/\square/\square$
q_4

Figure 2.3: DPDA with the language $a^n b^n$. The transitions are read as (event/pop/push).

Event	State	Stack	Comment
λ	q_1	$\square$	accepting
a	q_2	$\bullet\square$	push $\bullet$
a	q_2	$\bullet\bullet\square$	push $\bullet$
a	q_2	$\bullet\bullet\bullet\square$	push $\bullet$
b	q_3	$\bullet\bullet\square$	pop $\bullet$
b	q_3	$\bullet\square$	pop $\bullet$
b	q_3	$\square$	pop $\bullet$
λ	q_4	$\square$	accepting

Table 2.1: Consuming the word $aaabbb$

The marked language of this automaton is $L_m(A) = a^n b^n$ and the recognized language is $L(A) = a^n b^m$, $n \geq m$.

A DFA can be seen as a special form of DPDA. If the stack is not used, that is all transitions are of the form $(p, a, \square, \square, q)$, it behaves exactly as a DFA.
For a less verbose notation of DPDAs, the extended DPDA (EDPDA) is defined. It is identical to a DPDA in all parts except for the pop part u of the transitions (p, a, u, v, q). It is redefined as $u \in \Gamma^*$. This enables the automaton to pop multiple stack symbols at once and can lead to less verbose notation.
Additionally, the simple DPDA (SDPDA) is defined. It is more verbose than the DPDA, but can be easier to handle, because all the SDPDA's transitions must fall into one of three categories:

1. read transitions (p, a, u, u, q) with $p, q \in Q$, $a \in \Sigma$, $u \in \Gamma$,
2. push transitions (p, λ, u, uv, q) with $p, q \in Q$, , $u, v \in \Gamma$, or
3. pop transitions $(p, \lambda, u, \lambda, q)$ with $p, q \in Q$, , $u \in \Gamma$.

EDPDAs, DPDAs and SDPDAs can be transformed into each other by merging transitions and deleting states or by splitting up transitions and inserting new states.

2.2.3 LR(1) Parsers

LR Parsers are parsers that can recognize deterministic context-free languages. They parse the input from left to right (hence the "L") and produce a reversed rightmost derivation (hence the "R"). An LR(k) parser has the ability to look k symbols ahead and make decisions based on what symbols will be coming. LR parsers are bottom-up parsers, which means that they will first recognize the fine-grained parts of a language, namely the grammar's terminals. They will then retrace the productions that were used to produce the terminal. New terminals can be found, which are retraced as well. This retracing

continues all the way back to the start nonterminal, at which point the parser can recognize the word as part of the language.
Formally, an LR(1) parser is defined as a 6-tuple

$$P = (Q, \Sigma, q_0, q_m, R, \$) \tag{2.6}$$

consisting of the elements

- Q - the set of states,
- Σ - the set of terminals,
- q_0 - the intial state,
- q_m - the marked state,
- R - the set of parser actions
- $\$$ - the augment symbol,

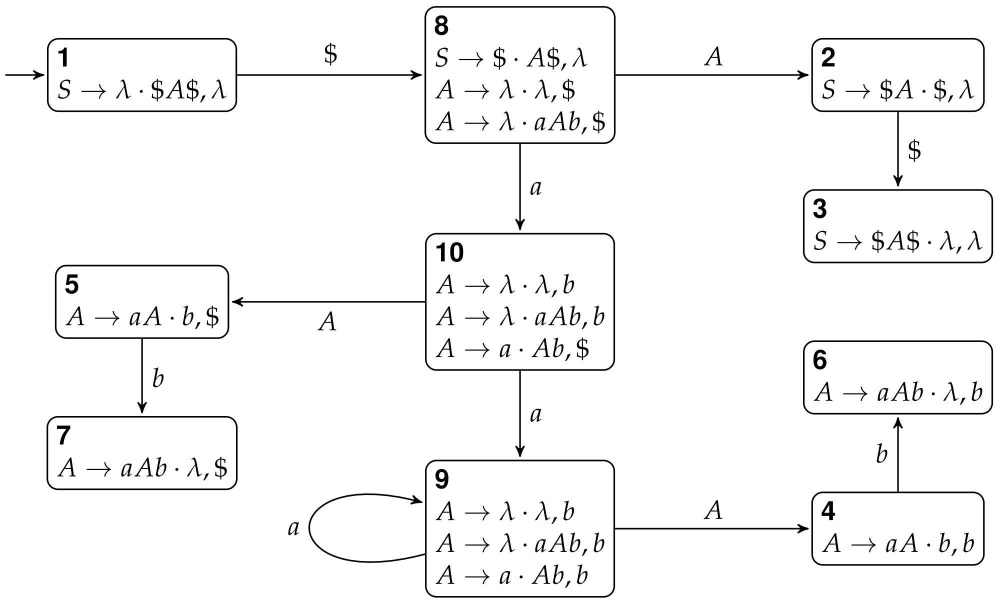
A parser action is a 4-tuple (q, a, q', a') (written as $q|a \rightarrow q'|a'$) with the state sequences $q = q_1, ..., q_i$ and $q' = q'_1, ..., q'_j$, $i, j \geq 1$, the terminal $a \in \Sigma$ that will be parsed next and the symbol a' that denotes whether a is consumed or not.
When parsing a word, the parser is always in a certain configuration. These configurations relate directly to the grammar productions that are read. A parser configuration is a 4-tuple (U, V, W, x), written as

$$U \rightarrow V \cdot W, x \tag{2.7}$$

with $U \in G.N$, $V, W \in (G.N \cup G.\Sigma)^*$ and $x \in (G.\Sigma \cup \{\lambda\})$. $U \rightarrow VW$ must be a production of G. The divider $\cdot$ represents how far the parser has progressed parsing the production. x is the lookahead terminal that will occur after the production has been consumed.
Consider the modified $a^n b^n$ grammar G, which was augmented with $\$$. The $\$$ symbol signals the parser where the input starts and where it ends.

$$\begin{aligned} S &\rightarrow \$A\$ \\ A &\rightarrow aAb \\ A &\rightarrow \lambda \end{aligned} \tag{2.8}$$

The starting configuration for this grammar is $S \rightarrow \lambda \cdot \$A\$, \lambda$, because at the very start the parser has not progressed through the first production. Since it is the first production, the parser knows that the lookahead, that is, the symbol that is read after the production is consumed, must be λ.
When the $\$$ is consumed, the dot gets shifted and the successor configuration is $S \rightarrow \$ \cdot A\$, \lambda$. Now the dot is right before the nonterminal A. This leads to two more possible configurations in this parser state, because there are two productions for A in the grammar.
If the next input symbol turns out to be a, the successor configuration could just as well be $A \rightarrow \lambda \cdot aAb, \$$. The lookahead for this configuration is $\$$, because the parser knows that $\$$ is the symbol that must follow after consuming the production. If the following symbol is not a but $\$$, the production $A \rightarrow \lambda$ needs to be consumed before reading $\$$. Thus, the successor configuration could also be $A \rightarrow \lambda \cdot \lambda, \$$.
Because all of these configurations are possible, they are called a configuration set. Each configuration set forms a state of the so called LR machine, see Figure 2.4. Configurations sets are connected by transitions with events that correspond to the terminal or nonterminal symbol over which the dot has been shifted in order to reach it.
While parsing, the parser simulates the changing of the LR machine's state and it tracks the state sequence by means of a stack. Depending on the stack and the input symbol, one of two action types is applied. It can either be a shift action, in which an input symbol is consumed, or a reduce action, in which an input symbol is looked at, but not consumed. Instead, a grammar production is completed, that is, a nonterminal is consumed. The simulated change of the LR machine's state depends on the lookahead symbol and the current stack.
Shift actions make the parser simulate the transition that originats at the stack top state and has the consumed symbol as its event. The $\cdot$ symbol in $A \rightarrow w_1 \cdot aw_2, z$ is shifted over a. This leads to q_a as a



1
$S \rightarrow \lambda \cdot \$A\$, \lambda$
$
8
$S \rightarrow \$ \cdot A\$, \lambda$
$A \rightarrow \lambda \cdot \lambda, \$$
$A \rightarrow \lambda \cdot aAb, \$$
A
2
$S \rightarrow \$A \cdot \$, \lambda$
$
3
$S \rightarrow \$A\$ \cdot \lambda, \lambda$
a
10
$A \rightarrow \lambda \cdot \lambda, b$
$A \rightarrow \lambda \cdot aAb, b$
$A \rightarrow a \cdot Ab, \$$
A
5
$A \rightarrow aA \cdot b, \$$
b
7
$A \rightarrow aAb \cdot \lambda, \$$
a
9
$A \rightarrow \lambda \cdot \lambda, b$
$A \rightarrow \lambda \cdot aAb, b$
$A \rightarrow a \cdot Ab, b$
a
A
4
$A \rightarrow aA \cdot b, b$
b
6
$A \rightarrow aAb \cdot \lambda, b$

Figure 2.4: LR machine for grammar 2.8, $q_0 = 1$, $q_m = 3$

new stack top state. Shift actions R_{Shift} are inferred as follows:

$$\begin{aligned}
&\forall q \in Q_{LR}, \forall a \in G.\Sigma : \\
&\qquad q_a \leftarrow \delta_{LR}(q, a) \\
&\qquad \forall [A \rightarrow w_1 \cdot a w_2, z] \in q_a, \text{NextPossibleTerminal}(w_2 z) \neq \emptyset : \\
&\qquad\qquad R_{\text{Shift}} \leftarrow R_{\text{Shift}} \cup \{q|a \rightarrow q q_a|\lambda\}
\end{aligned} \tag{2.9}$$

Reduce actions make the parser simulate the completion of a rule $A \rightarrow w$ and make it jump to a state that is not necessarily connected to the current stack top state by a transition. However, this target state can always be reached from a state q further down the stack with a transition that has the consumed nonterminal A as its event. The jump is done by swapping the stack top states q_w for a new stack top state q_A. Reduce actions R_{Reduce} are inferred as follows:

$$\begin{aligned}
&\forall q \in Q_{LR}, \forall [A \rightarrow \lambda \cdot w, z] \in q, |w| = n : \\
&\qquad q_A \leftarrow \delta_{LR}(q, A) \\
&\qquad q_w \leftarrow (q_{w,1}, \ldots, q_{w,n}) = (\delta_{LR}(q, w^1), \ldots, \delta_{LR}(q, w^n)) \\
&\qquad \forall [A \rightarrow w \cdot \lambda, y] \in q_{w,n} : \\
&\qquad\qquad R_{\text{Reduce}} \leftarrow R_{\text{Reduce}} \cup \{q q_w|y \rightarrow q q_A|y\}
\end{aligned} \tag{2.10}$$

The parser actions for the example grammar can be inferred as shown in Table 2.2.

Shift actions	Reduce actions
$8\|a \rightarrow 8, 10\|\lambda$	$9\|b \rightarrow 9, 4\|b$
$9\|a \rightarrow 9, 9\|\lambda$	$10\|b \rightarrow 10, 5\|b$
$10\|a \rightarrow 10, 9\|\lambda$	$9, 9, 4, 6\|b \rightarrow 9, 4\|b$
$4\|b \rightarrow 4, 6\|\lambda$	$10, 9, 4, 6\|b \rightarrow 10, 5\|b$
$5\|b \rightarrow 5, 7\|\lambda$	$8\|\$ \rightarrow 8, 2\|\$$
	$8, 10, 5, 7\|\$ \rightarrow 8, 2\|\$$

Table 2.2: Shift and reduce actions inferred from the LR machine in Fig. 2.4

Note that actions for consuming the $\$$ symbol are omitted, because $\$$ does not belong to the grammar $a^n b^n$. Likewise, the starting state and marked state of the parser are not equal to the LR machine's starting state and marked state for the same reason. $q_{\text{LRM}0}$ and $q_{\text{LRM}m}$ represent states not belonging to the gramamr $a^n b^n$. The parser's starting state is $q_0 = 8$, because it is the immediate successor of $q_{\text{LRM}0}$, the parser's marked state is $q_m = 2$, because it is the immediate predecessor of $q_{\text{LRM}m}$.
As an example, the shift action $8|a \rightarrow 8, 10|\lambda$ means that if the LR machine is in state 8 and a is read, the following state is 10. The reduce action $9, 9, 4, 6|b \rightarrow 9, 4|b$ means that if the LR machine visited the states $9, 9, 4, 6$ last and a b will be read next, change the stack top states to $9, 4$. This is because the sequence $9, 9, 4, 6$ corresponds to the input aAb, which is the right side of the production $A \rightarrow aAb$. The parser recognizes this production and consumes A instead of aAb at state 9.
The word $\$aaabbb\$$ is parsed as shown in Table 2.3.

Input	Lookahead	State Stack	Stack modification	Consumed
$\$aaabbb\$$	$\$$	8	push $q_0 = 8$,	$\$$
$aaabbb\$$	a	$8, 10$	push 10	a
$aabbb\$$	a	$8, 10, 9$	push 9	a
$abbb\$$	a	$8, 10, 9, 9$	push 9	a
$bbb\$$	b	$8, 10, 9, 9, 4$	push 4	$A \rightarrow \lambda$
$bbb\$$	b	$8, 10, 9, 9, 4, 6$	push 6	b
$bb\$$	b	$8, 10, 9, 4$	pop $9, 4, 6$, go to 4	$A \rightarrow aAb$
$bb\$$	b	$8, 10, 9, 4, 6$	push 6	b
$b\$$	b	$8, 10, 5$	pop $9, 4, 6$, go to 5	$A \rightarrow aAb$
$b\$$	b	$8, 10, 5, 7$	push 7	b
$\$$	$\$$	$8, 10, 2$	pop $10, 5, 7$, go to 2	$A \rightarrow aAb$
$\$$	$\$$	$8, 2$	finished, because $q_m = 2$	

Table 2.3: Parsing the word $aaabbb$

Chapter 3

Discrete Event Control Theory

This chapter introduces the supervisory control theory, which was developed by Ramadge and Wonham in the 1980s [4]. Control theory seeks to control and regulate the behavior of systems. The properties of such a system, called the plant P, may not be optimal or may make the system unsafe. For example, the model of a factory could allow for machines to have too much downtime or the model of a tank could allow for overflowing of the tank, potentially spilling toxic substances. For such plants, it is desirable to find a controlling mechanism that reduces machine downtime or prevents the tank from overflowing in the first place.
In the first section of this chapter, blockingness of systems and essential operations on automata like products, accessibility and coaccessibility are explained and the algorithm by Ramadge and Wonham for computing a controller is presented. This algorithm is restricted to plants and specifications that can be represented by regular languages. In the second section, another algorithm by Schneider and Hess [5] is presented. This algorithm allows for deterministic context-free instead of regular specifications, which allows for more freedom in regulating the plant's behavior.

3.1 Supervisory Control Theory

The concepts of this section are taken and, in some cases slightly modified, from [4], [6] and [7]. When regulating a plant with the language L_P, the restrictions imposed by the specification, defined by the language L_S, are applied. L_S defines the maximum allowable behavior of the plant. However, the exact behavior of L_S might not always be enforceable. If L_S specifies that a tank must not overflow, the actual event of overflowing is probably not controllable. Instead, the events leading to overflowing are controllable and the correct action would to close the tank's valves before the overflow can occur. The specification changes from "the tank must not overflow" to "the valves must be closed when an overflow is imminent". Thus, the controlled system's behavior is given by a range of behaviors in between the specification L_S and a minimum of required behavior L_{min}.

$$L_{min} \subseteq L_{P_{\text{controlled}}} \subseteq L_S \tag{3.1}$$

It is desirable to restrict the plant's behavior as minimally as possible. This restriction is called the minimally restrictive controller or supremal sublanguage.

3.1.1 Basic Automaton Operations

When enforcing L_S, it can happen that previously perfectly working systems are suddenly blocking and can get stuck after certain events. Consider two almost identical machines that process materials, the only difference being that one needs fluids to work while the other does not. The fluid are retrieved from a reservoir without fluid level detection systems. New material to process can only be handed to a machine if the currently running machine finished its job. Eventually, the reservoir will be emptied by the machine that depends on the fluids and this machine will stop working. The whole system will shut down if the now broken machine is handed material. Even though the other machine is perfectly functional, it can not receive new materials.

Blocking situations like this must be detected and resolved. An obvious solution to this problem is to never hand the fluid-dependant machine any material. This will not affect production time, because only one machine can be handed material at the same time anyway. Formally, all blocking states of the system must be deleted, which is achieved with the Accessible and Coaccessible operations.

