Technische Universität Berlin


**Forschungsberichte**
**der Fakultät IV – Elektrotechnik und Informatik**

# Controller Synthesis for Deterministic Context Free Specification Languages

Sven Schneider, Anne-Kathrin Schmuck

December 11, 2013

Technische Universität Berlin
Bericht-Nr. 2013 - 09
ISSN 1436-9915

# Contents

# 1. Introduction

Ramadge and Wonham [14] established *supervisory control theory* (SCT) for controller synthesis on formal languages. Given a plant and a specification, SCT defines a *proper minimally restrictive supervisor* as a controller which generates a closed loop system (i.e., a plant restricted by a controller) that contains as many words allowed by the plant as possible while respecting the specification, not preventing uncontrollable events and always guiding the system to a satisfactory (marking) state. Wonham and Ramadge [17] presented an implementable fixed point algorithm to calculate the desired marked controller language $C$, formally defined by the above stated supervisory control problem (SCP), for regular plant and specification languages ($P$ and $S$, respectively), by operating on their DFA representations, generating a DFA representation of $C$. This fixed point algorithm iteratively ensures

(i) controllability, and
(ii) blockfreeness.

Obviously, step (i) may generate new blocking issues, while step (ii) may lead to new controllability problems. The algorithm terminates iff no more controllability problems or blocking situations are present. We perceive the languages $P$, $S$, and $C$ involved in the SCP as trace abstractions of the finite operational DFA-models used in the fixed-point algorithm.

The applicability of SCT is significantly increased if larger specification language classes can be used. In this report we investigate the case of deterministic context free specification languages. The class of deterministic context free languages (DCFL) contains the class of regular languages, and DCFL can be represented by deterministic pushdown automata (DPDA). Therefore, to obtain a constructive algorithm to solve this generalized supervisory control problem, steps (i) and (ii) of the fixed point algorithm must be implementable for DPDA.

Before discussing effective procedures for ensuring controllability and blockfreeness for DPDA in Chapter 3 and Chapter 4, respectively, we investigate the overall iteration outlined above for the setting of DPDA specifications more closely in Chapter 2. This is necessary because the operational computation of the fixed-point algorithm for the DFA setting in [14], outlined above, does not correspond well to their trace-abstract characterization of the desired closed-loop behavior used in the SCP. Furthermore, while the standard trace abstraction is sufficient for DFA and observable Petri nets we show in Chapter 2 that it is insufficient for DPDA with their unobservable $\lambda$-steps. This insight enforces the perspective of synthesizing automata realizations instead of languages, taken in this report.

Note that Griffin suggested an algorithm for the more restrictive setting of a prefix closed regular plant and a prefix closed deterministic context free specification language in [8]. However, as shown by the counterexample in Appendix A.1, Griffin's algorithm does not construct the minimally restrictive supervisor in general.

# 2. Reducing an Operational Supervisory Control Problem by Decomposition for DPDA

The purpose of Supervisory Control Theory (SCT) is to synthesize a controller for a plant and a specification such that the desired closed-loop behavior is enforced. Effective solvers have been constructed in the past for the setting of plants and specifications modeled by Deterministic Finite Automata (DFA). We extend the domain of the specification to Deterministic Pushdown Automata (DPDA) and verify an effective solver (up to two basic building blocks which ensure controllability and blockfreeness, effectively solved for this setting in Chapter 4 and Chapter 3). We verify the enforcement of desired operational criteria, which are, in contrast to the setting of DFA, partly oblivious to the (un)marked language of the closed loop. Our general approach trivially covers the setting of DFA and can be reused and adapted to develop effective solvers for other settings as the realizability of solutions to the supervisory control problem (SCP) is considered on an abstract level.

## 2.0 Introduction

The SCP was introduced and solved for plants and specifications modeled by DFA in [14, 17]. Subsequently, the SCP was considered for other settings including certain Petri nets by, e.g., [5, 6]. Later, [8, 9] addressed the SCP for DFA plants and DPDA specifications with prefix-closed marked languages. However, one key criterion of the SCP, namely minimal restrictiveness, is violated by his approach as explained in Chapter 3.

Wonham and Ramadge synthesize the desired controller language $C$, formally defined by the SCP, for regular plant and specification languages ($P$ and $S$, respectively), by operating on their DFA representations, generating a DFA representation of $C$ in [17]. We perceive the involved languages $P$, $S$, and $C$ as trace abstractions of the finite operational DFA-models. While this standard trace abstraction is sufficient for DFA and observable Petri nets we show that it is insufficient for DPDA with their unobservable $\lambda$-steps. This insight enforces the perspective of synthesizing automata realizations instead of languages.

In Section 2.1, the models, relevant to this chapter, including DPDA, are introduced. Afterwards we define trace abstractions for DPDA and analyze their basic properties in Section 2.2. In Section 2.3 we provide operational criteria of the desired (least restrictive) controller, and formally define a reduction of synthesizing a controller satisfying these constraints to the SCP by introducing an operational SCP (OSCP) based on trace abstractions introduced before.

The operational computation of the concrete solver for the DFA setting of [14], which is extensively used in the control of discrete event systems, does not correspond well to

their trace-abstract characterization of the desired closed-loop behavior. To bridge this gap,

(i) in Section 2.4, we formalize the correspondence between operational and trace-abstract solutions by suitably characterizing satisfactory controllers and verify their equivalence to the standard SCP solutions; and

(ii) in Section 2.5, we introduce adequate fixed-point algorithms which
  (a) determine only satisfactory controllers and
  (b) are constructed over operations ensuring controllability and blockfreeness for the underlying finite state models.

These fixed-point algorithms constitute solvers for the SCP for DFA plants and DPDA specifications when using the effective procedures for ensuring controllability and blockfreeness for DPDA, discussed in Chapter 4 and Chapter 3. In summary, we consider the implications of choosing DPDA as specifications on SCT

(a) by formalizing operational criteria for the desired controllers,

(b) by reducing the synthesis problem to the standard SCP by means of an OSCP using adequate trace abstractions which guarantee the desired criteria for controller realizations,

(c) by expressing the SCP by supremal elements of a complete lattice over our trace abstractions, and

(d) by decomposing the suprema-based characterization into fixed-point algorithms consisting of basic building blocks implemented in Chapter 4 and Chapter 3.

While outlining the process of verifying the adequacy of the trace abstractions of Section 2.2, we have verified the automata foundations of Section 2.1 and the central results of Section 2.4 and Section 2.5 in the interactive theorem prover Isabelle/HOL [13].

## 2.1 Models of Behavior

We assume a fixed set of possible events $\Sigma$ shared by all models contained in this chapter. This set is, as usual, partitioned into a set of controllable events $\Sigma_c$ and a set of uncontrollable events $\Sigma_{uc}$. In examples we assume $\Sigma_c = \{a, b, c, d, e\}$ and $\Sigma_{uc} = \{u, v\}$.

We use the following notation throughout the chapter.

**Notation 1**
*Let $A$ be a set. Then*

*(i)* $A^*$ *denotes the set of finite words over* $A$,
*(ii)* $A^{\omega *}$ *denotes the set of finite and infinite words over* $A$,
*(iii) single symbols are denoted by greek letters (except for* $\lambda$*, the empty word),*
*(iv) words are denoted by* $s, w$,
*(v)* $\cdot$ *is the (usually omitted) concatenation operation on words (and languages),*
*(vi)* $\sqsubseteq$ *is the prefix relation,*
*(vii)* $\overline{A}$ *is the prefix-closure of* $A$,
*(viii)* $\sqsupseteq$ *is the suffix relation, and*
*(ix)* $\lambda x.f(x)$ *is the nameless function equal to* $f$ *(e.g.,* $f(x) = x^2$ *implies* $f = \lambda x.x^2$*).*

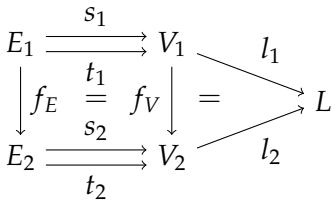
### 2.1.1 *Labeled Graphs*

We use labeled graphs as representations of discrete (operational) behavior, where edges correspond to steps.

**Definition 1 (Labelled Graphs)**
$G = (V, E, L, s, t, l) \in \text{LGraph}$ *iff*
*(i)* $V$ *is a set of vertices,*
*(ii)* $E$ *is a set of edges,*
*(iii)* $L$ *is a set of labels,*
*(iv)* $s : E \to V$ *maps each edge to its source-vertex,*
*(v)* $t : E \to V$ *maps each edge to its target-vertex, and*
*(vi)* $l : V \to 2^L$ *maps each vertex to the set of its labels.*

Two operational behaviors (given as LGraphs) are equivalent iff they are renamings of each other.

**Definition 2 (LGraph-Isomorphisms)**
*Let* $G_1 = (V_1, E_1, L, s_1, t_1, l_1)$ *and* $G_2 = (V_2, E_2, L, s_2, t_2, l_2)$ *be two* LGraph*s with identical sets of labels.*
*Then* $f = (f_V : V_1 \to V_2, f_E : E_1 \to E_2) : G_1 \to G_2$ *containing mappings for vertices and edges is an* LGraph*-isomorphism iff*
*(i)* $f_V$ *and* $f_E$ *are bijections,*
*(ii) sources are preserved:* $s_2 \circ f_E = f_V \circ s_1$,
*(iii) targets are preserved:* $t_2 \circ f_E = f_V \circ t_1$, *and*
*(iv) labels are preserved:* $l_1 = l_2 \circ f_V$.




*Furthermore,* $G_1 \cong G_2$ *iff there is an* LGraph*-isomorphism* $f : G_1 \to G_2$.

### 2.1.2 *Discrete Event Systems*

We introduce Discrete Event Systems (DES) as a denotational model which is entirely decoupled from the syntax and semantics of concrete (e.g., automata) realizations. The DES is given by the sets of
(i) all (possible) observations—the unmarked language $L_{um}$ and
(ii) all (possible) desired observations—the marked language $L_m$.

**Definition 3 (Discrete Event System)**
$D = (\!| L_{um}, L_m |\!) \in \text{DES}$ *iff* $L_m \subseteq L_{um} = \overline{L_{um}} \subseteq \Sigma^*$.
$\mathrm{L_{um}}(D)$ *and* $\mathrm{L_m}(D)$ *denote the two components of* $D$.

We repeat the well known notions of language-blockfreeness and language-controllability of DES which are central to the standard SCT (cf. [14]).

**Definition 4 (Blockfree and Controllable DES)**
*Let* $D_1, D_2 \in \text{DES}$. *Then*
*(i)* $D_1$ *is language-blockfree*[1] *iff* $\mathrm{L_{um}}(D_1) \subseteq \overline{\mathrm{L_m}(D_1)}$ *and*

[1] Every observation is a prefix of a desired observation.

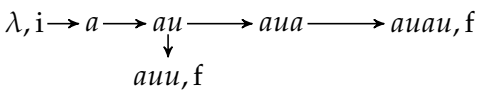



Figure 2.1.: $P = (\overline{\{auau, auu\}}, \{auau, auu\}) \in$ DES is represented as an LGraph: the names of the vertices in the visualization are the labels of the vertices.

*(ii)* $D_1$ *is language-controllable*[2] *w.r.t* $D_2$ *(denoted by* $\mathrm{LCont}(\mathrm{L_{um}}(D_1), \mathrm{L_{um}}(D_2), \Sigma_{uc})$*) iff* $(\mathrm{L_{um}}(D_1)\cdot\Sigma_{uc}) \cap \mathrm{L_{um}}(D_2) \subseteq \mathrm{L_{um}}(D_1)$.

We will use the following complete lattice in Section 2.4 to characterize important DES. Suprema and infima of sets of languages are denoted using $\cup$ and $\cap$ in this chapter.

**Lemma 1 (Complete Lattice of DES)**
DES *forms a complete lattice using the following operations where* $\{A, B\} \cup M \subseteq$ DES.
*(i) the least element:* $\bot = \langle\!|\emptyset, \emptyset|\!\rangle$,
*(ii) the greatest element:* $\top = \langle\!|\Sigma^*, \Sigma^*|\!\rangle$,
*(iii) the inclusion:* $A \leq B$ *iff* $\mathrm{L_{um}}(A) \subseteq \mathrm{L_{um}}(B)$ *and* $\mathrm{L_m}(A) \subseteq \mathrm{L_m}(B)$.
*(iv) the strict inclusion:* $A < B$ *iff* $A \leq B$ *and* $A \neq B$.
*(v) the synchronous (infimal) composition (denoted by* $A \times B$ *in the rest of this report):* $\inf(A, B) = \langle\!|\, \mathrm{L_{um}}(A) \cap \mathrm{L_{um}}(B), \mathrm{L_m}(A) \cap \mathrm{L_m}(B)|\!\rangle$,
*(vi) the alternative (supremal) composition:* $\sup(A, B) = \langle\!|\, \mathrm{L_{um}}(A) \cup \mathrm{L_{um}}(B), \mathrm{L_m}(A) \cup \mathrm{L_m}(B)|\!\rangle$,
*(vii) the maximal* DES *included in all* DES *from* $M$[3]*:* $\mathrm{Inf}(M) = \langle\!|\, \cap \mathrm{L_{um}}(M), \cap \mathrm{L_m}(M)|\!\rangle$*, and*
*(viii) the least* DES *which includes all* DES *from* $M$*:* $\mathrm{Sup}(M) = \langle\!|\, \cup \mathrm{L_{um}}(M), \cup \mathrm{L_m}(M)|\!\rangle$.

We give the natural operational behavior of a DES.

**Definition 5 (Natural Operational Behaviour of DES)**
*Let* $D \in$ DES.
*Then* $[\![D]\!]^{\mathrm{DES}}_{\mathrm{LGraph}} = (V, E, L, s, t, l)$ *is the (natural)* LGraph*-representation of* $D$ *iff*
*(i)* $V = \mathrm{L_{um}}(D)$,
*(ii)* $E = \{(w, w\cdot\sigma) \mid w\cdot\sigma \in \mathrm{L_{um}}(D)\}$,
*(iii)* $L = \{\mathrm{i}, \mathrm{f}\} \cup \mathrm{L_{um}}(D)$,
*(iv)* $s(w, w') = w$,
*(v)* $t(w, w') = w'$*, and*
*(vi)* $l(w) = \{w\} \cup \{\mathrm{i} \mid w = \lambda\} \cup \{\mathrm{f} \mid w \in \mathrm{L_m}(D)\}$.