Product

The operation to determine a systems behavior after enforcing L_S is called product. It is denoted by the $\times$ operator. The product automaton $A_\times$ of two DFAs A_1 and A_2 is defined as

$$A_\times = A_1 \times A_2 = (Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, \delta_\times, (q_{0_1}, q_{0_2}), Q_{m_1} \times Q_{m_2}) \tag{3.2}$$

with

$$((p_1, p_2), a, (q_1, q_2)) \in \delta_\times \qquad \text{if } (p_1, a, q_1) \in \delta_1 \wedge (p_2, a, q_2) \in \delta_2 \tag{3.3}$$

On a language level, the operation is equal to the intersection of the languages of both automata.

$$\begin{aligned} L(A_1 \times A_2) &= L(A_1) \cap L(A_2) \\ L_m(A_1 \times A_2) &= L_m(A_1) \cap L_m(A_2) \end{aligned} \tag{3.4}$$

An example is shown in Figure 3.1, 3.2 and 3.3. The product automaton $A_1 \times A_2$ recognizes and generates only the words that both A_1 and A_2 can recognize and generate.
The product automaton also contains a lot of redundant states. States like (q_2, p_1) can never be reached and once a word is started with b, the states (q_1, p_2) and (q_1, p_3) are entered and a marked state can never reached. These blocking and unreachable states can be removed by the Accessible and Coaccessible operations.

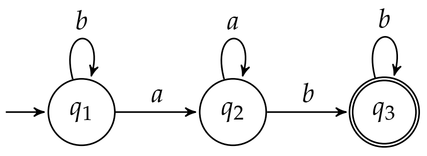


Figure 3.1: Automaton A_1

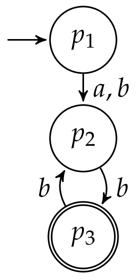


Figure 3.2: Automaton A_2

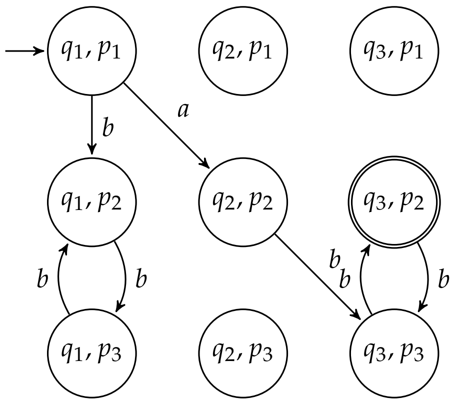


Figure 3.3: Product automaton $A_1 \times A_2$

Accessibility

The accessibility operation, denoted as $Ac(A)$, removes all states from A that cannot be reached from the initial state q_0. It is defined as

$$Ac(A) = (Q_{Ac}, \Sigma, \delta_{Ac}, q_0, Q_{Ac,m}) \tag{3.5}$$

with

$$\begin{aligned} Q_{Ac} &= \{q \in Q : \exists s \in \Sigma^*, \delta(q_0, s) = q\} \\ Q_{Ac,m} &= Q_m \cap Q_{Ac} \\ \delta_{Ac} &= \{(p, a, q) \in \delta : p, q \in Q_{Ac}\} \end{aligned} \tag{3.6}$$

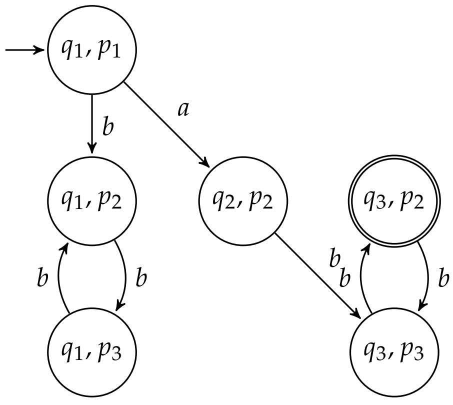


Figure 3.4: Accessible automaton $Ac(A_1 \times A_2)$

Coaccessibility

The coaccessibility operation, denoted as $CoAc(A)$, removes all states from A from whom a marked state cannot be reached. It is defined as

$$CoAc(A) = \{Q_{CoAc}, \Sigma, \delta_{CoAc}, q_{CoAc,0}, Q_m\} \tag{3.7}$$

with

$$\begin{aligned} Q_{CoAc} &= \{q \in Q : \exists s \in \Sigma^*, \delta(q, s) \in Q_m\} \\ q_{CoAc,0} &= \begin{cases} q_0 & \text{if } q_0 \in Q_{CoAc} \\ \text{undefined} & \text{otherwise} \end{cases} \\ \delta_{CoAc} &= \{(p, a, q) \in \delta : p, q \in Q_{CoAc}\} \end{aligned} \tag{3.8}$$

The language $L(A)$ of a coaccessible automaton always equals the prefixes of the marked language $\overline{L_m(A)}$.

3.1.2 Controller Synthesis

Requirements

A controller C that is computed from a given plant P and specification S modifies the language of $L(P)$ to $L(C/P)$ (read as "C controlling P"). C is called implementable, if for $K \subseteq L_m(P)$, $K \neq \emptyset$, and $L(C/P) = \overline{K}$, $L_m(C/P) = K$

$$\overline{K}\Sigma_{uc} \cap L(P) \subseteq \overline{K} \tag{3.9}$$

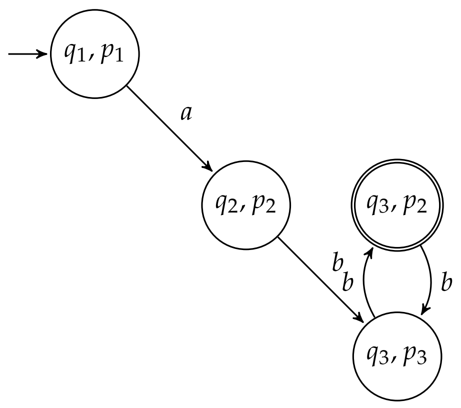


Figure 3.5: Coaccessible automaton $CoAc(Ac(A_1 \times A_2))$

holds. This means that every word from the controlled language $\overline{K}$ that is followed by an uncontrollable event from Σ_{uc} must again be in the controlled language $\overline{K}$, as long as the plant has this word in its language $L(P)$ as well. In other words, the controller must not prevent an uncontrollable event that can occur in the plant. A proof of this condition is given in [8, Thm. 3.4.2].
If no implementable controller for K exists, a subset of K must be used. This subset should be restricted as little as possible and is called the supremal controllable sublanguage $K^{\uparrow C} \subseteq K$. Every solution K_{other} of the implementability requirement 3.9 must be a subset of $K^{\uparrow C}$.

$$K_{\text{other}} \subseteq K^{\uparrow C} \tag{3.10}$$

For the closed loop $L(C/P)$, nonblockingness is required. This is characterized by

$$\overline{L_m(C/P)} = L(C/P) \tag{3.11}$$

Note that while it is often assumed that the specification introduces no new markings to the closed loop, this closed loop closure condition (see [6, p. 163]) is not required here. Such specifications are called marking nonblocking supervisors [8, Thm. 3.5.2]. Dropping this requirement is reasonable, because the advantage of context-free languages over regular languages is counting and ordering events. Such languages would lose their intended meaning if they were prefix closed.

Closed Loop Computation

To compute the supremal controllable closed loop language $K^{\uparrow C}$, the involved languages are represented by DFAs. The following steps are necessary

Step 1 Let the DFA $P = (X, \Sigma, \delta_P, x_0, X_m)$ be the plant and the DFA $S = (Y, \Sigma, \delta_S, y_0, Y_m)$ be the specification

Step 2 Initialize a counter $i = 0$ and compute the product of the plant and the specification

$$H_i = P \times S = (Q_i = X \times Y, \Sigma, \delta_i, q_{0i}, Q_{mi}) \tag{3.12}$$

Step 3.1 Look at every state (x, y) of H_i. If the corresponding state x of P allows an uncontrollable event a but (x, y) does not allow a, then delete (x, y).

$$Q_i' = \{(x,y) \in Q_i : \{a \in \Sigma : (x,a,x') \in \delta_P\} \cap \Sigma_{uc} \subseteq \{a \in \Sigma : ((x,y),a,(x',y')) \in \delta_i\}\} \tag{3.13}$$

Update the transition function δ_i accordingly and restrict it to states in Q_i'.

$$\delta_i' = \delta_i | Q_i' \tag{3.14}$$

If marked states were deleted, update the marked states Q'_{mi}.

$$Q'_{mi} = Q_{mi} \cap Q'_i \tag{3.15}$$

Step 3.2 To resolve blocking issues and delete redundant states, make the automaton accessible and coaccessible

$$H_{i+1} = CoAc(Ac(H'_i)) \tag{3.16}$$

If H_{i+1} is empty (the initial state was deleted), set $K^{\uparrow C} = \emptyset$ and stop the algorithm.

Step 4 If no changes were made in the previous steps, that is $H_i = H_{i+1}$, the supremal controllable sublanguage is found and the algorithm can stop.

$$L_m(H_{i+1}) = K^{\uparrow C} \tag{3.17}$$

If changes were made, increment $i = i + 1$ and continue at Step 3.1.

3.2 Discrete Event Control Theory with Context-Free Specifications

Up until now, only discrete event systems that can be represented as a regular language have been discussed, although the requirements presented in 3.1.2 apply to any kind of language. With regular languages being the most restrictive language class, the controller computed from regular plants and specifications is also very restrictive and may be more restrictive than desired. If the next less-restrictive language class, the deterministic context-free languages, is used, a more fine grained control can be achieved. Moving from regular to deterministic context-free languages means moving from DFAs to DPDAs as well.
However, DFAs cannot be removed entirely. The reason is the product operation $\times$, that needs to be redefined to accomodate DPDAs. On a language level, $L(A_1 \times A_2) = L(A_1) \cap L(A_2)$. For DFAs $A_{1/2}$ and regular languages $L(A_{1/2})$ respectively, the intersection operation is closed. For context-free languages, the intersection operation is not closed. A product operation cannot properly be defined on DPDAs.
To alleviate this problem, the plant model will remain regular and only the specification will be deterministic context-free. This ensures that the product language will always remain within the correct scope, because the intersection between a regular and a deterministic context-free language always yields a deterministic context-free language.
This section presents an algorithm and concepts to compute the supremal controllable sublanguage from a regular plant and a deterministic context-free specification. It aims to provide an intuitive approach to understanding, rather than giving complete and exact definitions of the algorithm, which can be found in [5].

3.2.1 The Product Operation

The product operation is not much different from the product defined in section 3.1.1. The product automaton $A_\times$ of a DFA A_r and DPDA A_{cf} is defined as

$$A_\times = A_r \times A_{cf} = (Q_r \times Q_{cf}, \Sigma_r \cup \Sigma_{cf}, \delta_\times, (q_{0_r}, q_{0_{cf}}), Q_{m_r} \times Q_{m_{cf}}, \Gamma, \Box) \tag{3.18}$$

with

$$\begin{aligned} ((p_r, p_{cf}), a, u, v, (q_r, q_{cf})) \in \delta_\times \qquad & \text{if } (p_r, a, q_r) \in \delta_r \wedge (p_{cf}, a, u, v, q_{cf}) \in \delta_{cf} \\ ((p_r, p_{cf}), \lambda, u, v, (p_r, q_{cf})) \in \delta_\times \qquad & \text{if } (p_{cf}, \lambda, u, v, q_{cf}) \in \delta_{cf} \end{aligned} \tag{3.19}$$

Note that the main difference is the need to allow for the DPDA's lambda transitions. Transitions with lambda events cannot occur in DFAs, because they would either be redundant or make the automaton nondeterministic. This is not the case for DPDAs, because the transition depends on the stack top and thus cannot always occur.

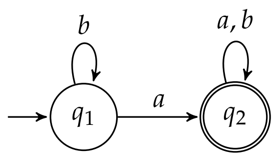


Figure 3.6: Automaton A_r

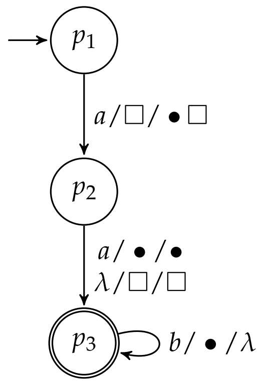


Figure 3.7: Automaton A_{cf}

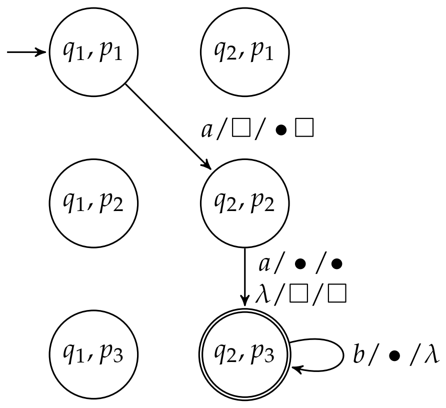


Figure 3.8: Product automaton $A_r \times A_{cf}$

3.2.2 Closed Loop Computation

Computation of the closed loop is similar in structure, but much more complex than the algorithm presented in 3.1.2. It involves the following steps, whose operations are explained in the following sections in more detail.

Step 1 Let the DFA P be the plant and the DPDA S be the specification. Compute the product $S \times P$ and make it nonblocking. The nonblocking operation is comparable to the coaccessible operation, but works differently.

$$O \leftarrow S \times P$$
$$O \leftarrow \text{Nonblock}(O)$$

Step 2 In step two, the product of the current candidate for the closed loop O and the plant P is calculated again and then made accessible

$$O' \leftarrow O \times P$$
$$O' \leftarrow \text{Ac}(O')$$

The next step is to remove noncontrollable states as to satisfy the implementability requirement 3.9. The problem with DPDAs is that controllability does not only depend on the events, but also on the stack top symbol. Consider an uncontrollable event in a state where it must be allowed and a transition that allows said event. Since the uncontrollable event is bound to the transition, it is also bound to the top stack symbol that the transition requires. It is not just the state that allows the uncontrollable event, but the combination of the state and the top stack symbol. Therefore, combinations of this state and other stack symbols must be prevented, because they violate the implementability condition.
For an example, refer to figures 3.9 to 3.11 with $\Sigma_{uc} = \{b\}$, $\Gamma = \{\bullet, \square\}$ and the plant state p_1 allowing b. The product state (q_1, p_1) must allow b as well and it does, but only with $\square$ as the top stack symbol. To solve the problem, the state is split up and two new states, called "ears", are added to the original "head" state (q_1, p_1). The ears account for the different top stack symbols. Only the $\bullet$-ear has a controllability problem and must be deleted.

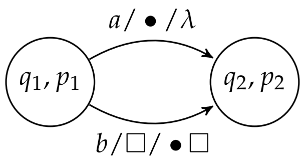

$$\begin{tikzcd}
q_1,p_1 \arrow[r, bend left, "a/\bullet/\lambda"] \arrow[r, bend right, "b/\square/\bullet\square"'] & q_2,p_2
\end{tikzcd}$$

Figure 3.9: Product automaton with $\Sigma_{uc} = \{b\}$, $\Gamma = \{\bullet, \square\}$ and the plant state p_1 allowing b

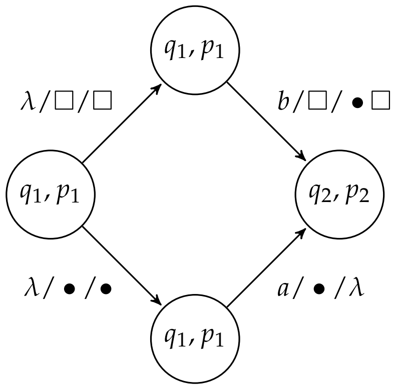

$$\begin{tikzcd}
 & q_1,p_1 \arrow[dr, "b/\square/\bullet\square"] & \\
q_1,p_1 \arrow[ur, "\lambda/\square/\square"] \arrow[dr, "\lambda/\bullet/\bullet"'] & & q_2,p_2 \\
 & q_1,p_1 \arrow[ur, "a/\bullet/\lambda"'] &
\end{tikzcd}$$

Figure 3.10: Split product automaton

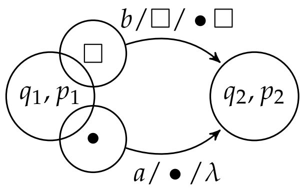


Figure 3.11: Short-hand form for the split automaton

The algorithm employs the Split() function to split all states into heads and ears. Ruls() serves to remove useless states, i. e. all ears that have no outgoing transition. Marked ears are not deleted if they are reachable, even if they have not outgoing transition. Finally, Rnce() will delete all ears that have a controllability problem.

$$\begin{aligned}
O' &\leftarrow \text{Split}(O') && \textit{Split into heads and ears} \\
O' &\leftarrow \text{Ruls}(O') && \textit{Remove useless states} \\
O'' &\leftarrow \text{Rnce}(O') && \textit{Remove noncontrollable ears}
\end{aligned}$$

Removing noncontrollable ears may introduce blocking behavior to the system. If states were removed in the previous step, any potential blocks need to be removed. This step is analogous to making the system coaccessable. If no states were removed, there was no controllability problem in the first place and all changes in this step were unnecessary.

$$(|O''.Q| \neq |O'.Q|) \Rightarrow R \leftarrow \text{Nonblock}(O'')$$
$$(|O''.Q| = |O'.Q|) \Rightarrow R \leftarrow O$$

Step 3 Compare the number of states of R to the number of states of O. If no states were removed, the algorithm finishes with the result O. If states were removed, set O to R and continue at Step 2.

$$(|R.Q| = |O.Q|) \Rightarrow \text{Finished}$$
$$(|R.Q| \neq |O.Q|) \Rightarrow O \leftarrow R$$

3.2.3 The Accessible Operation

Making a DPDA O accessible follows the same basic concepts as making a DFA accessible. If a state is not reachable from the initial state, the state can be deleted. If a transition is not reachable from the intial state, the transition can be deleted. In essence, it looks like the following:

$$\begin{aligned} &\forall q \in O.Q : (\neg \mathrm{Ts}(q) \Rightarrow O.Q \leftarrow O.Q \backslash \{q\}, && \textit{Test } q \textit{ and delete if not reachable} \\ &\forall e = (q, a, u, v, q') \in O.\delta : \neg \mathrm{Te}(e) \Rightarrow O.\delta \leftarrow O.\delta \backslash \{e\}) && \textit{Test } e \textit{ and delete if not reachable} \end{aligned}$$

This introduces two new operations, Ts() (Test Transition) and Te() (Test Edge). Both make use of the previously mentioned Nonblock().