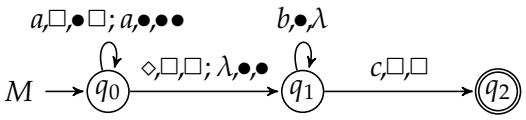
An example of this encoding is given in Figure 2.1 where a DES is visualized as an LGraph.

Obviously, there is an isomorphism between the LGraph-representations of two DES iff the DES are identical.

**Lemma 2 (Sound Encoding)**
*Let* $D_1, D_2 \in$ DES.
*Then* $[\![D_1]\!]^{\mathrm{DES}}_{\mathrm{LGraph}} \cong [\![D_2]\!]^{\mathrm{DES}}_{\mathrm{LGraph}}$ *iff* $D_1 = D_2$.

[2] Whenever $D_2$ has the observation $w$, $D_1$ does not prevent $w$, and $w\cdot u$ is an observation of $D_2$ (for an uncontrollable event $u$), then $w\cdot u$ is not prevented by $D_1$.

[3] Remark: $\mathrm{L_m}$ and $\mathrm{L_{um}}$ are here computing images, i.e., the sets of the (un)marked languages of DES from $M$.




$(\bot, (q_0,\lambda,\square)) \vdash_M ((q_0,a,\square,\bullet\square,q_0), (q_0,a,\bullet\square)) \vdash_M ((q_0,a,\bullet,\bullet\bullet,q_0), (q_0,aa,\bullet\bullet\square))$
$\vdash_M ((q_0,\lambda,\bullet,\bullet,q_1), (q_1,aa,\bullet\bullet\square)) \vdash_M ((q_1,b,\bullet,\lambda,q_1), (q_1,aab,\bullet\square))$
$\vdash_M ((q_1,b,\bullet,\lambda,q_1), (q_1,aabb,\square)) \vdash_M ((q_1,c,\square,\square,q_2), (q_2,aabbc,\square))$

Figure 2.2.: $M \in$ FPDA with $L_m(M) = \{a^{n+1}b^{n+1}c \mid n \in \mathbf{N}\}$ and an exemplary initial derivation in which edges are printed in gray.

### 2.1.3 *Finalizing Pushdown Automata* (FPDA)

FPDA, introduced here, are DFA enriched with a single stack-variable which can be used to remember aspects for later reuse of a generated word. An example of FPDA generating a language unacceptable by any DFA is given in Figure 2.2: The FPDA $M$ operates by remembering the number of generated $a$ as an equally long sequence of $\bullet$ in its stack—then, for any generated $b$ one $\bullet$ is popped from the stack. Furthermore, we additionally assume that the automaton can decide to stop generating symbols by "generating" the end-of-output marker $\diamond$. This intuitive explanation is formalized in the following definition of FPDA and their operational semantics.

**Definition 6 (Finalizing Pushdown Automata (FPDA))**
$M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F, \diamond) \in$ FPDA *iff*
*(i)* *the states $Q$, the output alphabet $\Sigma$, the stack alphabet $\Gamma$, and the set of edges $\delta$ are finite,*
*(ii)* $\delta : Q \times (\Sigma \cup \{\lambda, \diamond\}) \times \Gamma \times \Gamma^* \times Q$,
*(iii)* *the end-of-output marker $\diamond$ is not contained in $\Sigma$,*
*(iv)* *the end-of-stack marker $\square$ is contained in $\Gamma$,*
*(v)* *the end-of-stack marker is never removed from the stack ($(q, \sigma, \square, s', q')$ implies $s' \sqsupseteq \square$),*
*(vi)* *the marking states $F$ and the initial state $q_0$ are contained in $Q$.*

We provide the slightly nonstandard branching semantics of an FPDA $M$ which utilizes a history variable in the configurations to greatly simplify the definitions of the trace abstractions presented in Section 2.2. Furthermore, this branching semantics corresponds to the intuition that the finite state realizations are generators rather than acceptors of languages.

**Definition 7 (Semantics of FPDA)**
*(i)* *the set of configurations $\mathcal{C}(M) = Q \times \Sigma^* \cdot \{\lambda, \diamond\} \times \Gamma^+$ where $(q, w, s) \in \mathcal{C}(M)$ consists of a state $q$, a history variable $w$ (storing the symbols generated), and a stack variable $s$,*
*(ii)* *the initial configuration $\mathcal{C}_{\text{init}}(M)$ is $(q_0, \lambda, \square)$,*
*(iii)* *the set of marking configurations $\mathcal{C}_{\text{fin}}(M)$ is defined by $\{(q, w, s) \in \mathcal{C}(M) \mid q \in F\}$,*
*(iv)* *the annotated configurations $\mathcal{C}^\delta(M) = ((\delta \cup \{\bot\}) \times \mathcal{C}(M))$ additionally contain "pre-edges" from $\delta$*
*(v)* *the single-step relation (operating on the annotated configurations) $\vdash_M : \mathcal{C}^\delta(M) \times \mathcal{C}^\delta(M)$ is defined by $(e, (p, w, s \cdot s'')) \vdash_M ((p, w', s, s', p'), (p', w \cdot w', s' \cdot s''))$ (i.e., $q'$ is the new state, $w'$ is added to the history variable, and the prefix $s$ of the stack-variable $s \cdot s''$ is replaced by $s'$) where $w \sqsupseteq \diamond$ implies $w' = \lambda$ (i.e., once the end-of-output marker $\diamond$ has been generated, the history variable cannot be extended),*

| | FPDA | PDA | DPDA | NFA | DFA |
|---|---|---|---|---|---|
| deterministic | | | ✓ | | ✓ |
| $\lambda$-step-free | | | | ✓ | ✓ |
| $\diamond$-step-free | | ✓ | ✓ | ✓ | ✓ |
| stack-free | | | | ✓ | ✓ |

Table 2.1.: Subclasses of FPDA.

*(vi)* *the set of derivations* $\mathcal{D}(M)$ *contains all elements from* $\mathcal{C}^\delta(M)^{\omega *}$ *starting in a configuration of the form* $(\bot, c)$ *where all adjacent* $(e_1, c_1), (e_2, c_2) \in \mathcal{C}^\delta(M)$ *satisfy* $(e_1, c_1) \vdash_M (e_2, c_2)$,

*(vii)* *the set of initial derivations* $\mathcal{D}_\mathrm{I}(M)$ *contains all elements of* $\mathcal{D}(M)$ *starting with the initial configuration (i.e.,* $(\bot, (q_0, \lambda, \square))$*),*

*(viii)* *the set of reachable configurations* $\mathcal{C}_\mathrm{reach}(M)$ *is defined by* $\{c \in \mathcal{C}(M) \mid \exists d \in \mathcal{D}_\mathrm{I}(M), n \in \mathbf{N} \,.\, d(n) = (e, c)\}$,

*(ix)* *let* $\mathrm{out} : (\mathcal{C}(M) \cup \mathcal{C}^\delta(M)) \to \Sigma^*$ *be defined by* $\mathrm{out}(e, (q, w, s)) = \mathrm{out}(q, w, s) = (\text{if } w \sqsupseteq \diamond \text{ then } \mathrm{butlast}(w) \text{ else } w)$ *(i.e., we drop the possibly contained end-of-output marker* $\diamond$ *from the history variable to obtain the output of a configuration),*

*(x)* *the marked language* $\mathrm{L_m}(M)$ *is defined by* $\mathrm{out}(\mathcal{C}_\mathrm{fin}(M) \cap \mathcal{C}_\mathrm{reach}(M))$*, and*

*(xi)* *the unmarked language* $\mathrm{L_{um}}(M)$ *is defined by* $\mathrm{out}(\mathcal{C}_\mathrm{reach}(M))$.

*The concatenation of* $d_1, d_2 \in \mathcal{D}(M)$ *at index* $n \in \mathbf{N}$ *is given by* $(d_1 \cdot_n d_2) = \lambda i.$ if $i \leq n$ then $d_1(i)$ else $d_2(i - n)$.

An example of an FPDA-derivation is given in Figure 2.2. The well known sub-classes of FPDA having one ore more of the properties below are defined in Table 2.1.

**Definition 8 (Sub-classes of FPDA)**
*An* FPDA *is deterministic iff for every reachable configuration all two distinct steps append distinct elements of* $\Sigma \cup \{\diamond\}$ *to the history variable. An* FPDA *is* $\lambda$*-step-free iff no edge in* $\delta$ *is of the form* $(p, \lambda, s, s', p')$*. An* FPDA *is* $\diamond$*-step-free iff no edge in* $\delta$ *is of the form* $(p, \diamond, s, s', p')$*. An* FPDA *is stack-free iff every edge in* $\delta$ *is of the form* $(p, a, \square, \square, p')$.

**Remark 1**
*There is no complete lattice over* DFA *(*DPDA, FPDA*) since regular languages (deterministic context free languages, context free languages) are not closed under infinite union*[4] *and intersection*[5]*. Therefore, similarly to [14], we state the* SCP *over the complete lattice of trace abstractions. In particular, we are using the complete lattice over* DES *from Lemma 1.*

## 2.2 Adequacy of DES w.r.t. FPDA

Since we want to express the SCP for FPDA in terms of an SCP over DES it is essential that DES adequately describe the operational behavior of FPDA. We provide three variations of encodings of

(i) FPDA into DES and

[4] A union of infinitely many singletons is not context free: $\cup\{\{a^n b^n\} \mid n \in \mathbf{N}, n \text{ prime}\} = \{a^n b^n \mid n \in \mathbf{N}, n \text{ prime}\}$

[5] An intersection of infinitely many singletons is not context free: $\cap\{\Sigma^* \setminus \{a^n b^n\} \mid n \in \mathbf{N}, n \text{ prime}\} = \Sigma^* \setminus \{a^n b^n \mid n \in \mathbf{N}, n \text{ prime}\}$

(ii) FPDA into LGraphs

and investigate in each of the three variations whether the FPDA to DES encoding preserves the operational behavior, i.e., we compare the operational behavior of the FPDA and the resulting DES by translating both into LGraphs using the encodings DES to LGraph and FPDA to LGraph. In fact, we provide adequate encodings for DFA and DPDA.

### 2.2.1 *Adequate Encoding for* DFA

The following encoding defines for an FPDA (and its operational semantics) the *observable* operational behavior, expressed as an LGraph. In this variation we expect every step to be fully observable which is only true for DFA and $\lambda$-step free DPDA.

**Definition 9 (Natural Operational Behaviour of FPDA)**
*Let* $M \in$ FPDA.
*Then* $[\![M]\!]^{\text{FPDA}}_{\text{LGraph}} = (E, V, L, s, t, l)$ *is the (natural)* LGraph*-representation of* $M$ *iff*
*(i)* $V$ *is the smallest set containing* $\mathcal{C}_{\text{init}}(M)$ *which is closed under* $\vdash_M$,
*(ii)* $E = \vdash_M$,
*(iii)* $L = \{\mathrm{i}, \mathrm{f}\} \cup \mathrm{L}_{\text{um}}(M)$,
*(iv)* $s(c_1, e, c_2) = c_1$,
*(v)* $t(c_1, e, c_2) = c_2$, *and*
*(vi)* $l(c) = \{\text{out}(c)\} \cup \{\mathrm{i} \mid c = \mathcal{C}_{\text{init}}(M)\} \cup \{\mathrm{f} \mid c \in \mathcal{C}_{\text{fin}}(M)\}$.

Since each step is observable, two FPDA are equivalent w.r.t. $[\![\cdot]\!]^{\text{FPDA}}_{\text{LGraph}}$ iff they are renamings of each other.

**Proposition 1 (Sound Encoding)** *Let* $M^1, M^2 \in$ FPDA *be accessible.*
*Then* $[\![M^1]\!]^{\text{FPDA}}_{\text{LGraph}} \cong [\![M^2]\!]^{\text{FPDA}}_{\text{LGraph}}$ *iff* $M^1$ *is a renaming (states, stack-elements, stack-end-marker, and end-of-output-marker) of* $M^2$. □

Observe that the standard/natural encoding $[\![\cdot]\!]^{\text{FPDA}}_{\text{DES}}$ is the (implicitly used) trace abstraction from FPDA into DES used in [17] as a denotational description of the operational behavior of DFA.

**Definition 10 (Natural Encoding $[\![\cdot]\!]^{\text{FPDA}}_{\text{DES}}$)**
*Let* $M \in$ FPDA.
*Then* $[\![M]\!]^{\text{FPDA}}_{\text{DES}} = \langle\!| \mathrm{L}_{\text{um}}(M), \mathrm{L}_{\text{m}}(M) |\!\rangle \in$ DES.

We can conclude that for two FPDA which are DFA or $\lambda$-step free DPDA the observable operational behavior coincides iff their DES representations have equivalent observable operational behaviors, i.e., they have identical (un)marked languages.

**Theorem 1 ($[\![\cdot]\!]^{\text{FPDA}}_{\text{DES}}$ is Fully Abstract w.r.t. $[\![\cdot]\!]^{\text{FPDA}}_{\text{LGraph}}$)**
*Let* $M^1, M^2 \in$ FPDA, *deterministic,* $\diamond$*-step-free, and* $\lambda$*-step-free.*
*Then,* $[\![M^1]\!]^{\text{FPDA}}_{\text{LGraph}} \cong [\![M^2]\!]^{\text{FPDA}}_{\text{LGraph}}$ *iff* $[\![[\![M^1]\!]^{\text{FPDA}}_{\text{DES}}]\!]^{\text{DES}}_{\text{LGraph}} \cong [\![[\![M^2]\!]^{\text{FPDA}}_{\text{DES}}]\!]^{\text{DES}}_{\text{LGraph}}$.

Confer to Figure 2.3 for a visualization of Theorem 1. However, since none of the assumptions of Theorem 1 can be dropped, as stated in the following corollary, the encodings of FPDA into LGraph and DES are unsatisfactory (e.g., consider DPDA with $\lambda$-steps).

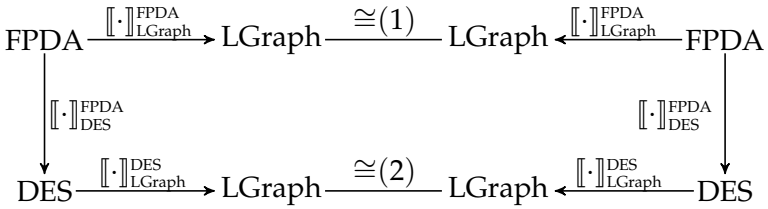



Figure 2.3.: For Theorem 1: (1) holds iff (2) holds.