Ts To test a state q for reachability, it is set as the only marked state of the DPDA C, which is a copy of O. Then C is made nonblocking and the marked states are examined. If the state q was not reachable, every state of C will have been blocking, all useless states (including q) are deleted by Nonblock() and the set of marked states $C.Q_m$ is empty.

$$\begin{aligned} C &\leftarrow O \\ C.Q_m &\leftarrow \{q\} \\ \mathrm{Nonblock}(C).Q_m = \emptyset &\Rightarrow \text{not reachable} \\ \mathrm{Nonblock}(C).Q_m \neq \emptyset &\Rightarrow \text{reachable} \end{aligned}$$

Figures 3.12 to 3.14 provide an example. q_3 can only be reached from q_2, which can only be reached with $\bullet$ as a stack top. The transition from q_2 to q_3 requires $\square$ as a stack top and can never be taken, thus q_3 is not reachable.

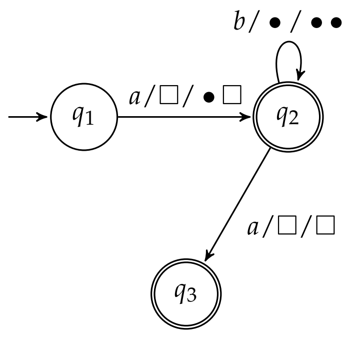


Figure 3.12: DPDA O whose state q_3 will be tested for reachability

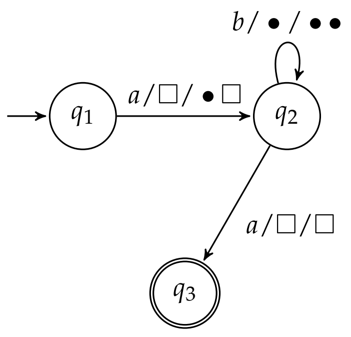


Figure 3.13: DPDA C with only q_3 set as marked

Figure 3.14: Because q_3 is not reachable, Nonblock(C) is empty and has no marked states

Te To test a transition $e = (q, a, u, v, q')$, a new state (new, q') is introduced to C, which, again, is a copy of O. e is removed from C and replaced by $(q, a, u, u, (\text{new}, q'))$. (new, q') is set as C's only final state and C is made nonblocking. As with Ts(), if $C.Q_m$ is empty, the edge was not reachable.

$$
\begin{aligned}
C &\leftarrow O \\
C.\delta &\leftarrow C.\delta \setminus \{(q,a,u,v,q')\} \\
C.\delta &\leftarrow C.\delta \cup \{(q,a,u,v,(\text{new},q'))\} \\
C.Q &\leftarrow C.Q \cup \{(\text{new},q')\} \\
C.Q_m &\leftarrow \{(\text{new},q')\} \\
\text{Nonblock}(C).Q_m = \emptyset &\Rightarrow \text{not reachable} \\
\text{Nonblock}(C).Q_m \neq \emptyset &\Rightarrow \text{reachable}
\end{aligned}
$$

Figures 3.15 to 3.17 provide an example, which is very similar to the example for Te(). Note that the actual DPDA returned by Nonblock() is likely to look very different than the DPDA shown in Figure 3.17, but will have the same language.

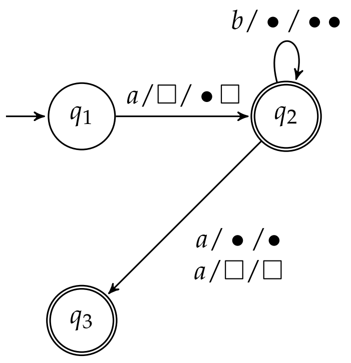


Figure 3.15: DPDA O whose transition $(q_2,a,\bullet,\bullet,q_3)$ will be tested for reachability

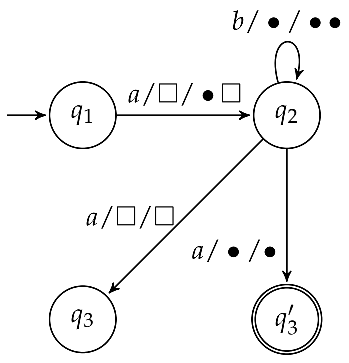


Figure 3.16: DPDA C with new state q_3' and only q_3' set as marked

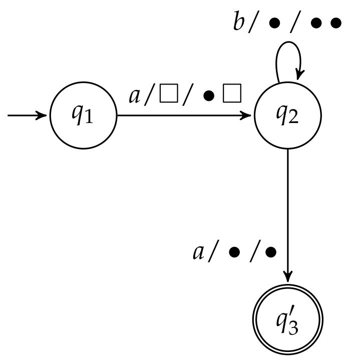


Figure 3.17: Because $(q_2,a,\bullet,\bullet,q_3')$ is reachable, Nonblock(C) is not empty and has marked states

3.2.4 The Nonblock Operation

The nonblock operation is the largest and most complex part of the algorithm. It leaves the scope of automatons altogether, extends into grammars and parsers, only to come back to DPDAs. An overview of the steps is given in Figure 3.18.

The aim of the nonblock operation is to convert the initial DPDA into a DPDA that is nonblocking, i. e. has no states that do not lead to a marked state. When converting the automaton to a grammar, blocking states create productions that are not reachable. Unreachable grammar productions are easier to recognize than unreachable transitions, because stack symbols are no longer an issue. As such, the unreachable productions can easily be deleted.

What remains is a reconversion from the grammar to an automaton via a parser. Due to the different conversions between the language representations automaton, grammar and parser, preservation of the initial DPDA's structure is unlikely. The marked language, however, does not change.

Nonblock is similar to the accessible operation in that its result automaton is always accessible. The difference is that the accessible operation preserves the automaton's structure while nonblock does not. This is an important distinction and the reason why accessible is used in step 2 instead of nonblock.

As before, the following sections will aim to provide only an intuitive understanding of the nonblock algorithm, because a complete explanation is beyond the scope of this thesis. Refer to [5] for a more in-depth view.

Consider the automaton shown in Figure 3.19. Its state q_2 is blocking, because it can only be entered with $\bullet$ as a stack top symbol, but requires $\square$ for leaving it. Blockingness is easy to see in this case and gets intuitively resolved by deleting q_2. This is similar to the coaccessible operation for DFAs, where blocking states were deleted as well.

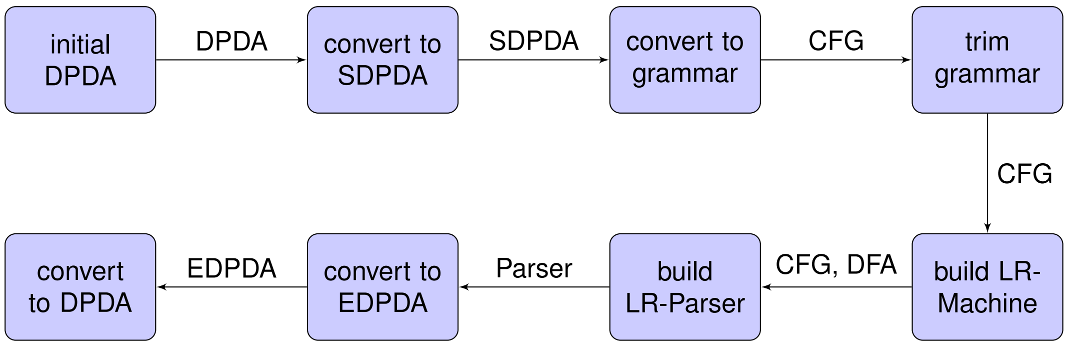


Figure 3.18: Overview of the nonblock operation

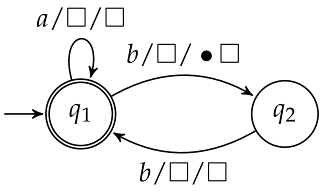


Figure 3.19: DPDA M with q_2 blocking

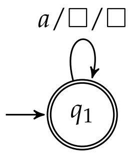


Figure 3.20: solved nonblock problem by deleting q_2

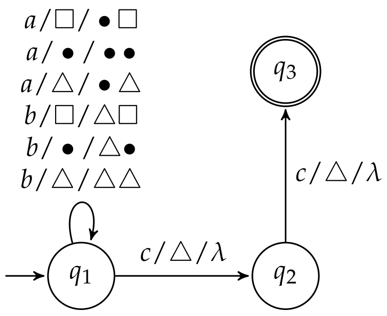

$a/\square/\bullet\square$
$a/\bullet/\bullet\bullet$
$a/\triangle/\bullet\triangle$
$b/\square/\triangle\square$
$b/\bullet/\triangle\bullet$
$b/\triangle/\triangle\triangle$
q_3
$c/\triangle/\lambda$
q_1
$c/\triangle/\lambda$
q_2

Figure 3.21: DPDA with q_2 blocking, but only when the stack top is not $\triangle\triangle$ after leaving q_1

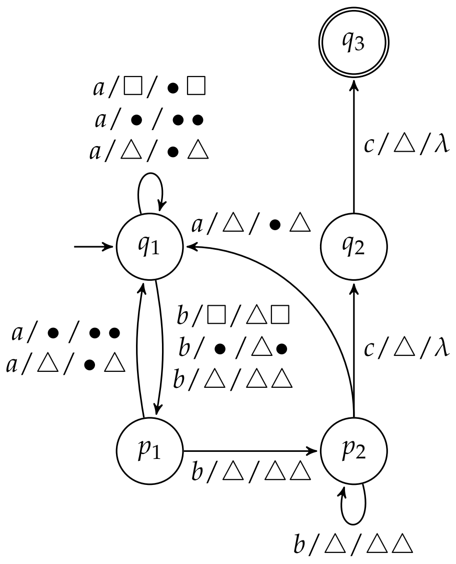

q_3
$a/\square/\bullet\square$
$a/\bullet/\bullet\bullet$
$a/\triangle/\bullet\triangle$
$c/\triangle/\lambda$
q_1
$a/\triangle/\bullet\triangle$
q_2
$b/\square/\triangle\square$
$b/\bullet/\triangle\bullet$
$b/\triangle/\triangle\triangle$
$a/\bullet/\bullet\bullet$
$a/\triangle/\bullet\triangle$
$c/\triangle/\lambda$
p_1
p_2
$b/\triangle/\triangle\triangle$
$b/\triangle/\triangle\triangle$

Figure 3.22: solved nonblocking problem by ensuring the correct stack when entering q_2

As a more complicated example, consider Figure 3.21. Again, state q_2 is blocking, it depends on the events happening in q_1. It is only blocking if q_1 is left with a stack top other than $\triangle\triangle$. q_2 cannot just be deleted, as was the intuition from the coaccessible operation. Instead, it must be ensured that q_2 can only be entered with a nonblocking stack. This requires adding states and transitions instead of deleting them. In general, the cause of blocking of a state q does not lie with q, which requires remodelling of the automaton. The following sections illustrate this remodelling process using the example automaton M of Figure 3.19.

From DPDA to CFG

According to Knuth [9], who also provides an algorithm, every deterministic DPDA can be converted into a CFG. This CFG is parsable by an LR(1) parser and therefore called an LR(1) grammar. To start the transformation into a grammar, the DPDA must be converted to an SDPDA. SDPDA transitions, by definition, fulfill only a single role. They can either be reading, popping or pushing (see section 2.2.2). The SDPDA S corresponding to the input DPDA M from Figure 3.19 is shown in Figure 3.23.

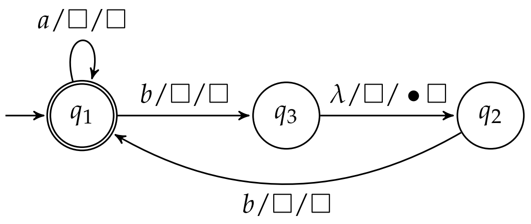


Figure 3.23: M converted to an SDPDA S

The SDPDA S can then be translated into a context-free grammar G. The grammar's terminals correspond to the SDPDA's alphabet, $G.\Sigma = S.\Sigma$. The nonterminals are 2-tuples and 3-tuples, $G.N = (S.Q \times G.\Sigma) \cup (S.Q \times G.\Sigma \times S.Q)$. The start nonterminal is $G.q_0 = (S.q_0, S.\Box)$. The grammar productions $G.P$ are derived from the transitions $S.\delta$ and the marked states $S.Q_m$:

$$\begin{aligned}
&\forall q_i \in S.Q_m, \forall B \in S.\Gamma && :((q_i, B) \to \lambda) \in G.P \\
&\forall e = (q_i, a, B, B, q_j) \in \delta_{\text{Read}} && :((q_i, B) \to (q_j, B)) \in G.P \\
&\forall e = (q_i, a, B, B, q_j) \in \delta_{\text{Read}}, \forall q_t \in S.Q && :((q_i, B, q_t) \to a(q_j, B, q_t)) \in G.P \\
&\forall e = (q_i, \lambda, B, \lambda, q_j) \in \delta_{\text{Pop}} && :((q_i, B, q_j) \to \lambda) \in G.P \\
&\forall e = (q_i, \lambda, B, CB, q_j) \in \delta_{\text{Push}} && :((q_i, B) \to (q_j, C)) \in G.P \\
&\forall e = (q_i, \lambda, B, CB, q_j) \in \delta_{\text{Push}}, \forall q_s \in S.Q && :((q_i, B) \to (q_j, C, q_s)(q_s, B)) \in G.P \\
&\forall e = (q_i, \lambda, B, CB, q_j) \in \delta_{\text{Push}}, \forall q_s, q_t \in S.Q && :((q_i, B, q_t) \to (q_j, C, q_s)(q_s, B, q_t)) \in G.P
\end{aligned} \tag{3.20}$$

For the example S, this amounts to 26 grammar productions with the start symbol being $(q_1, \Box)$, as shown in Table 3.1.