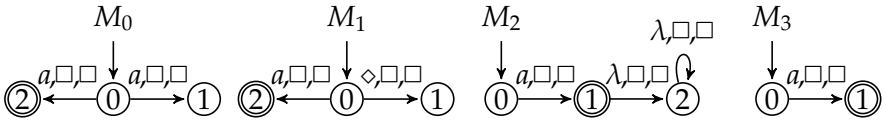



Figure 2.4.: $M_0$, $M_1$, $M_2$, and $M_3$ have the same $[\![\cdot]\!]^{\text{FPDA}}_{\text{DES}}$ image $\langle\!\langle \{\lambda, a\}, \{a\} \rangle\!\rangle$ which is language-blockfree. Only $M_3$ is operationally-blockfree.

**Corollary 1**
*Let* $M^1 \in$ FPDA *which is not deterministic, not* $\diamond$*-step-free, or not* $\lambda$*-step-free and let* $M^2 \in$ DFA. *Then,* $[\![M^1]\!]^{\text{FPDA}}_{\text{LGraph}} \cong [\![M^2]\!]^{\text{FPDA}}_{\text{LGraph}}$ not if but only if $[\![[\![M^1]\!]^{\text{FPDA}}_{\text{DES}}]\!]^{\text{DES}}_{\text{LGraph}} \cong [\![[\![M^2]\!]^{\text{FPDA}}_{\text{DES}}]\!]^{\text{DES}}_{\text{LGraph}}$.

For example, consider the FPDA in Figure 2.4. When choosing $M_0$, $M_1$, or $M_2$ for $M^1$ and $M_3$ for $M^2$ in Corollary 1 then $[\![[\![M^1]\!]^{\text{FPDA}}_{\text{DES}}]\!]^{\text{DES}}_{\text{LGraph}} \cong [\![[\![M^2]\!]^{\text{FPDA}}_{\text{DES}}]\!]^{\text{DES}}_{\text{LGraph}}$ is satisfied but not $[\![M^1]\!]^{\text{FPDA}}_{\text{LGraph}} \cong [\![M^2]\!]^{\text{FPDA}}_{\text{LGraph}}$.

### 2.2.2 *Adequate Encoding for* DPDA

Corollary 1 states that DES are no sound denotational model for FPDA (including DPDA) w.r.t. the full observability defined via $[\![\cdot]\!]^{\text{FPDA}}_{\text{LGraph}}$. The problem stems from the steps which are invisible in the DES ($\lambda$-steps, $\diamond$-steps, and nondeterministic choices (note that this kind of step is also not explicitly contained in the operational semantics)) but visible to the behavioral equivalence $[\![M^1]\!]^{\text{FPDA}}_{\text{LGraph}} \cong [\![M^2]\!]^{\text{FPDA}}_{\text{LGraph}}$. These steps could be made visible by modification of the FPDA, however, this is not reasonable for non-determinism and the end-of-output marker (see Remark 2 in Section 2.3). In this section we consider FPDA which are deterministic and $\diamond$-step-free (i.e., DPDA): as the $\lambda$-steps represent the internal steps of the controller, properties on its occurrence in executions of the closed loop may be of great importance.

We distinguish between two kinds of $\lambda$-step-sequences: finite sequences and infinite sequences.

▶*Finite* $\lambda$*-step-sequences* do not necessarily need to be observable for DPDA because DPDA do not have a worst-case-execution-time in general[6]. Thus, we assume in this chapter that there are no properties to be enforced on these finite sequences. Observe that in DPDA at most finitely many $\lambda$-steps, which are executed deterministically, occur between two visible steps. We can therefore replace maximal finite sequences by a single vertex (with label f iff some vertex in the finite sequence had the label f) by $[\![\cdot]\!]^{\text{FPDA-}\lambda}_{\text{LGraph}}$ as exemplified in Figure 2.5 (page 14).

[6]Consider $\{a^n b^m c^n \mid n, m \in \mathbf{N}\} \cup \{a^n b^m d^m \mid n, m \in \mathbf{N}\}$: all $a$ and $b$ have to be recorded by the stack: when reaching a $c$ all records of $b$ have to be removed in an unbounded number of steps.

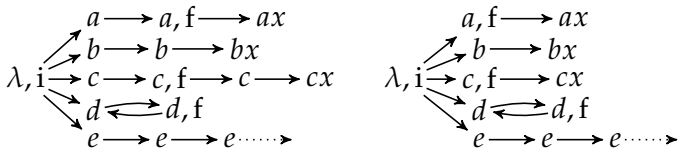



Figure 2.5.: The operation $[\![\cdot]\!]^{\text{FPDA-}\lambda}_{\text{LGraph}}$ transforms the LGraph returned by $[\![\cdot]\!]^{\text{FPDA}}_{\text{LGraph}}$ on the left into the LGraph on the right.

**Definition 11 (Operational Encoding $[\![\cdot]\!]^{\text{FPDA-}\lambda}_{\text{LGraph}}$)**
*Let* $M \in$ DPDA.
*Let* $[\![M]\!]^{\text{FPDA}}_{\text{LGraph}} = (V, E, L, s, t, l)$.
*Then* $[\![M]\!]^{\text{FPDA-}\lambda}_{\text{LGraph}}$ *is the greatest fixed point of the following operation: Whenever*
*(i)* $e_1, e_2 \in E$
*(ii)* $t(e_1) = s(e_2)$
*(iii)* $w \in \Sigma^*, \sigma \in \Sigma$
*(iv)* $w \in l(s(e_1))$,
*(v)* $w \in l(s(e_2))$, *and*
*(vi)* $w\sigma \in l(t(e_2))$
*then*
*(i)* $e_1, e_2$ *are removed from* $E$,
*(ii)* $s(e_2)$ *is removed from* $V$,
*(iii) a fresh edge* $e$ *is added to* $E$ *s.t.* $s(e) = s(e_1)$ *and* $t(e) = t(e_2)$, *and*
*(iv)* $\mathrm{f}$ *is added to* $l(s(e))$ *if* $\mathrm{f} \in l(s(e_2))$.

►*Infinite λ-step-sequences* represent lifelocks which can be contracted in the operational semantics into a single step which makes that step visible with the symbol $\circlearrowleft \in \Sigma_{\text{uc}}$.

**Definition 12 (Encoding $[\![\cdot]\!]^{\text{FPDA-}\omega}_{\text{DES}}$)**
*Let* $M \in$ DPDA.
*Then* $[\![M]\!]^{\text{FPDA-}\omega}_{\text{DES}} = \langle\!| \,\text{noteLL}(\text{L}_{\text{um}}(M)), \text{noteLL}(\text{L}_{\text{m}}(M)) \,|\!\rangle \in \text{DES}$ *where* $\text{noteLL}(A) = \{w \circlearrowleft \mid w \in A \wedge \exists d \in \mathcal{D}_{\text{I}}(M), N \in \mathbf{N} \,.\, \forall k \geq N \,.\, \text{out}(d(k)) = w\}$.

We can conclude that when finite sequences of $\lambda$-steps are not to be observed (by using $[\![\cdot]\!]^{\text{FPDA-}\lambda}_{\text{LGraph}}$) and lifelocks are made visible in the DES (by using $[\![\cdot]\!]^{\text{FPDA-}\omega}_{\text{DES}}$), then for two DPDA the observable operational behavior coincides iff their DES representations have equivalent observable operational behaviors.

**Theorem 2 ($[\![\cdot]\!]^{\text{FPDA-}\omega}_{\text{DES}}$ is Fully Abstract w.r.t. $[\![\cdot]\!]^{\text{FPDA-}\lambda}_{\text{LGraph}}$)**
*Let* $M^1, M^2 \in$ DPDA.
*Then,* $[\![M^1]\!]^{\text{FPDA-}\lambda}_{\text{LGraph}} \cong [\![M^2]\!]^{\text{FPDA-}\lambda}_{\text{LGraph}}$ *iff* $[\![[\![M^1]\!]^{\text{FPDA-}\omega}_{\text{DES}}]\!]^{\text{DES}}_{\text{LGraph}} \cong [\![[\![M^2]\!]^{\text{FPDA-}\omega}_{\text{DES}}]\!]^{\text{DES}}_{\text{LGraph}}$.

Finally, since the encodings preserve the observable behavior (except for the $\circlearrowleft$-appending encoding $[\![\cdot]\!]^{\text{FPDA-}\omega}_{\text{DES}}$), they preserve in particular the (un)marked languages.

**Corollary 2 (Preservation of (un)marked languages)**
*If* $F \in \{[\![\cdot]\!]^{\text{DES}}_{\text{LGraph}}, [\![\cdot]\!]^{\text{FPDA}}_{\text{LGraph}}, [\![\cdot]\!]^{\text{FPDA-}\lambda}_{\text{LGraph}}, [\![\cdot]\!]^{\text{FPDA}}_{\text{DES}}\}$
*then* $\text{L}_{\text{m}}(X) = \text{L}_{\text{m}}(F(X))$ *and* $\text{L}_{\text{um}}(X) = \text{L}_{\text{um}}(F(X))$.
*Furthermore, for* $[\![\cdot]\!]^{\text{FPDA-}\omega}_{\text{DES}}$*:*
$\text{L}_{\text{m}}(X) = \text{L}_{\text{m}}([\![X]\!]^{\text{FPDA-}\omega}_{\text{DES}}) \cap \Sigma^*$ *and* $\text{L}_{\text{um}}(X) = \text{L}_{\text{um}}([\![X]\!]^{\text{FPDA-}\omega}_{\text{DES}}) \cap \Sigma^*$.

### 2.2.3 *Adequate Encodings for* FPDA

To determine adequate encodings of all FPDA, the uncovering of lifelocks can be extended to the uncovering of all formerly invisible steps (then also including non-determinism, $\diamond$-steps, and possibly even finite sequences of $\lambda$-steps which were hidden in the previous subsection). For such an encoding, a similar soundness theorem can be formulated. However, we will explain in Remark 2 in Section 2.3 that this encoding fails to be satisfactory for the task of properly reducing the synthesis problem by means of an OSCP as some operational criteria are no longer enforced on realizations of controllers.

## 2.3 (Operational) Supervisory Control Problem

According to the previous section, lifelocks (i.e., infinite $\lambda$-sequences) are observable when using the trace abstraction $[\![\cdot]\!]_{\text{DES}}^{\text{FPDA}^\omega}$. This allows us to specify an operational SCP (OSCP) preventing lifelocks in FPDA realizations of constructed controllers. Before introducing the OSCP we formalize the SCP as introduced in [17] in our notation using DES as a fundamental model rather than the marked language alone.

**Definition 13 (SCP)**
*Let* $P, S \in \text{DES}$ *be a plant and a specification.*
*Then* $\text{SCP}(P,S)$ *contains the least restrictive controllers* $C \in \text{DES}$ *w.r.t.*
*(i)* $\text{L}_\text{m}(C \times P) \subseteq \text{L}_\text{m}(S)$,
*(ii)* $C \times P$ *is language-*$\Sigma_{\text{uc}}$*-controllable w.r.t.* $P$*, and*
*(iii)* $C \times P$ *is language-blockfree.*
*Least restrictive means that* $C' \times P \leq C \times P$ *for any* $C' \in \text{DES}$ *satisfying (i)–(iii).*

Based on the SCP, we introduce the following OSCP.

**Definition 14 (Operational SCP (OSCP))**
*Given a plant* $P \in \text{DFA}$ *and a specification* $S \in \text{DPDA}$.
*Then* $\text{OSCP}(P,S)$ *is the set of all* $C \in \text{DPDA}$ *satisfying* $[\![C]\!]_{\text{DES}}^{\text{FPDA}^\omega} \in \text{SCP}([\![P]\!]_{\text{DES}}^{\text{FPDA}}, [\![S]\!]_{\text{DES}}^{\text{FPDA}})$.

Since the closed-loop construction relies on the synchronization of a controller $C \in \text{DPDA}$ (broader classes for $C$ from FPDA are only considered in Remark 2) and a plant $P \in \text{DFA}$ ($P \in \text{DFA}$ throughout this chapter), we give such a construction which returns a DPDA. This construction is quite similar to the synchronous composition of DPDA with DFA from, for example, [10, page 135].

**Definition 15 (FPDA-DFA-Synchronous Composition)**
*Let* $M^1 = (Q^1, \Sigma, \Gamma, \delta^1, q_0^1, \square, F^1, \diamond) \in \text{FPDA}$.
*Let* $M^2 = (Q^2, \Sigma, \Gamma', \delta^2, q_0^2, \square', F^2, \diamond') \in \text{DFA}$.
*Then* $M^1 \times M^2 = M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F, \diamond)$ *is given by*
*(i)* $Q = Q^1 \times Q^2$
*(ii)* $((q_1,q_2), w, s, s', (q_1', q_2')) \in \delta$ *iff* $(q_1, w, s, s', q_1') \in \delta^1$ *and either* $(w \in \Sigma$ *and* $(q_2, w, \square', \square', q_2') \in \delta^2)$ *or* $(w \in \{\lambda, \diamond\}$ *and* $q_2 = q_2')$,
*(iii)* $q_0 = (q_0^1, q_0^2)$*, and*
*(iv)* $F = F^1 \times F^2$.

The main criteria for reasonability of the OSCP definition are the properties of closed loops for controllers which are solutions to the OSCP. After giving such relevant criteria in the following definition, we present our first main result.

**Definition 16 (Properties of FPDA)**
*Let* $M^1, M^2 \in \text{FPDA}$. *Then*

*(i)* $M^1$ *has a lifelock iff for some infinite* $d \in \mathcal{D}_\text{I}(M^1)$ *there is an* $N \in \mathbf{N}$ *such that the unmarked language of* $d$ *is constant after* $N$ *(i.e., for all* $k \geq N$*:* $\text{out}(d(N)) = \text{out}(d(k))$*),*

*(ii)* $M^1$ *is operational-blockfree iff for any finite* $d_i \in \mathcal{D}_\text{I}(M^1)$ *of length* $n \in \mathbf{N}$ *ending in* $d_i(n) = (e, c)$ *there is a continuation* $d_c \in \mathcal{D}(M^1)$ *such that* $d_i \cdot_n d_c$ *is a marking derivation and* $d_i$ *and* $d_c$ *match at the gluing point* $n$ *(i.e.,* $d_c(0) = (\bot, c)$*),*

*(iii)* $M^1$ *is operational-satisfying* $M^2$ *iff for any* $d_1 \in \mathcal{D}_\text{I}(M^1)$ *of length* $n_1$ *ending in a marking state, there is some* $d_2 \in \mathcal{D}_\text{I}(M^2)$ *of length* $n_2$ *ending in a marking state such that* $\text{out}(d_1(n_1)) = \text{out}(d_2(n_2))$*, and*