$(q_1,\bullet) \to \lambda$	$(q_2,\Box) \to b(q_1,\Box)$	$(q_3,\Box,q_1) \to (q_2,\bullet,q_2)(q_2,\Box,q_1)$
$(q_1,\Box) \to \lambda$	$(q_2,\Box,q_1) \to b(q_1,\Box,q_1)$	$(q_3,\Box,q_1) \to (q_2,\bullet,q_3)(q_3,\Box,q_1)$
$(q_1,\Box) \to a(q_1,\Box)$	$(q_2,\Box,q_2) \to b(q_1,\Box,q_2)$	$(q_3,\Box,q_2) \to (q_2,\bullet,q_1)(q_1,\Box,q_2)$
$(q_1,\Box) \to b(q_3,\Box)$	$(q_2,\Box,q_3) \to b(q_1,\Box,q_3)$	$(q_3,\Box,q_2) \to (q_2,\bullet,q_2)(q_2,\Box,q_2)$
$(q_1,\Box,q_1) \to a(q_1,\Box,q_1)$	$(q_3,\Box) \to (q_2,\bullet)$	$(q_3,\Box,q_2) \to (q_2,\bullet,q_3)(q_3,\Box,q_2)$
$(q_1,\Box,q_1) \to b(q_3,\Box,q_1)$	$(q_3,\Box) \to (q_2,\bullet,q_1)(q_1,\Box)$	$(q_3,\Box,q_3) \to (q_2,\bullet,q_1)(q_1,\Box,q_3)$
$(q_1,\Box,q_2) \to a(q_1,\Box,q_2)$	$(q_3,\Box) \to (q_2,\bullet,q_2)(q_2,\Box)$	$(q_3,\Box,q_3) \to (q_2,\bullet,q_2)(q_2,\Box,q_3)$
$(q_1,\Box,q_2) \to b(q_3,\Box,q_2)$	$(q_3,\Box) \to (q_2,\bullet,q_3)(q_3,\Box)$	$(q_3,\Box,q_3) \to (q_2,\bullet,q_3)(q_3,\Box,q_3)$
$(q_1,\Box,q_3) \to a(q_1,\Box,q_3)$	$(q_3,\Box,q_1) \to (q_2,\bullet,q_1)(q_1,\Box,q_1)$	

Table 3.1: All grammar productions created from S (reachable and productive productions in grey)

Of those 26 grammar productions, only two are reachable and productive (marked in grey), meaning all their nonterminals can be replaced by terminals. Notice that no production with b is left. The b transition that led to the blocking state in the original automaton has been taken out.

Building the LR Machine and Parser

The process of determining the LR machine and parser will not be outlined here, because example has already been given in section 2.2.3. More information on parser construction, including algorithms, can

be found in Parsing Theory by Sippu and Soisalon-Soininen [10].
The grammar is augmented to

$$\begin{aligned} &S \to \$(q_1, \square)\$ \\ &(q_1, \square) \to \lambda \\ &(q_1, \square) \to a(q_1, \square) \end{aligned} \tag{3.21}$$

The LR machine for this augmented grammar is shown in Figure 3.24. The resulting parser actions are shown in Table 3.2, where the start state is $q_0 = 5$ and the marked state is $q_m = 2$.

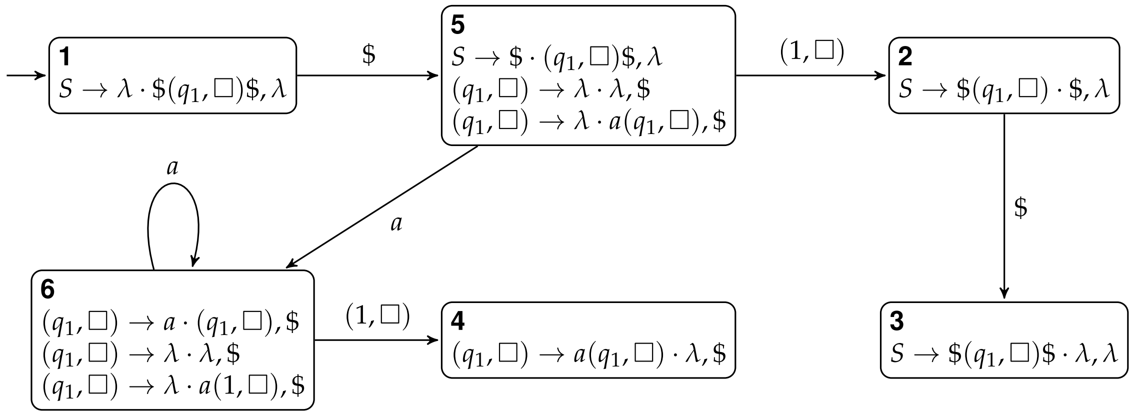

1
$S \to \lambda \cdot \$(q_1,\square)\$, \lambda$
$\$$
5
$S \to \$ \cdot (q_1,\square)\$, \lambda$
$(q_1,\square) \to \lambda \cdot \lambda, \$$
$(q_1,\square) \to \lambda \cdot a(q_1,\square), \$$
$(1,\square)$
2
$S \to \$(q_1,\square) \cdot \$, \lambda$
$\$$
a
a
6
$(q_1,\square) \to a \cdot (q_1,\square), \$$
$(q_1,\square) \to \lambda \cdot \lambda, \$$
$(q_1,\square) \to \lambda \cdot a(1,\square), \$$
$(1,\square)$
4
$(q_1,\square) \to a(q_1,\square) \cdot \lambda, \$$
3
$S \to \$(q_1,\square)\$ \cdot \lambda, \lambda$

Figure 3.24: LR machine for grammar 3.21

Shift actions	Reduce actions	consumed productions
$(5\|a) \to (5,6\|\lambda)$	$(5,6,4\|\$) \to (5,2\|\$)$	$(q_1,\square) \to a(q_1,\square)$
$(6\|a) \to (6,6\|\lambda)$	$(6,6,4\|\$) \to (6,4\|\$)$	$(q_1,\square) \to a(q_1,\square)$
	$(5\|\$) \to (5,2\|\$)$	$(q_1,\square) \to \lambda$
	$(6\|\$) \to (6,4\|\$)$	$(q_1,\square) \to \lambda$

Table 3.2: Shift and reduce actions of the resulting parser, $q_0 = 5$, $q_m = 2$

From Parser to DPDA

A new EDPDA R is constructed from the parser P and the LR machine M as follows:

$$\begin{aligned} &R.Q \leftarrow P.N \times (P.\Sigma \cup \{\lambda\}) \\ &R.q_0 \leftarrow (P.q_0, \lambda) \\ &R.Q_m \leftarrow \{(P.q_m, \lambda)\} \\ &R.\Sigma \leftarrow P.\Sigma \\ &R.\Gamma \leftarrow P.N \cup \{0\} \\ &R.\square \leftarrow 0 \end{aligned} \tag{3.22}$$

The transitions are a result of the parser actions. For every shift action $A|y \to AB|\lambda$, add the transitions

$$\begin{aligned} &R.\delta \leftarrow R.\delta \cup \{((A,\lambda), y, \lambda, A, (B,\lambda))\} \\ &R.\delta \leftarrow R.\delta \cup \{((A,y), \lambda, \lambda, A, (B,\lambda))\} \end{aligned} \tag{3.23}$$

For every reduce action $A|y \to AB|y$, add the transitions

$$\begin{aligned} &R.\delta \leftarrow R.\delta \cup \{((A,\lambda), y, \lambda, A, (B,y))\} \\ &R.\delta \leftarrow R.\delta \cup \{((A,y), \lambda, \lambda, A, (B,y))\} \end{aligned} \tag{3.24}$$

For every reduce action $AwC|y \to AB|y$, add the transitions

$$\begin{aligned} R.\delta &\leftarrow R.\delta \cup \{((C,\lambda), y, \lambda, w^{-1}A, (B,y))\} \\ R.\delta &\leftarrow R.\delta \cup \{((C,y), \lambda, \lambda, w^{-1}A, (B,y))\} \end{aligned} \tag{3.25}$$

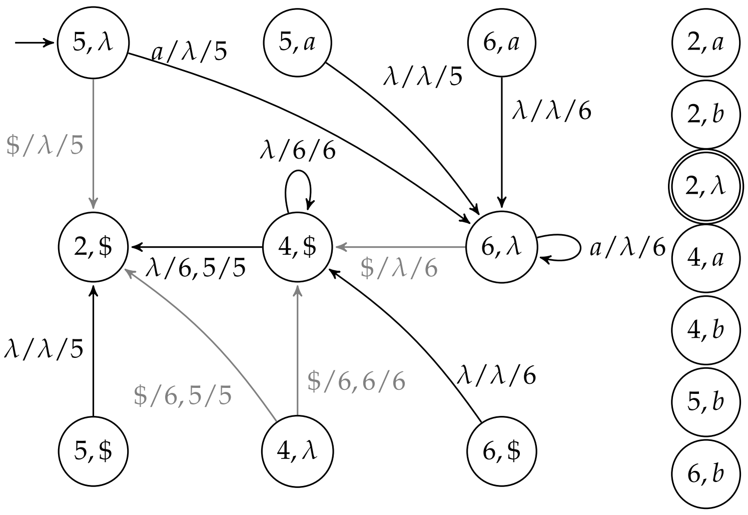


Figure 3.25: EDPDA constructed from parser ($-read transitions marked grey)

The final result of the transformation can be seen in Figure 3.25. This automaton is an EDPDA, because it has transitions popping λ and transitions popping more than one stack symbol at once. Because only the unaugmented grammar is important, states derived from the LR machine states 1 and 3 have been omitted. They model parser states before the first and after the last $, which do not belong to the original grammar.

Every state is annotated with a symbol (an event) $\in \Sigma_{Aug}$. This symbol is used to model the parser's ability to look one symbol ahead without consuming the symbol. This is necessary because the EDPDA cannot look ahead and must consume any symbol it sees. If a reduce action is executed in the EDPDA, the action's lookahead symbol $s \in \Sigma_{Aug}$ is consumed. It cannot be "unread". The information on the current lookahead is stored in the annotation of a state. Reduce actions, which never consume the lookahead symbol in the parser, always lead to an EDPDA state annotated with a symbol other than the lookahead symbol. Shift actions, which always consume the lookahead symbol, always "delete" the annotated information by leading to a state annotated with λ.This means that once a state annotated with s is entered, only s-annotated states must a be entered subsequently until an s-consuming shift action has been executed.

Having transitions that read $ (marked in grey), the example automaton still includes parts of the augmented grammar. The $ event marks the end of a word of the original language. Deleting all $ transitions and setting their starting state as marked will revert the automaton's language to the original language.

The next step is to remove all useless states. States are useless if they have no transition leading to them, regardless of required stack operations. Removing useless states is basically the same as the accessible operation for DFAs. All pop/push properties of the transitions are ignored and the DPDA is made DFA-accessible. The result can be seen in Figure 3.27.

Lastly, the EDPDA must be transformed back into a DPDA. This entails remodelling transitions that do not pop exactly one stack symbol. In the example, there are no transitions that pop two or more stack symbols. They could be removed by splitting them up into one transition for each pop symbol and inserting intermediary states.

What is in the example are transitions (q, a, λ, v, q') that pop nothing (only λ). These transitions are deleted. Instead, transitions (q, a, u, vu, q') are inserted for every $u \in R.\Gamma$. Every u needs to be considered, because it is generally unknown which stack symbols can be the top stack symbol in state q.

If there are transitions $(q, a, \lambda, \Box, q')$, this method is impossible, because there cannot be any stack symbol beneath the stack bottom symbol. To make sure this will never be the case, a new stack bottom

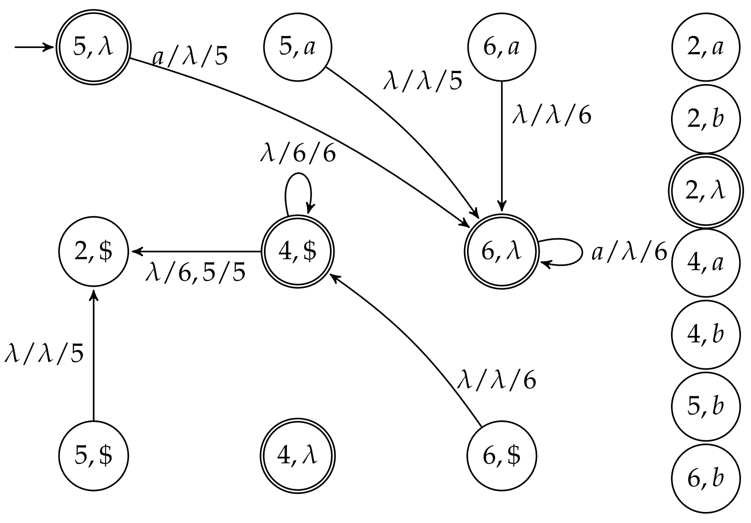


Figure 3.26: EDPDA without $ transitions

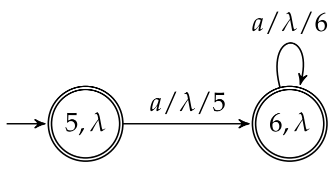


Figure 3.27: EDPDA after removing useless states

symbol is inserted. A new initial state and a new transition are inserted before the old initial state to push the old stack bottom onto the new stack bottom, see Fig. 3.28.

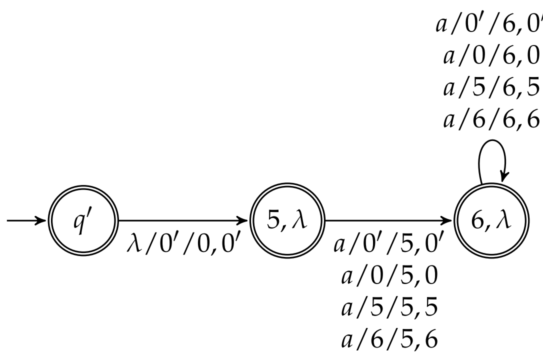


Figure 3.28: converted into DPDA with new stack bottom $0'$

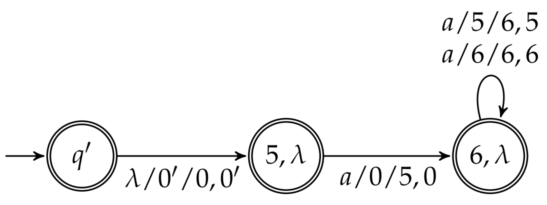


Figure 3.29: after basic transition optimization

Figure 3.29 shows an optimized version with less transitions. Transitions can be cut down if a state does not have incoming read and pop transitions and has only incoming push transitions that push the stack symbols $s \in S \subseteq \Gamma$. All outgoing transitions that require the stack top symbol $s' \notin S$ can be deleted. This completes the nonblock operation. The DPDA has the same marked language as before, but no states are blocking.

3.3 Summary

This chapter has given a basic introduction to supervisory control theory as introduced by Ramadge and Wonham [4]. Blockingness was explained, as was Accessibility and Coaccessibility, two operations that together make a DPA nonblocking. If synthesising a controller and computing the closed loop, nonblockingness is required. Furthermore, the controller must be implementable, with the condition being $\overline{K}\Sigma_{uc} \cap L(P) \subseteq \overline{K}$, and the closed loop should represent the minimally restrictive sublanguage $K^{\uparrow C}$.
All these conditions still apply to supervisory control theory if the specification used is deterministic context-free instead of regular. However, the algorithms involved are more complex. Accessibility and controllability are not just a property of a state, but a property of a pair of state and stack top symbol. The nonblock operation makes a DPDA nonblocking, but has to convert the DPDA into a grammar to cut away the blocking parts. Converting the grammar back to a DPDA necessitates the construction of a parser and a LR machine.

Chapter 4

Realization

4.1 Libfaudes

Libfaudes [1] is a framework for working on discrete event systems with a control theoretic approach. It is written in C++ and is available under the GNU Lesser General Public License [3].
Libfaudes can be extended via plugins to provide additional functionalities. In its most basic form with only the CoreFaudes plugin activated, it supports DPAs and operations on DPAs such as accessibility and coaccessibility. Numerous other plugins extend Libfaudes with controllability and observability of events, supremal controller generation, timed transitions and more.

4.1.1 Framework Structure

The architecture of Libfaudes can, as stated in the API, be split into six major parts:

- *generator classes* for representing automatons,
- *container classes* for events, states and transitions,
- *functions* for implementing algorithms,
- *plugins* for extending and deriving the former three,
- the *runtime interface* for Lua support, and
- *token-IO* for serialization

Automatons in Libfaudes are called “generators” and represent DPAs $(Q, \Sigma, \delta, q_0, Q_M)$. The base generator from which every other generator is derived is the `vGenerator`. Other generator classes like `TaGenerator` and `TcGenerator` extend the base generator and offer further extension points.
Container classes, mainly sets and vectors, are not only for storing events, states and transitions, but can also extend the behavior of their contained elements. This is done via templates, as shown in Figure 4.1. The class `TaGenerator` inherits from `vGenerator` and takes four template arguments. These templates are handed over to the the sets of states, transitions, events and to a global generator attribute, effectively enhancing them. As an example, consider the `System` typedef in the class `TcGenerator`, which implements controllability of events. It takes three `AttributeVoid` arguments, which are dummy clases and do not add behavior, and one `AttributeCFlags` argument, which is a class with controllability flags. `System` passes `AttributeCFlags` to the internal set storing the events, thereby attaching controllability flags to each event.
Events as well as states can be referred to by symbolic names, but are internally handled as `Idx` types, which is a typedef for `long unsigned int`. Integers are used for performance reasons, because they scale better than string representations. Translating from symbolic names to integers is handled by Libfaudes through static global symbol tables.
Functions can be implemented as a member of a class or as a standalone function. Functions as well as classes can be made publicly available for end users via the runtime interface, which can connect to

the Lua interpreter luafaudes and the graphical interface DESTool. Both can be found on the Libfaudes homepage.
Plugins typically introduce new data types and algorithms to Libfaudes. They integrate into the global build system, but are modular and can be excluded or included as needed. Although each plugin has its seperate folder, it is important to know that the build system does not recognize this folder structure, but sees every file as if in one big folder. Because of this, every file needs to have a name unique to the whole of Libfaudes.

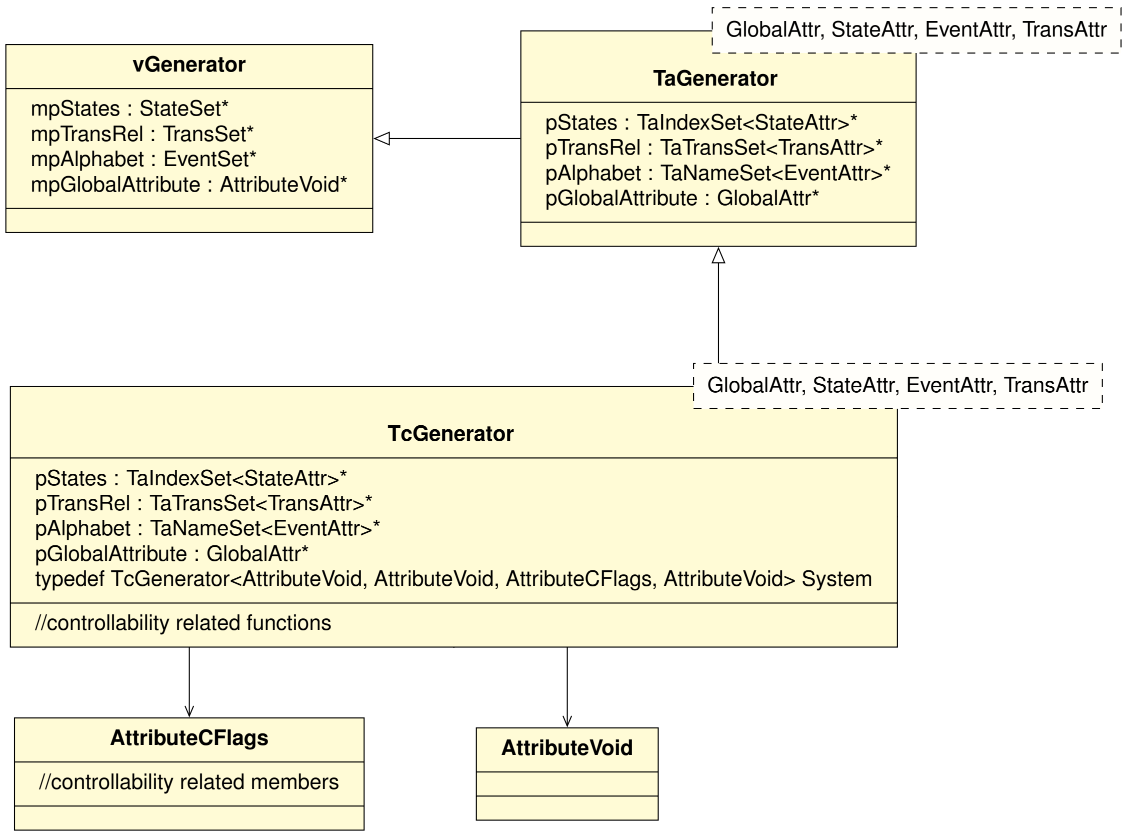

vGenerator
mpStates : StateSet*
mpTransRel : TransSet*
mpAlphabet : EventSet*
mpGlobalAttribute : AttributeVoid*
GlobalAttr, StateAttr, EventAttr, TransAttr
TaGenerator
pStates : TaIndexSet<StateAttr>*
pTransRel : TaTransSet<TransAttr>*
pAlphabet : TaNameSet<EventAttr>*
pGlobalAttribute : GlobalAttr*
GlobalAttr, StateAttr, EventAttr, TransAttr
TcGenerator
pStates : TaIndexSet<StateAttr>*
pTransRel : TaTransSet<TransAttr>*
pAlphabet : TaNameSet<EventAttr>*
pGlobalAttribute : GlobalAttr*
typedef TcGenerator<AttributeVoid, AttributeVoid, AttributeCFlags, AttributeVoid> System
//controllability related functions
AttributeCFlags
//controllability related members
AttributeVoid

Figure 4.1: Base generators of libfaudes and extending mechanism

4.2 Plugin Architecture

It is the goal of this thesis to present an implementation for the algorithm that constructs a controller for a regular plant and a context free specification. Libfaudes has a well-equipped environment to work with DFAs of any kind, but lacks the tools to handle DPDAs. Because introducing DPDAs to libfaudes is, structurally, one of the biggest and most important points, this is done via a new plugin called "push-down". Libfaudes does not offer data structures for grammars and parsers and even though LR machines are DFAs, they are a specific kind of DFA, which does not exist in the required form. Grammars, parsers and LR machine will be implemented as well in the pushdown plugin.
Figure 4.2 shows the class diagram of the EDPDA implementation, which can also be used for DPDAs and SDPDAs. The generator class is called `TpdGenerator` and inherits all required controllability features from `TcGenerator`. All four extension templates for transitions, states, events and the global attribute have been used to define the type `PushdownGenerator`.
The extension for events, `AttributeCFlags`, is just carried over from the `System` typedef.
`AttributePushdownGlobal` introduces the stack symbols Γ and the stack bottom $\Box$ to the generator. They are added as a global attribute, because in the scope of the generator, they are globally available for all transitions and states. As with events and states, stack symbols are given a symbolic name, but

are internally handled as `Idx`.
`AttributePushdownTransition` adds the pop and the push part to each transition. The transitions of an EDPDA defined in 2.2.2 had a pop $u \in \Gamma^*$ and push $v \in \Gamma^*$, but this definition does not translate well into Libfaudes data structures. In Libfaudes, transitions are uniquely identified by their start state q, end state q' and event a. This is not the case with EDPDA transitions, as the transitions $(q, a, \bullet, \square, q')$ is clearly different from $(q, a, \square, \square, q')$. To work around that constraint, pop and push attributes for each transitions are not implemented as a pair $p \in (\Gamma^* \times \Gamma^*)$, but as a set of pairs $P_{\text{PopPush}} \subset (\Gamma^* \times \Gamma^*)$. The aforementioned transitions would form one transitions object $(q, a, \{(\square, \square), (\bullet, \square)\}, q')$. Special care must be taken when iterating over transitions of an EDPDA, because it requires iterating over the pop push pairs of each transition as well.
`AttributePushdownState` extends EDPDA states to hold information about how a certain state was created. This is not so much a requirement of EDPDAs as of the algorithm that is implemented. The algorithm uses a lot of transformations on generators, where new states are created from transitions, are split into heads and ears or are merged from other states. Keeping track of this information allows for easier debugging and in some cases is required as the algorithm needs to look “back”. The `MergeAbstract` class is an extension point for classes that can hold arbitrary information on states.
λ events are not included in Libfaudes, because they are irrelevant for DFAs, but they are, along with λ pops and pushes, important parts of DPDAs that need representation. In theory, λ events could be represented by an empty event string, but Libfaudes does not allow empty events. This necessitates an event uniquely identifiable as λ. Unique identification is not possible via event indices, because indices are run-time generated. Instead, the plugin introduces the string constant `FAUDES_PD_LAMBDA` for identification, which is to be used every time when working with λ events. The same approach, with the same constant, holds true for “empty” stack symbols.
Figure 4.3 shows the class diagram for parsers, grammars and LR machines. The implementation of the `Grammar` class and the `Lr1Parser` class is straightforward and corresponds to the definitions given in sections 2.2.2 and 2.2.3. The `Nonterminal` class can accomodate nonterminals $\in Q \times \Gamma \times Q$ by design and nonterminals $\in Q \times \Gamma$ by setting the second state index to 0. The `Terminal` class is a wrapper class for events $\in \Sigma$.
The LR machine, here called `GotoGenerator`, has its states extended with a set of LR1 configurations and its transitions with grammar symbol pointers. Ideally, transitions would be defined as (q, a, q') with the event $a \in N \cup \Sigma$, but this is not possible, because Libfaudes does not allow redefintion of $a \in \Sigma$. A workaround is to define transitions as (q, a, b, q') with $b \in N \cup \Sigma$, disregard a completely and treat b as the real event of the transition.

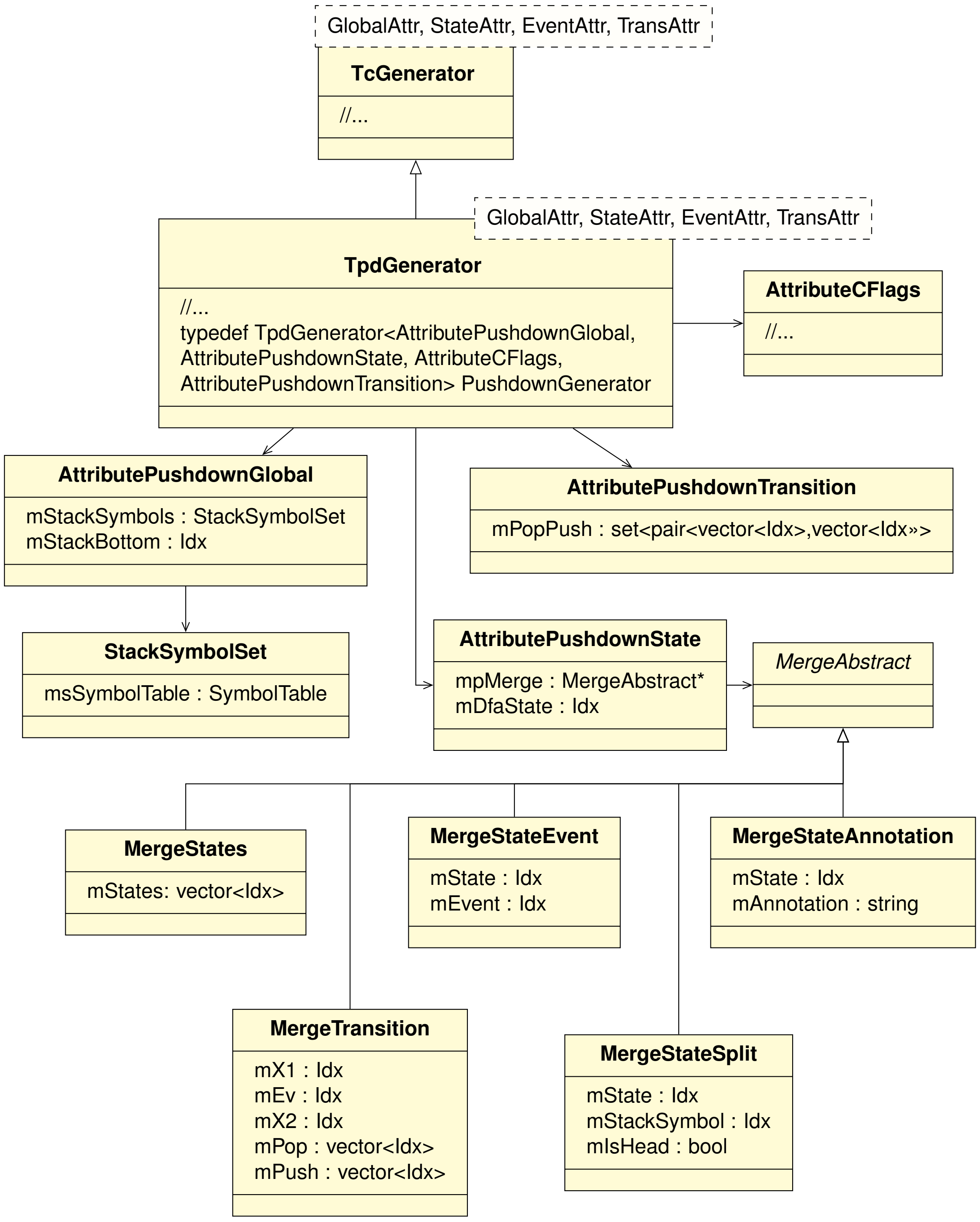


Figure 4.2: class structure for EDPDAs

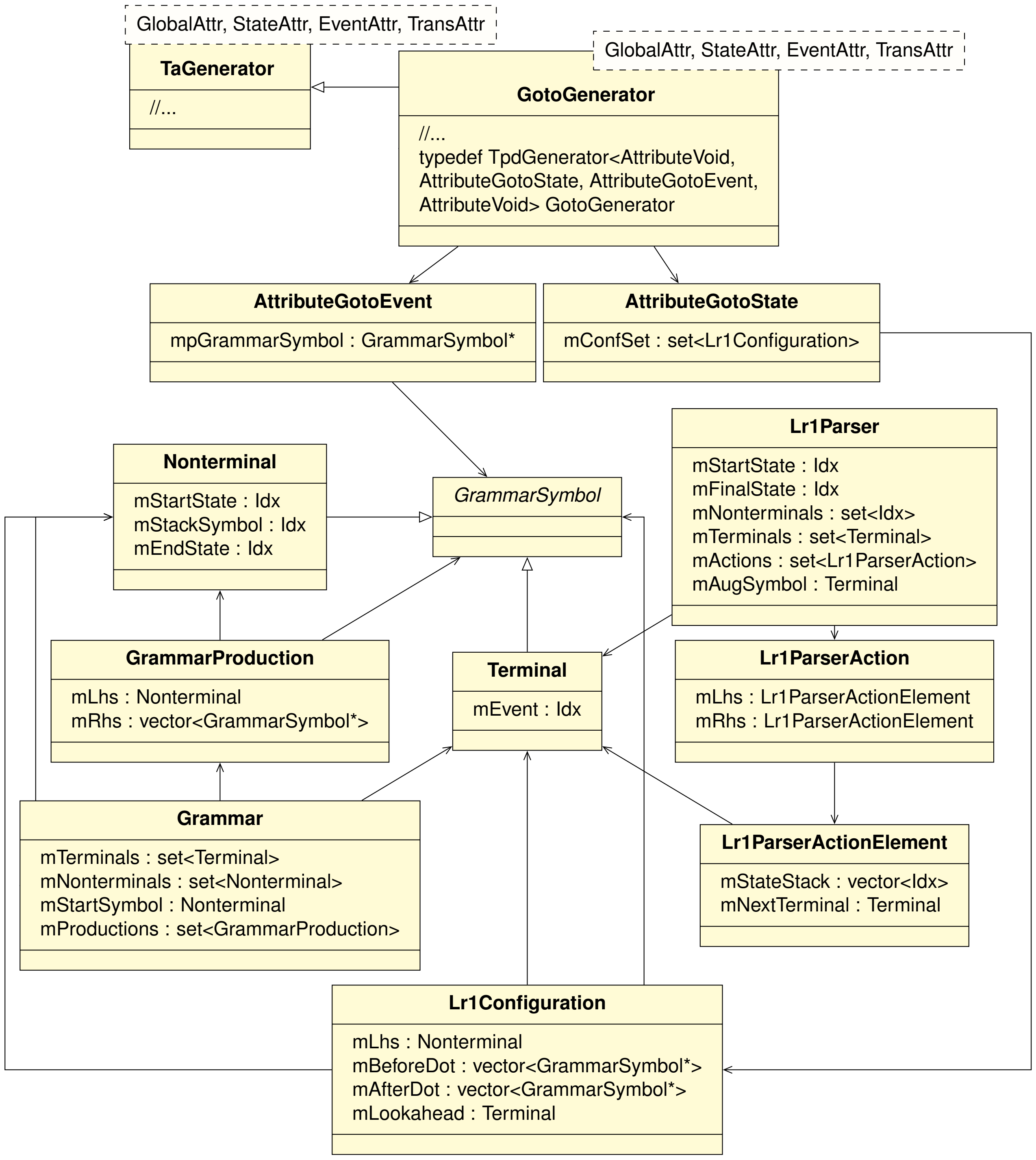

GlobalAttr, StateAttr, EventAttr, TransAttr
TaGenerator
//...
GlobalAttr, StateAttr, EventAttr, TransAttr
GotoGenerator
//...
typedef TpdGenerator<AttributeVoid,
AttributeGotoState, AttributeGotoEvent,
AttributeVoid> GotoGenerator
AttributeGotoEvent
mpGrammarSymbol : GrammarSymbol*
AttributeGotoState
mConfSet : set<Lr1Configuration>
Lr1Parser
mStartState : Idx
mFinalState : Idx
mNonterminals : set<Idx>
mTerminals : set<Terminal>
mActions : set<Lr1ParserAction>
mAugSymbol : Terminal
Nonterminal
mStartState : Idx
mStackSymbol : Idx
mEndState : Idx
GrammarSymbol
GrammarProduction
mLhs : Nonterminal
mRhs : vector<GrammarSymbol*>
Terminal
mEvent : Idx
Lr1ParserAction
mLhs : Lr1ParserActionElement
mRhs : Lr1ParserActionElement
Grammar
mTerminals : set<Terminal>
mNonterminals : set<Nonterminal>
mStartSymbol : Nonterminal
mProductions : set<GrammarProduction>
Lr1ParserActionElement
mStateStack : vector<Idx>
mNextTerminal : Terminal
Lr1Configuration
mLhs : Nonterminal
mBeforeDot : vector<GrammarSymbol*>
mAfterDot : vector<GrammarSymbol*>
mLookahead : Terminal

Figure 4.3: class structure for grammars, parsers and LR machines

The overall file structure of the plugin is shown in Figure 4.4. The two makefiles `Makefile.plugin` and `Makefile.tutorial` are responsible for integrating the plugin source code and the tutorial source code into the build system. In the case of the pushdown plugin, `Makefile.tutorial` is used to compile and run unit tests. All tests are kept in the `tutorial` folder as well. Data structures and algorithms are located in the `src` folder. In `doxygen`, pictures for the end user documentation are stored, while `registry` holds files that pass functions and data types to the Libfaudes runtime interface. `pd_definitions.rti` defines functions available to the end user via Lua and `pd_interface.rti` does the same for the type `PushdownGenerator`. `pushdown_index.fref` documents the plugin for the end user.