*(iv)* $M^1$ *is operational-*$\Sigma_\text{uc}$*-controllable w.r.t.* $M^2$ *iff whenever* $d_1 \in \mathcal{D}_\text{I}(M^1)$ *is of length* $n_1 \in \mathbf{N}$ *ending in* $d_1(n_1) = (e_1, c_1)$*,* $d_2 \in \mathcal{D}_\text{I}(M^2)$ *is of length* $n_2 \in \mathbf{N}$*,* $\text{out}(d_1(n_1)) = \text{out}(d_2(n_2))$*,* $d_2(n_2) \vdash_{M^2} (e_2, c_2)$*,* $\text{out}(e_2, c_2) = \text{out}(d_2(n_2))u$*, and* $u \in \Sigma_\text{uc}$*, then there is a continuation* $d_c \in \mathcal{D}(M^1)$ *of length* $n_3 \in \mathbf{N}$ *such that* $\text{out}((d_1 \cdot_{n_1} d_c)(n_1 + n_3)) = \text{out}(d_2(n_2))u$ *and* $d_1$ *and* $d_c$ *match at the gluing point* $n_1$ *(i.e.,* $d_c(0) = (\bot, c_1)$*).*

Incidentally (mainly due to the determinism), all of the above properties are satisfied for closed loops generated from solutions to the OSCP from Definition 14.

**Theorem 3 (Further Properties of OSCP controllers)**
*Given a plant* $P \in \text{DFA}$ *and a specification* $S \in \text{DPDA}$.
*Let* $C \in \text{OSCP}(P, S)$.
*Then*

*(i)* $C \times P$ *is operational-satisfying* $S$*,*

*(ii)* $C \times P$ *is operational-*$\Sigma_\text{uc}$*-controllable w.r.t.* $P$*,*

*(iii)* $C \times P$ *has no lifelocks, and*

*(iv)* $C \times P$ *is operational-blockfree.*

**Remark 2 (Nonextendability of Theorem 3)**
*Theorem 3 cannot be extended to* $S \in \text{NFA}$ *or* $S \in \text{FPDA}$ *which are not* $\diamond$*-step-free since language-blockfreeness is insufficient for operational-blockfreeness in general (see* $M_0$ *and* $M_1$ *in Figure 2.4 page 13).*

We have integrated enough information into the DES representations of DPDA and DFA to be able to reuse the unmodified SCP. The more direct approach, of modifying the SCP to include conditions enforcing, e.g., lifelockfreeness, operational blockfreeness, or even more advanced properties where cost-optimal controllers are to be synthesized, is nontrivial because the validity of the SCP has to be verified as well. Even with extra conditions, the set of sound controllers must be closed under arbitrary union, which is not true if for example lifelockfreeness is to be enforced (for example, if $\tau \in \Sigma_\text{c}$ represents a $\lambda$-step, the supremal language of bounded executions of $\tau$ allows lifelock executions of $\tau$: $\cup\{\{\tau^n\} \mid n \in \mathbf{N}\} = \{\tau\}^*$).

Using Theorem 2 we can conclude that all closed loops for solutions to the OSCP have equivalent observable behavior; therefore, the concrete choice of a controller realization is irrelevant.

**Theorem 4 (OSCP Solutions are Equivalent)**
*Given a plant* $P \in \text{DFA}$*, a specification* $S \in \text{DPDA}$*, and* $C_1, C_2 \in \text{OSCP}(P, S)$.
*Then* $[\![C_1 \times P]\!]^{\text{FPDA-}\lambda}_{\text{LGraph}} \cong [\![C_2 \times P]\!]^{\text{FPDA-}\lambda}_{\text{LGraph}}$.

## 2.4 SCP Characterizations via Suprema

In this section we formalize the SCP over DES by defining sound and maximal (i.e., least restrictive) controllers and compare this formalization to the supremal characterization of [14]. The DES based representations are used in Section 2.5 to verify solvers for the SCP which are also adequate for DPDA specifications. These solvers deterministically generate the sound and maximal controllers which are additionally smallest (i.e., contain the least set of words).

**Definition 17 (Sound, Maximal, and Smallest Solutions)**
*Given a plant* $P \in \mathrm{DES}$ *and a specification* $S \in \mathrm{DES}$.
*Let* $C \in \mathrm{DES}$.
*Then (by using Equations (2.1)–(2.3) in Table 2.2 on page 18)*

*(i)* *$C$ is a sound controller iff the closed loop $P \times C$ is safe in the sense of (i)–(iii) from Definition 13, formally $C \in \mathcal{S}_{\mathrm{sat}}(P, S)$,*
*(ii)* *$C$ is a maximal controller iff the closed loop $P \times C$ is sound and less restrictive than any other safe closed loop, formally $C \in \mathcal{S}_{\max}(P, S)$, and*
*(iii)* *$C$ is a smallest maximal controller iff it is a maximal controller and any strictly smaller maximal controller produces a more restrictive closed loop, formally $C \in \mathcal{S}^{\min}_{\max}(P, S)$.*

We explain the basic differences between these types of controllers by an example.

**Example 1 (Comparison of Solutions)**
*Let $P$ be a language-blockfree plant. Let $S = \langle\!|\Sigma^*, \Sigma^*|\!\rangle$ be a specification. The controller $C_1 = \langle\!|\emptyset, \emptyset|\!\rangle$ is sound but not maximal. The controller $C_2 = S$ is maximal. The controller $C_3 = P$ is smallest maximal. While $C_1$ is not desirable (and just defined for presentation purposes), there is no difference between $C_2$ and $C_3$ w.r.t. the overall goal of the SCT to determine a controller enforcing the desired observable operational behavior on the closed loop. For implementation on a physical device, $C_2$ is for $P \neq S$ more compact and requires therefore less space. Nevertheless, the solvers, we are aware of, produce the controller $C_3$. The problem of determining the size-optimal automata realization of some sound and maximal controller is left for the future.*

Similarly to Theorem 4 we can state that the plant $P$ is consistently restricted by all controllers $C \in \mathcal{S}_{\max}(P, S)$.

**Theorem 5 (Consistent Restriction)**
*Given a plant* $P \in \mathrm{DES}$ *and a specification* $S \in \mathrm{DES}$.
*Let* $C, C' \in \mathcal{S}_{\max}(P, S)$.
*Then* $P \times C = P \times C'$.