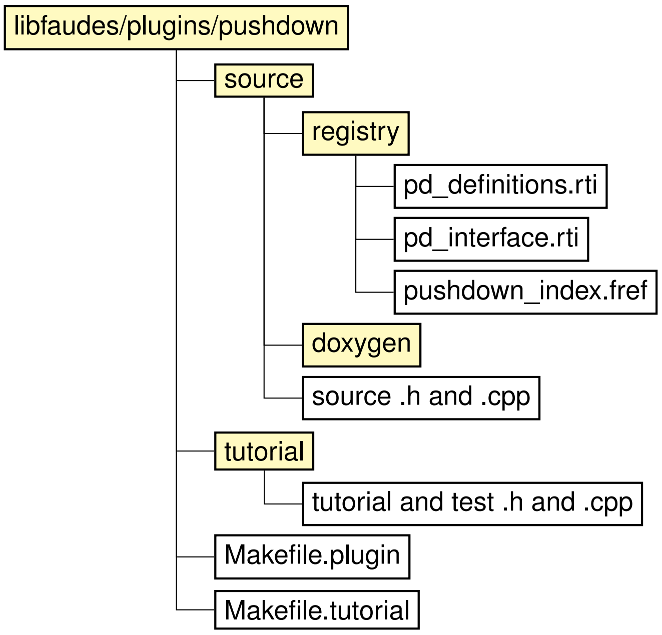


Figure 4.4: file structure of the plugin

4.3 Supervisor Algorithm

The algorithm to compute the supervisor is comprised of over 56 subfunctions. Due to this size, the subfunctions are split up into different files, all of which are prefixed with `pd_alg_*`. Functions are loosely grouped by dependencies and context within each file.
To make comprehension and understanding of the code as easy as possible, the implementation is closely oriented on the algorithm structure presented in [5]. For most of the functions, a pseudocode counterpart can be found in [5], although the order of loops and iterations may vary, as can the naming and assignment order of variables and the execution order of functions.
As shown in section 3.2.2, the algorithm consists of three basic steps:

1. Construct the product O of plant P and specification S and make it nonblocking
2. Construct the product of O' of O and P, make it accessible and controllable, and make it nonblocking if ears were removed
3. Reiterate, if ears were removed and nonblock changed the number of states

These three steps can be rearranged into three top level functions that call each other, which are called Cc() (Construct controller, step 1), Ccs() (Construct controller single, step 2) and Ccl() (Construct controller loop, step 3). Together, they form the entry point into the algorithm. Their dependencies are shown in Figure 4.6. Cc() is the top level function and calls the loop Ccl(). The loop keeps calling Ccs() until the

controller is found. Other functions are called in the process as well, some of which have already been mentioned.
The following sections will detail the subfunctions of the algorithm as they appear in [5], grouped by files.

4.3.1 Top Level Functions

The file `pd_alg_cc` groups nearly all functions that are at the topmost level of the algorithm, that is, that are directly called by Cc(), Ccl() or Ccs(). Every other function will be a subfunction of Nonblock(). The execution order of functions is shown in Figure 4.5 while the dependencies of the functions are shown in Figure 4.6.

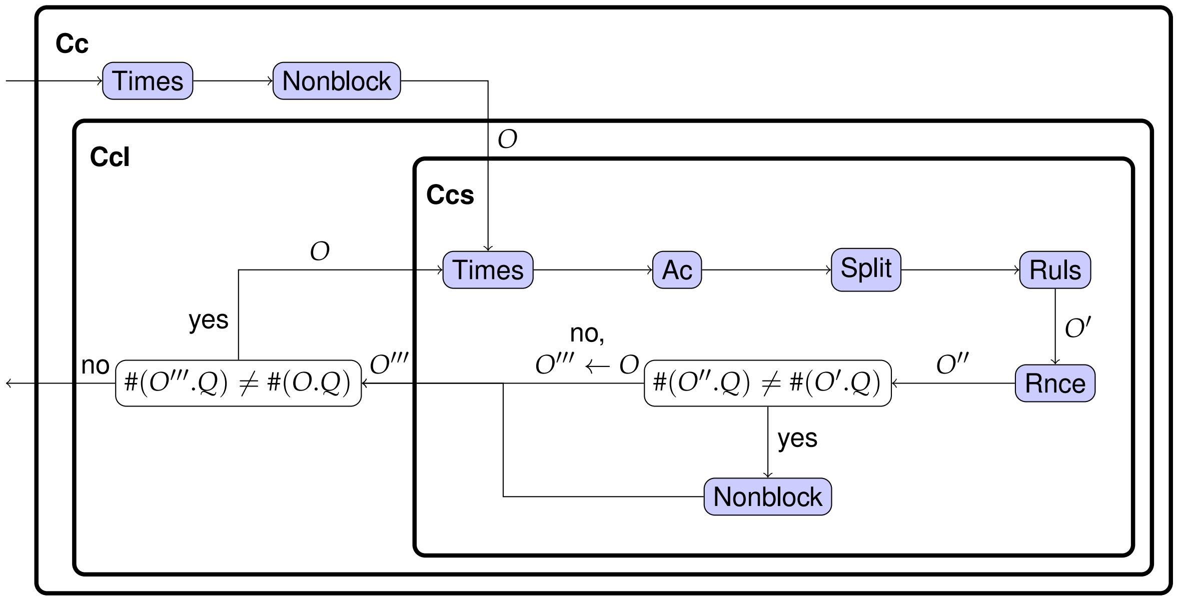


Figure 4.5: Overview of the top-level functions

Cc is the top-level function that is called to start the algorithm. Its input is a regular plant and a context free specification.

Ccl is a loop that checks the termination condition.

Ccs executes one iteration of calculating a minimally restrictive controller.

Times is used to compute the product of a DFA and a DPDA.

Nonblock will transform a generator into a nonblocking generator.

Ac constructs the accessible part of a generator.

Te tests a transition (an edge) for reachability.

Ts tests a state for reachability.

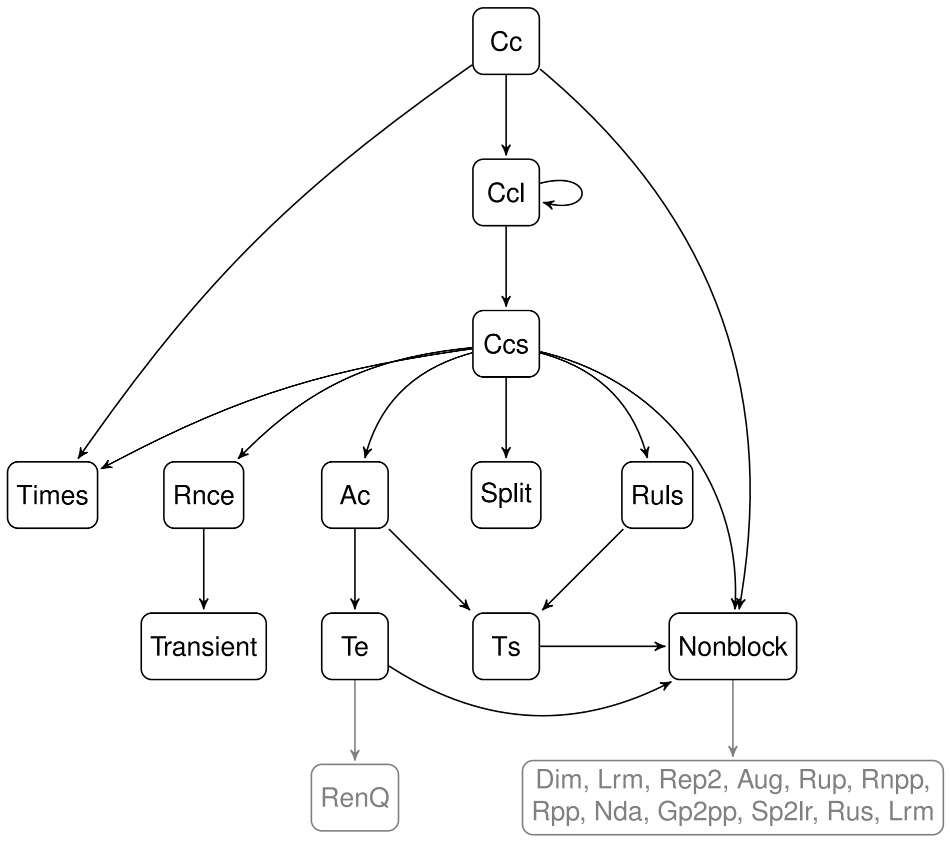

Cc
Ccl
Ccs
Times
Rnce
Ac
Split
Ruls
Transient
Te
Ts
Nonblock
RenQ
Dim, Lrm, Rep2, Aug, Rup, Rnpp, Rpp, Nda, Gp2pp, Sp2lr, Rus, Lrm

Figure 4.6: Dependencies of top-level functions

Split splits a generator into heads and ears. The ears contain information on the current state and the current top stack symbol.

Ruls removes all useless states (ears only) from a generator. An ear is useless if it does not have an outgoing transition. Marked ears without an outgoing transitions are only useless if they are not reachable.

Rnce removes noncontrollable ears to satisfy the implementability requirement of a controller.

Transient is used to filter out ears that have λ read transitions, because such states can never pose a controllability problem.

4.3.2 Nonblock Subfunctions not Related to Parsers and LR Machines

The files `pd_alg_nb_sub_a` and `pd_alg_nb_sub_b` hold subfunctions of Nonblock() that are not immediately related to parsing and Lr machines. Mostly, these functions get called before converting the generator into a grammar and after converting the parser back into a generator. The order of execution is shown in Figure 4.7 and the dependecies are shown in Figure 4.8.

Rpp restricts the generator to transitions that are either pop only, push only, or read only. Effectively, its output is an SDPDA.

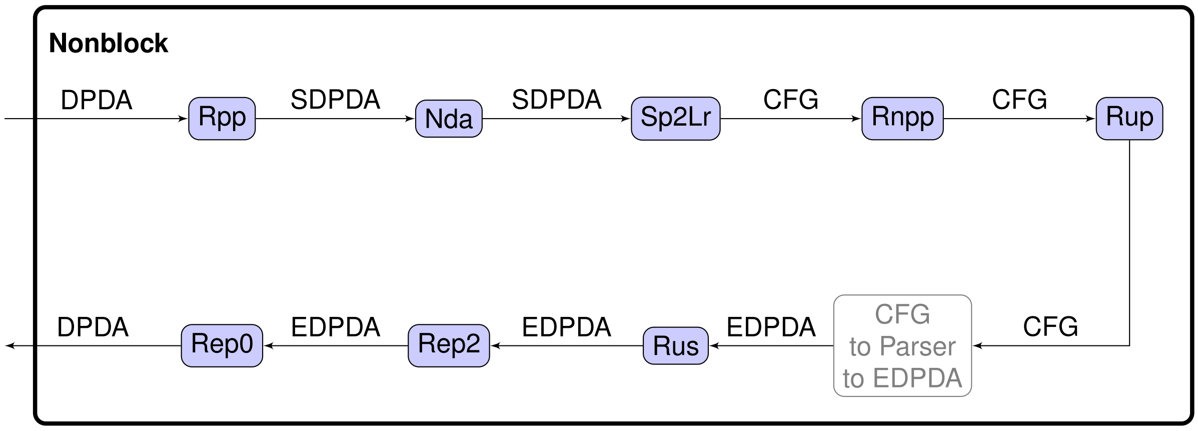


Figure 4.7: Overview of the nonblock operation

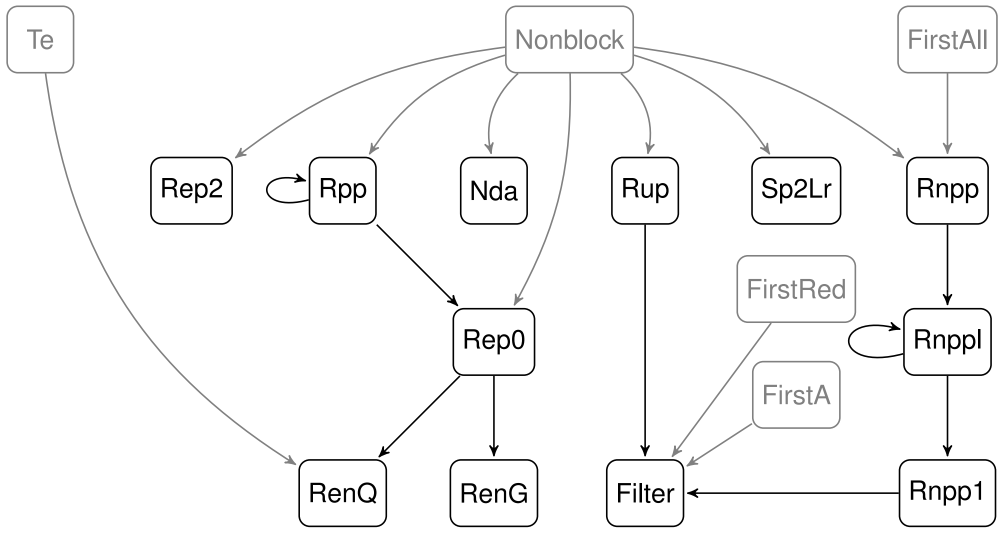


Figure 4.8: Dependencies of nonblock subfunctions

Rep0 removes transitions that pop λ by inserting transitions that pop (and push) every possible stack symbol in its stead. This is required because an SDPDA must not have an empty pop value. Section 3.2.4 offers an example of this process.

RenG renames the stack symbols of the generator, so a new stack bottom symbol can be safely inserted when calling Rep0().

RenQ renames the states of the generator. As with RenG(), this renaming is used to safely insert a new initial state. Although this function is implemented, it is not strictly necessary, because Libfaudes identifies states by their indices instead of their names.

Nda remodells the generator to prevent double (or multiple) acceptance of the same input string. If an event has been read and a marked state is entered, no λ read transition may lead directly or indirectly to another marked state. Only after an event that is not λ has been read, another marked state can be entered. No double acceptance is a requirement for transforming the SDPDA into a CFG (see [9]).

Sp2Lr converts the generator into a context-free grammar (an LR grammar).

Rnpp removes nonproductive productions from the grammar. A production is called nonproductive when it contains nonterminals that can never be eliminated.

Rnppl is a loop that gathers all nonterminals of a grammar that can be eliminated. It only stops when no new eliminable nonterminals can be found.

Rnpp1 looks at each grammar production and decides whether all righthand-side nonterminals can be eliminated, considering the currently known eliminable nonterminals. If all nonterminals on the righthand-side are eliminable, the production can be considered productive and the lefthand-side nonterminal is eliminable as well.

Filter can extract a list of specified symbols from a string of grammar symbols. It is used by Rnpp1() to extract all nonterminals from a production's right-hand side.

Rup removes all unreachable productions from a grammar. A production is unreachable when it cannot be reached from the start symbol.

Rep2 removes all transitions from a generator that pop more than one stack symbol at once. Such generators can occur after the transformation from parser to generator. Together with Rep0() it transforms an EDPDA into a DPDA.

4.3.3 Lr-Machine-Related Functions

The file `pd_alg_lrm` holds all functions that are needed to augment a grammar and construct an LR machine from it. See Figure 4.9 for function dependencies.

Aug augments a grammar with an augment symbol $\$$ and a new start symbol S. Care must be taken to call this function with an appropriate $\$$ and S, because $\$$ must not exist in the grammar's terminals and S must not exist in the grammar's nonterminals.

Lrm creates and LR machine for the augmented grammar.

ValidEmpty determines the initial parser configuration set for a grammar. Descendants of the initial configurations may need to be computed to constitute the whole configuration set. A configuration $A \rightarrow u \cdot v, w$ has descendants, when the grammar symbol string v begins with a nonterminal symbol.

DescInitial determines the initial parser configurations for a grammar.

Desc obtains all descendants of a given configuration set. It calls itself recursively, each time getting the immediate descendants (see Desc11) of the currently found descendants, until no more descendants can be found.

Desc1 obtains immediate descendants for each configuration of a configuration set.

Desc11 obtains all immediate descendants of a given configuration. Immediate descendants only exist for configurations $A \rightarrow u \cdot Bv', w$ that have a nonterminal B after the $\cdot$ symbol. For each grammar production $B \rightarrow x$ there is one immediate descendant $B \rightarrow \lambda \cdot x, y$. This may not yield all possible descendants, because x might still be a string of grammar symbols starting with another nonterminal. Desc() calls Desc1() iteratively to obtain all descendants. y is calculated by FirstLeq1(), which will be explained later.

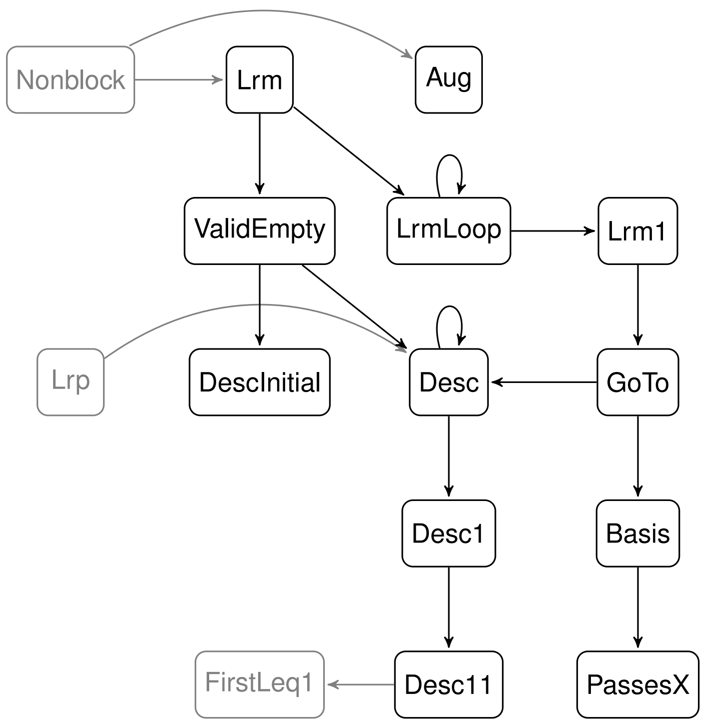


Figure 4.9: Dependencies of LR-machine-related functions

LrmLoop generates all transitions and states (configuration sets) for a complete LR machine. It calls itself recursively until no more transitions and states can be found.

Lrm1 generates new outgoing transitions for a every state of a (probably incomplete) LR machine. The end state of a new transition may be a new state or an already existing one. Transitions are created by trying to shift the $\cdot$ symbol of each configuration in a state q over each symbol $b \in (G.\Sigma \cup G.N)$ of the grammar. If this shift is possible, a new transition (q, b, q') can be inserted and the configuration set contained in q' must be calculated.

GoTo determines the successor state (successor configuration set) when shifting over b. The successor state includes all successor configurations and their descendants.

Basis determines the immediate successor configurations for every configuration of a configuration set when shifting over b.

PassesX returns the immediate successor configurations or an empty set, depending on if a shift over b is possible.

4.3.4 Parser-Related Functions

All parser-related functions are grouped in the file `pd_alg_lrp`. They encompass creating the parser from the LR machine, transforming the parser into an EDPDA and diminishing the parser's language back to the unaugmented language.

Lrp constructs an LR parser from an LR machine. It sets the parser states as the LR machine state. The LR machine's start and end states are excluded because they belong to the augmented grammar

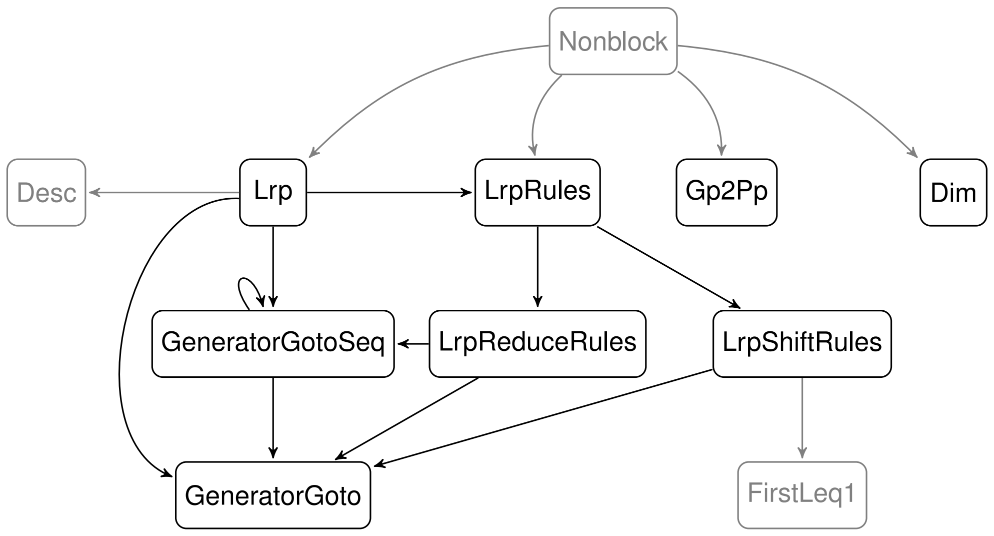


Figure 4.10: Dependencies of parser-related functions

only. Terminals are set as the unaugmented grammar's terminals and the marked state is set as the state preceding the LR machine's marked state. Finally, the parser actions are computed.

LrpRules computes all the parser actions by aggregating reduce and shift actions.

LrpReduceRules computes all reduce actions for the parser, as shown in equation 2.10.

GeneratorGotoSeq determines the sequence of states the LR machine takes from a certain state when given a certain input string.

GeneratorGoto determines the successor state of an LR machine state when given a certain input symbol.

LrpShiftRules computes all reduce actions for the parser, as shown in equation 2.9.

Gp2Pp turns the parser into an EDPDA, as per the rules 3.23, 3.24 and 3.25.

Dim deletes all transitions whose event is the augment symbol $\$$ and sets their start states as marked states. This step was shown from Figure 3.25 to Figure 3.26.

4.3.5 First-Related Functions

All functions relating to FirstLeq1() are grouped in the file `pd_alg_first`. Together, they serve to determine the set of possible terminals that can occur at the beginning of a word $w \in (G.\Sigma \cup G.N)^*$. This is useful for determining possible lookaheads, like with Desc11(), and for checking the existence of a follow-up, like with LrpShiftRules().

FirstLeq1 determines which words $\in \Sigma^{k \leq 1}$ can be at the beginning of w. Effectively, it determines either the set of following terminals in the case of $k = 1$ or tests for the existence of following terminals in the case of $k = 0$.

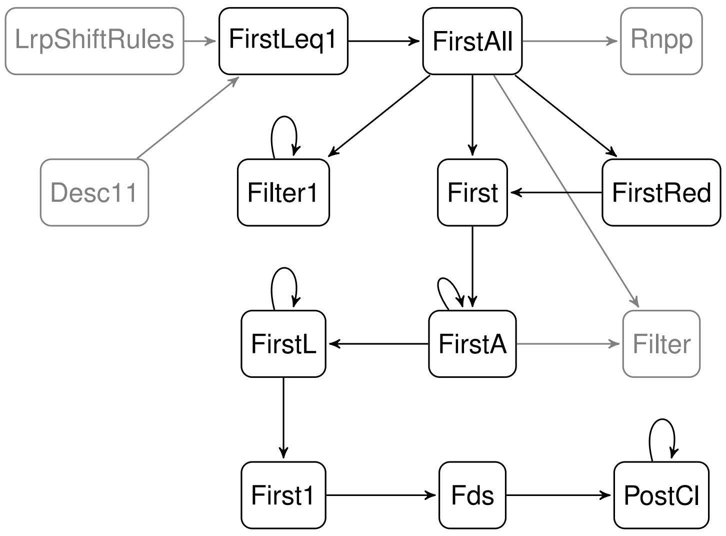


Figure 4.11: Dependencies of first-related functions

FirstAll determines the set of terminals that can be at the beginning of w. If no nonterminals are in w (tested by Filter()), the first terminal is selected via First(). If there are nonterminals in w, the first nonterminal A is selected and it is tested whether it can be reduced. This test is done by copying the grammar, setting the copy's start symbol as A, and calling Rnpp(). Only if A is reducible the first terminal that can be produced by A is determined by FirstRed().

First determines the set of terminals that can occur at the beginning of w. If $w = \lambda$, $\{\lambda\}$ will be returned. If $w = aw', a \in \Sigma$, $\{a\}$ will be returned. If $w = Aw', A \in N$, the terminals that can follow after applying productions $A \to w''$ will be returned. If this set contains λ, the set of terminals at the beginning of w' must be returned as well, resulting in a recursive call of First().

FirstA looks up the set of first terminals of w by calling a mapping function $f : (G.\Sigma \cup G.N)^* \to G.\Sigma$.

FirstL iteratively builds the mapping function f until no more changes are made to f.

First1 updates the mapping function f to the updated mapping function f' by iterating over each word returned by Fds(). If a word $w = \lambda$, $f'(w) \leftarrow f'(w) \cup \{\lambda\}$. If $w = aw', a \in \Sigma$, $f'(w) \leftarrow f'(w) \cup \{a\}$. If $w = Aw', A \in N$, each production $A \to x$ needs to be examined and the function is updated with $f'(w) \leftarrow f'(w) \cup f(x) \cup f(A)$. λ must be removed from this set if $w' \neq \lambda$, because eliminability of A does not imply eliminability of w'. If A is eliminable, another update $f'(w) \leftarrow f'(w) \cup f(w')$ takes place.

Fds constructs a set of words, consisting of each grammar production's left-hand side and the postfix closure of the right-hand side. These words are called the "first domain strings".

PostCl constructs the postfix closure of a word.

Filter1 extracts the first symbol $s \in S$ from a word, where S is a given symbol set.

FirstRed tests a word w for consistency before calling First(). A word is consistent if all symbols are either terminals or nonterminals of the grammar. The pseudocode for this function is written to work even if there are nonterminals $\notin G.N$, because it makes the proof of correctness easier. Since this case can never occur in the actual implementation, this functionality has been omitted.

4.4 Testing

Testing is a very important task in implementing, and a very time-consuming one as well. An estimated 80 % of the programming time has been spent on developing and analyzing over 70 test cases. Due to time concerns, test cases only exist for the supervisor algorithm and not for the underlying data structure. However, since the algorithm depends directly on the data structure, it is indirectly tested for correctness as well.
As a testing method, unit testing has been chosen. No unit testing framework has been used, because it would add to the dependencies of the Libfaudes framework and make it less portable. Testing was used as a tool while implementing to ensure the implementation worked as intended. It should also be used when changes are made to the Libfaudes to make sure the changes did not have unexpected side effects.
The smallest unit that is tested by a unit test is a function of the supervisor algorithm. Tests work on the function's interface so that the underlying implementation remains exchangable. Because functions depend on each other, initial testing and implementing has been done bottom up to prevent error propagation. If multiple errors from different test cases do pop up at once, the lower-level functions should always be checked first for the same reason. Since there are no circular depencies between functions, the error source is usually easy to narrow down. Unfortunately, unit testing is inherently flawed in that it is impossible to test all permutations of a test case's input parameters for correctness. Great care has been taken to cover as many parameters as possible, still, successful unit tests can never be a guarantee for correct behavior.
All unit tests for the supervisor algorithm are located in the `tutorial` subfolder of the plugin. Unit test file names correspond to the function file names with only `_test` appended at the end. Test case names give a short description of what the test is about. Nearly all tests rely on generators and grammars to modify, some of which can be reused in other tests. All test data structures are stored in `pd_test_util` for this purpose. No changes should be made there as it can have unforeseen effects on test case correctness.
The unit tests are run by executing the file `pd_algo_test`.

4.5 Problems and Optimization

The implementation of the supervisor algorithm has two known major bottlenecks, both of which are due to the complexity of the Sp2Lr() function contained within the nonblock function. The first is the quadratic scaling in the number of states it imposes on the nonblock function and the second is the quadratic memory consumption of Sp2Lr() itself. This makes testing of non-trivial examples difficult to impossible.
Considering the rules for creating grammar productions from an SDPDA, see eq. 3.20, the following number p of grammar productions will be generated each time Sp2Lr() is called:

$$\begin{aligned} p &= |t_{\text{pop}}| + |t_{\text{read}}||q| + |t_{\text{push}}|(1 + |q| + |q|^2) + |q_m||\Gamma| \\ &\Rightarrow \mathcal{O}(|t_{\text{push}}||q|^2) \end{aligned} \tag{4.1}$$

The complexity class is quadratic in the number of states multiplied with the number of push transitions. Experience from analyzing test cases shows that most push transitions are created from λ-push transitions when calling Rep0() within Rpp(). This replaces every λ-push transition with a number of push transitions equal to $|\Gamma|$. The complexity class of Sp2Lr(), and thereby Nonblock(), can be approximated by $\mathcal{O}(|\Gamma||q|^2)$.
Nonblock() itself is called by the accessible function, Ac(), and the function to remove useless ears, Ruls(), through the functions Te() and Ts() to test transition and state reachability. Ac() and Ruls() scale with the number of states as well, because every state (or a fraction thereof) must be tested for reachability. This sets the complexity of these two functions to $\mathcal{O}(|\Gamma||q|^3)$.
Although this complexity class cannot be diminished without severely altering the supervisor algorithm, steps can be taken to reduce the number of states, stack symbols and push transitions of input automatons and lessen execution time.

Memory usage can be cut down greatly by rewriting Sp2Lr() in such a way that it only generates productive productions. For an example, consider the automaton from Figure 2.3. Calling Nonblock() on it will result in an Sp2Lr() call with $|q| = 20$ and produce 5229 grammar productions, only 26 (≈ 0.49 %) of which are productive. With larger input sizes, the disparity gets even greater. If Sp2Lr() were to generate only productive productions, memory savings approaching 100 % can be achieved.
Note that while the complexity discussed here can be up to cubic, it is only the complexity of one call of Ccs(). Because it is currently unknown how often Ccs() will get called by Ccl(), the complexity of the whole supervisor algorithm may not even have polynomial bounds.

Treating the DPDA as a DFA

Reducing the size of a DFA without altering its marked language is achieved by the accessible and coaccessible operations. For DPDAs, a new accessible operation has been introduced in sec. 3.2.3. Both DFA-accessibility and DFA-coaccessibility operations can be applied to a DPDA without altering the DPDA's marked language. However, (co)accessibility on the DPDA is not guaranteed, because DFA-(co)accessibility does not test DPDA transitions for reachability.
In Libfaudes, the DPDA class `PushdownGenerator` inherits from the DFA class `System`, making DFA-(co)accessibility a built-in class feature. Calling these functions right before every call to Sp2Lr() potentially reduces the number of input states greatly at nearly no cost.

Removing Useless Transitions and Stack Symbols

When transforming a DPDA into an SDPDA, Rep0() creates $|\Gamma|$ push transititions for every λ transition. Usually, a lot of these new push transition will not be reachable. Testing them for reachability can be done via Te(), but this would defeat the purpose of reducing the number of push transitions prior to calling Nonblock(), because, Te() itself calls Nonblock(). Instead, a less complex test can be used to test for useless transitions.
Useless transitions can be detected on any EDPDA A as follows:

$$\begin{aligned}&\forall e = (q, a, uv, w, q') \in A.\delta : \\ &\quad Y \leftarrow \emptyset \\ &\quad X \leftarrow \text{GetPossibleStackTops}(A, q, Y) \\ &\quad v \notin X \land uv \neq \lambda \Rightarrow A.Q \leftarrow A.Q \backslash \{e\}\end{aligned} \tag{4.2}$$

where possible stack top symbols for the state q can be determined as

```
GetPossibleStackTops$(A, q, Y)$ :
    $X \leftarrow \emptyset$                                        set of possible stack tops
    $q \in Y \Rightarrow$ return $X$                                loop detection
    $Y' \leftarrow Y \cup \{q\}$                                    mark $q$ as examined
    $q = A.q_0 \Rightarrow X \leftarrow X \cup \{A.\Box\}$          start state always has $\Box$
    $\forall e = (p, a, u', v'w', q) \in A.\delta$ :               all transitions leading into $q$     (4.3)
        $w' \neq \lambda \Rightarrow X \leftarrow X \cup \{w'\}$    for push transitions
        $u' \neq \lambda \land v'w' = \lambda \Rightarrow X \leftarrow A.\Gamma$    for pop transitions
        $u' = \lambda \land v'w' = \lambda$                         for read transitions
            $\Rightarrow X \leftarrow X \cup$ GetPossibleStackTops$(A, p, Y')$
    return $X$
```

When a push transition leads to q, the symbol pushed onto the stack top by this push transition is a possible stack top. When a read transition leads to q, every stack top symbol of the read transition's start state p can be a stack top symbol of q as well. When a pop transition leads to q, every stack symbol can be a possible stack top symbol, because the symbol beneath the stack top is unknown and could be anything. If q is the start state, $\Box$ is always a possible stack top symbol. To avoid infinite loops, it must

be checked whether the stack top symbols of q have already been gathered.
Because the algorithm only retraces possible stack top symbols and not the whole stack, not every useless transition will be found. However, it potentially reduces the number of push transitions that will be handed over to Sp2Lr().
Consider the automaton shown in Figure 4.12, which shows a part of an SDPDA with $\Gamma = \{\square, \bullet, \diamond, \triangle\}$. The push transitions between q_2 and q_3 have probably been created by Rep0(). Three of them (marked in red) are not reachable, because q_2 can only be reached with $\bullet$ as a stack top.

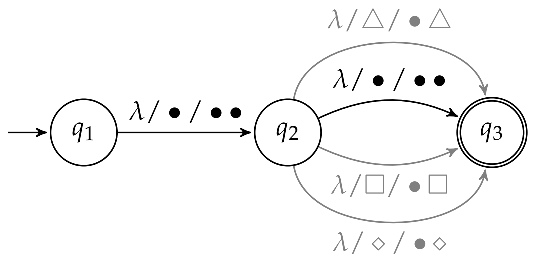

$\lambda/\triangle/\bullet\triangle$
q_1
$\lambda/\bullet/\bullet\bullet$
q_2
$\lambda/\bullet/\bullet\bullet$
q_3
$\lambda/\square/\bullet\square$
$\lambda/\diamond/\bullet\diamond$

Figure 4.12: automaton with $\Gamma = \{\square, \bullet, \diamond, \triangle\}$ and three useless transition (gray)

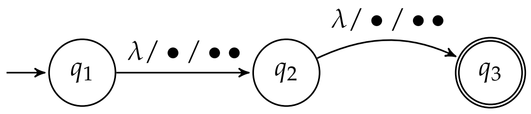

q_1
$\lambda/\bullet/\bullet\bullet$
q_2
$\lambda/\bullet/\bullet\bullet$
q_3

Figure 4.13: automaton with removed useless transitions, Γ can be reduce to $\{\square, \bullet\}$

Analysis of Rpp() shows that it will produce useless transitions (q', λ, u, u, q) when splitting up transitions of the form (p, a, u, u, q) into $(p, a, \lambda, \lambda, q')$ and (q', λ, u, u, q). Such transitions do not read anything and do not alter the stack and can be removed.

$$\begin{aligned} \forall e = &(q', \lambda, u, u, q) \in A.\delta, \forall e' = (q, b, u, v, r) \in A.\delta : \\ &A.\delta \leftarrow (A.\delta \backslash \{e\}) \cup \{(q', b, u, v, r)\} \end{aligned} \tag{4.4}$$

As shown in eq. 4.1, removing read transitions does not have much effect on the overall complexity of Sp2Lr(). However, making the resulting automaton DFA-accessible can potentially remove the state q if there were no other transition going into q. Reducing the number of states reduces Sp2Lr() complexity to a far greater degree. Figure 4.14 and 4.15 offer an example.

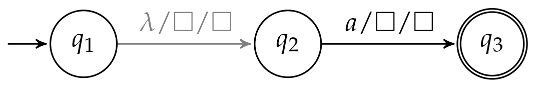


Figure 4.14: automaton with a useless λ-read transition (gray)

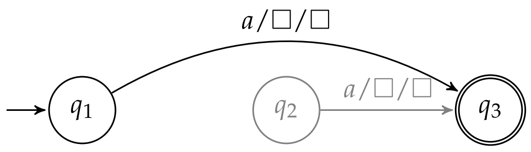


Figure 4.15: automaton with removed useless transition, can be reduced further when made DFA-accessible

After removing useless transitions in the two presented ways, unused stack symbols can be removed as well, see fig, 4.13. It is done as follows:

$$\begin{aligned} &\Gamma' \leftarrow \emptyset \\ &\forall e = (q, a, u, v, q') \in A.\delta, \forall \gamma_u \in u, \forall \gamma_v \in v : \\ &\qquad \Gamma' \leftarrow \Gamma' \cup \{\gamma_u\} \cup \{\gamma_v\} \\ &A.\Gamma \leftarrow \Gamma' \end{aligned} \tag{4.5}$$

Removing unused stack symbols in this way does not have much of a performance impact, see eq. 4.1. All the discussed functions for removing useless transitions and trimming the stack symbols can be found in the file `pd_alg_opt`.
Up until now, these optimizations have only been discussed if applied right before Sp2Lr(). They can be used for further performance gains if used at the very end of Nonblock(), right after the call to Rep0(). There, Rep0() can create useless transitions again. Removing them can reduce the number of needed

stack symbols. Even without removing these useless transitions, there are likely to be unused stack symbols. See Figure 3.25 for an example. The stack symbols of the automaton are the states of the LR machine minus start and marked state. $\Gamma = \{2, 4, 5, 6\}$, but only 5 and 6 are used. Now consider the top-level algorithm functions shown in Figure 4.5. If Ccs() gets called with unused stack symbols, so will be Ac(), Split() and Ruls() during the next iteration of Ccs().
Ac() indirectly calls Nonblock(). As explained above, the useless stack symbols would increase the number of push transitions and thereby the execution time of the nonblock's Sp2Lr(). This is prevented by the safeguard that is already in place, namely removing these useless transitions right before Sp2Lr(). Next up is Split(), which will multiply the number of states by the number of stack symbols. This increased state count will need to be processed by Ruls(). This leads to an increased number of calls of Ruls() to Nonblock(). Keeping this number of calls down will result in further performance gains of the supervisor algorithm.

Optimizing Memory Usage

The optimizations discussed until now were only concerned with shortening the runtime of the supervisor algorithm. The algorithm would work without them, although it would be slower. A far more important property to consider is the actual ability to complete a run of the algorithm without running out of memory. As shown above, memory is a serious concern with Sp2Lr() because of its quadratic scaling in the number of states. Nearly all grammar productions produced by Sp2Lr() are not productive. Rnpp() will remove them, but for large automatons, the algorithm will crash due to memory problems before getting to Rnpp(). The solution is to combine Sp2Lr() and Rnpp() so that the superfluous nonproductive productions do not get generated (and do not use memory) in the first place.
Consider the following rules for creating grammar productions, which are modified from 3.20.

$$
\begin{aligned}
&\forall q_i \in S.Q_m, \forall B \in S.\Gamma : \\
&\qquad G.P \leftarrow G.P \cup \{((q_i, B) \rightarrow \lambda)\} \\
&\forall e = (q_i, a, B, B, q_j) \in \delta_{\text{Read}} : \\
&\qquad (q_j, B) \in G.N \Rightarrow G.P \leftarrow G.P \cup \{((q_i, B) \rightarrow (q_j, B))\} \\
&\qquad \forall q_t \in S.Q : \\
&\qquad\qquad (q_j, B, q_t) \in G.N \Rightarrow G.P \leftarrow G.P \cup \{((q_i, B, q_t) \rightarrow a(q_j, B, q_t))\} \\
&\forall e = (q_i, \lambda, B, \lambda, q_j) \in \delta_{\text{Pop}} : \\
&\qquad G.P \leftarrow G.P \cup \{((q_i, B, q_j) \rightarrow \lambda)\} \\
&\forall e = (q_i, \lambda, B, CB, q_j) \in \delta_{\text{Push}} : \\
&\qquad (q_j, C) \in G.N \Rightarrow G.P \leftarrow G.P \cup \{((q_i, B) \rightarrow (q_j, C))\} \\
&\qquad \forall q_s \in S.Q : \\
&\qquad\qquad (q_j, C, q_s), (q_s, B) \in G.N \Rightarrow G.P \leftarrow G.P \cup \{((q_i, B) \rightarrow (q_j, C, q_s)(q_s, B))\} \\
&\qquad \forall q_s, q_t \in S.Q : \\
&\qquad\qquad (q_j, C, q_s)(q_s, B, q_t) \in G.N \Rightarrow G.P \leftarrow G.P \cup \{((q_i, B, q_t) \rightarrow (q_j, C, q_s)(q_s, B, q_t))\} \\
&\textit{repeat until } |G.P| \textit{ does not change}
\end{aligned}
\tag{4.6}
$$

New grammar productions are only added to $G.P$ if all the nonterminals on their right-hand side are in the set of nonterminals $G.N$. A nonterminal A must only be added to $G.N$ if a production $A \rightarrow x$ is added to $G.P$. This ensures that every production $\in G.P$ is productive, but requires iterative adding of productions to $G.P$ until no more productions can be found.
This version of Sp2Lr() is implemented as `Sp2Lr2()` in the same file as the original function. While it has a significantly increased runtime, this increase is partly made up by not needing to run Rnpp() anymore.

4.6 Summary

This chapter has introduced Libfaudes, an open source framework for working with discrete event systems. Libfaudes is extendable via plugins and does only support DFAs, which is why a new plugin, called “pushdown”, is written to facilitate working on DPDAs. This plugin contains all data structures such as grammars, parsers and pushdown automatons that are needed to implement the supervisor algorithm.
The algorithm itself consists of over 56 subfunctions, thematically split into different files that are all prefaced with `pd_alg_`. Dependencies between the files and functions and the general work flow have been shown and each function has been explained. For more information and formal definitions of each function, refer to [5].
Nearly all functions have unit tests assigned to them, contained in files with the same name as the original function, but with a `_test` suffix. The unit tests aim to cover as many test cases as possible and reasonable. They can be run to make sure eventual updates do not break the plugin and they provide error indicators in the case of something not working correctly.
Because the implementation of the supervisor algorithm is based on pseudocode derived from formal definitions that aim to be optimal in the sense of proving correctness, the efficiency of the algorithm is sometimes less than optimal. Functions can be rewritten and optimized to alleviate some of the problems. However, the algorithm’s overall complexity of at least polynomial scaling in the number of states is inherent and remains unaffected. The biggest bottleneck Sp2Lr(), both in memory consumption and time-wise, was optimized in different ways. Functions to reduce the size of the input automaton have been presented as well as an alternate Sp2Lr() version which has no memory issues.

Chapter 5

Conclusions

The supervisor algorithm by Schneider and Hess [5] introduces deterministic context-free languages to supervisory control theory, which usually is only applied to regular languages. This allows for a wider range of behaviors to be modelled by specifications, while still being able to compute a minimally restrictive controller.
The algorithm for computing a minimally restrictive controller for a regular specification, see Section 3.1.2, is straightforward and intuitive in that it involves only the operations accessible and coacceassible. The controller candidate does not change its structure and is iteratively approached by deleting states. This is very different from calculating the minimally restrictive controller for deterministic context-free specifications, which involves far more operations. The nonblock operation even leaves the scope of automatons altogether and knowledge of parsers a grammars is required to fully understand it. The amount of subfunctions involved makes it near impossible to execute all the algorithm's steps by hand, even for seemingly small examples. An intuitive approach to solve the nonblock problem by hand is hard to get right, because Nonblock() often requires remodelling of the input automaton and thus does not preserve the automaton's structure.
For these reasons, automated computation of the minimally restrictive controller is tremendously helpful. Besides, it serves as a proof of concept. The integration in Libfaudes provides an interface to work with deterministic context-free specifications not only in theory, but in praxis as well. However, even if automated computation pushes the size of processable automatons far beyond what can be done manually, the size of processable automatons is limited by execution time. The algorithm's execution time scales at least in the power of three of the number of automaton states.

5.1 Future Work

Future work on the supervisor algorithm should focus on optimizing the algorithm's runtime. Several optimizations have already been presented in Section 4.5, which can be expanded upon. Currently, nearly all execution time is spent converting SDPDAs into CFGs in the function Sp2Lr(). Further reduction of the input automaton size can reduce the runtime even more. For example, the detection of useless states and transitions can be done in a more sophisticated way. The approch presented in Section 4.5 takes only the stack top symbol into account to detect useless transitions. Tracking the stack at a depth of two or more stack symbols instead of just one could be beneficial. More transitions and states could be removed with the result being a more compressed automaton.
The implementation of the supervisor algorithm is currently single-threaded, although most modern computers usually have two or more cores. Rewriting Sp2Lr() to make use of concurrency can speed up the execution time by approximately the number of cores.
Libfaudes provides a graphical user interface to work on DFAs called DESTool [2]. Support for DPDAs and the "pushdown" plugin is not available. Usability would be greatly improved if a proper DESTool integration of the plugin were to be written.
Within the plugin itself, several implementation choices were dictated by the structure of Libfaudes, when they ideally would have been done differently. Usage of λ events and stack symbols depends on the user knowing to use the constant `FAUDES_PD_LAMBDA`. The user must never user the string `lambda`

as an event or as a stack symbol in another sense than the one intended by the constant. Hiding this behavior from the user would be a better programming practice, but is currently structurally impossible. Likewise, attaching pop and push parameters u and v to transitions has been done in an unintuitive way. Transitions in Libfaudes are unique only by their start state q, event a and end state q'. Different pop and push parameters must be attached as a set. This merges two transitions (q, a, u, v, q') and (q, a, u', v', q') into one transition object $(q, a, \{(u, v), (u', v')\}, q')$. A more intuitive programming practice would be to have two transition objects, one for each transition.

5.2 Summary

The implementation of the supervisor algorithm in Libfaudes provides an interface to work with deterministic context-free specifications in a control-theoretical environment. Because of its complexity, automated execution is the only way to properly apply the algorithm. While manual execution is possible, it is complicated, time-consuming and prone to errors. The biggest caveat of the algorithm is its scaling with the size of input automatons, which is at least the number of states to the power of three. Potential for optimization remains, but the complexity class cannot be changed and execution times will grow very quickly. The plugin containing the algorithm introduces DPDAs to Libfaudes for the first time. Support for DPDAs in Libfaudes and peripherial tools like DESTool is currently not optimal and remains to be improved.

Bibliography

[1] libFAUDES homepage http://www.rt.eei.uni-erlangen.de/FGdes/faudes/index.html

[2] DESTool homepage http://www.rt.eei.uni-erlangen.de/FGdes/destool/index.html

[3] GNU Lesser General Public License http://www.gnu.org/copyleft/lesser.html

[4] (Wonham, Ramadge, 1987) Wonham, W. M., Ramadge, P. J.: On the supremal controllable sublanguage of a given language. Siam Journal Control and Optimization, Vol 25., No. 3, May 1987

[5] (Schneider, Hess, 2013) Schneider, S., Hess, A.-K.: Internal Report, TU Berlin, 2013

[6] (Cassandras, Lafortune, 1987) Cassandras, C. G., Lafortune, S.: Introduction to Discrete Event Systems. Springer Science+Business Media, 2008

[7] (Raisch, 2011) Raisch, J.: Course Notes Discrete Event Systems. TU Berlin, Fachgebiet Regelungssysteme, Version 1.3, 2011

[8] (Wonham, 2010), Wonham, W. M.: Supervisory Control of Discrete-Event Systems. University of Toronto, Systems Control Group, 1997 - 2010, revised 01.07.2010

[9] (Knuth, 1965), Knuth, D. E.: On the Translation of Languages from Left to Right. Information and Control 8, p. 607 - 639, 1965

[10] (Sippu, Soisalon-Soininen, 1965), Sippu, S., Soisalon-Soininen, E.: Parsing Theory, Vol. II: LR(k) and LL(k) Parsing. EARCS Monographs on Theoretical Computer Science, Springer Verlag, 1990