$$C \in \mathcal{S}_{\text{sat}}(P,S) \text{ iff } \begin{pmatrix} \wedge\textbf{❶}\, \mathrm{L_m}(P{\times}C) \subseteq \mathrm{L_m}(S) \\ \wedge\textbf{❷}\, \mathrm{LCont}(\mathrm{L_{um}}(P{\times}C), \mathrm{L_{um}}(P), \Sigma_{\text{uc}}) \\ \wedge\textbf{❸}\, \mathrm{L_{um}}(P{\times}C) \subseteq \overline{\mathrm{L_m}(P{\times}C)} \end{pmatrix} \tag{2.1}$$

$$C \in \mathcal{S}_{\max}(P,S) \text{ iff } \begin{pmatrix} \wedge\textbf{❶} C \in \mathcal{S}_{\text{sat}}(P,S) \\ \wedge\textbf{❷} \forall C' \in \mathcal{S}_{\text{sat}}(P,S)\ .\ P{\times}C' \leq P{\times}C \end{pmatrix} \tag{2.2}$$

$$C \in \mathcal{S}^{\min}_{\max}(P,S) \text{ iff } \begin{pmatrix} \wedge\textbf{❶} C \in \mathcal{S}_{\max}(P,S) \\ \wedge\textbf{❷} \forall C' \in \mathcal{S}_{\max}(P,S)\ .\ C' < C \rightarrow P{\times}C' < P{\times}C \end{pmatrix} \tag{2.3}$$

$$\Phi_{\text{mm}}(A_m, S_m) \triangleq \begin{pmatrix} \wedge\textbf{❶} A_m \subseteq S_m \\ \wedge\textbf{❷}\, \mathrm{LCont}(\overline{A_m}, \mathrm{L_{um}}(P), \Sigma_{\text{uc}}) \end{pmatrix} \tag{2.4}$$

$$\Phi_{\text{um}}(A_{um}, S_m) \triangleq \begin{pmatrix} \wedge\textbf{❶} A_{um} \subseteq \overline{S_m} \\ \wedge\textbf{❷}\, \mathrm{LCont}(A_{um}, \mathrm{L_{um}}(P), \Sigma_{\text{uc}}) \\ \wedge\textbf{❸} A_{um} = \overline{A_{um}} \\ \wedge\textbf{❹} A_{um} \subseteq \overline{A_{um} \cap S_m} \end{pmatrix} \tag{2.5}$$

$$C \in \mathcal{L}_{\text{mm}}(P,S) \text{ iff } \begin{pmatrix} \wedge\textbf{❶}\, \mathrm{L_m}(P{\times}C) = \cup\{A \mid \Phi_{\text{mm}}(A, \mathrm{L_m}(S))\} \\ \wedge\textbf{❷}\, \mathrm{L_{um}}(P{\times}C) = \overline{\mathrm{L_m}(P{\times}C)} \end{pmatrix} \tag{2.6}$$

$$C \in \mathcal{L}_{\text{um}}(P,S) \text{ iff } \begin{pmatrix} \wedge\textbf{❶}\, \mathrm{L_{um}}(P{\times}C) = \cup\{A \mid \Phi_{\text{um}}(A, \mathrm{L_m}(S))\} \\ \wedge\textbf{❷}\, \mathrm{L_m}(P{\times}C) = \mathrm{L_m}(S) \cap \mathrm{L_{um}}(P{\times}C) \end{pmatrix} \tag{2.7}$$

$$\mathcal{L}^{\text{si}}_{\text{mm}}(P,S) \triangleq \mathcal{L}_{\text{mm}}(P, P{\times}S) \tag{2.8}$$

$$\mathcal{L}^{\text{si}}_{\text{um}}(P,S) \triangleq \mathcal{L}_{\text{um}}(P, P{\times}S) \tag{2.9}$$

$$C \in \mathcal{L}^{\text{si}}_{\text{mm}}(P,S) \text{ iff } \begin{pmatrix} \wedge\textbf{❶}\, \mathrm{L_m}(C) = \cup\{A \mid \Phi_{\text{mm}}(A, \mathrm{L_m}(P{\times}S))\} \\ \wedge\textbf{❷}\, \mathrm{L_{um}}(C) = \overline{\mathrm{L_m}(P{\times}C)} \end{pmatrix} \tag{2.10}$$

$$C \in \mathcal{L}^{\text{si}}_{\text{um}}(P,S) \text{ iff } \begin{pmatrix} \wedge\textbf{❶}\, \mathrm{L_{um}}(C) = \cup\{A \mid \Phi_{\text{um}}(A, \mathrm{L_m}(P{\times}S))\} \\ \wedge\textbf{❷}\, \mathrm{L_m}(C) = \mathrm{L_m}(S) \cap \mathrm{L_{um}}(P{\times}C) \end{pmatrix} \tag{2.11}$$

$$C \in \mathcal{D}_{\text{mm}}(P,S) \text{ iff } P{\times}C = \mathrm{Sup}(\{\langle\!|\overline{A}, A|\!\rangle \mid \Phi_{\text{mm}}(A, \mathrm{L_m}(S))\}) \tag{2.12}$$

$$C \in \mathcal{D}_{\text{um}}(P,S) \text{ iff } P{\times}C = \mathrm{Sup}(\{\langle\!|A, \mathrm{L_m}(S) \cap A|\!\rangle \mid \Phi_{\text{um}}(A, \mathrm{L_m}(S))\}) \tag{2.13}$$

$$\mathcal{D}^{\text{si}}_{\text{mm}}(P,S) \triangleq \mathcal{D}_{\text{mm}}(P, P{\times}S) \tag{2.14}$$

$$\mathcal{D}^{\text{si}}_{\text{um}}(P,S) \triangleq \mathcal{D}_{\text{um}}(P, P{\times}S) \tag{2.15}$$

Table 2.2.: Sound, maximal, and smallest maximal controllers: Equations (2.1)–(2.3); language based supremal closed loops: Equations (2.6)–(2.9); DES based supremal closed loops: Equations (2.12)–(2.15). Sets of controllers obtained by suprema over languages (DES) are denoted here by $\mathcal{L}(\mathcal{D})$ with markers.

### 2.4.1 *Language-Based Characterization by Suprema*

Ramadge and Wonham have, based on the supremum over languages in Equation (2.6), introduced the desired marked language of the desired closed loop and by assuming language-blockfreeness also the unmarked language of the desired closed loop in [14] . In Equation (2.6) we have given the marked-maximal solution $\mathcal{L}_{\text{mm}}$ in our notation which is based on the supremum over the marked languages of controller candidates. While Equation (2.6) has the advantage to be compact and obviously sound (being directly related to, for example, Definition 17) we propose the alternative characterization in Equation (2.7) which is based on a supremum over the unmarked language of the desired closed loop. Primary differences are the translation from marked to unmarked languages using the prefix-closure operator and the different enforcement of blockfreeness. Both characterizations of the desired closed loop are equivalent and produce the desired closed-loop behavior. The alternative characterization is advantageous because, e.g., the effective solver in the DFA-setting (presented in [17, Lemma 5.1]) removes unmarked words with controllability and blocking problems. The supremal characterization $\mathcal{L}_{\text{um}}(P, S)$ thereby describes the operation of the fixed-point algorithms more precisely.

**Theorem 6**
$\mathcal{L}_{\text{mm}}(P, S) = \mathcal{L}_{\text{um}}(P, S) \subseteq \mathcal{S}_{\text{max}}(P, S)$

Effective solvers usually produce deterministically a unique result which is the smallest maximal sound controller. This is achieved by simplifying the input by restricting the specification to the behavior of the plant: this is done in Equations (2.8) and (2.9) where the simple input marked maximal and simple input unmarked maximal solutions $\mathcal{L}^{\text{si}}_{\text{mm}}$ and $\mathcal{L}^{\text{si}}_{\text{um}}$ are defined. Incidentally, the controllers described with this restriction are all contained in $\mathcal{S}^{\text{min}}_{\text{max}}$.

**Theorem 7**
$\mathcal{L}^{\text{si}}_{\text{mm}}(P, S) = \mathcal{L}^{\text{si}}_{\text{um}}(P, S) \in \mathcal{S}^{\text{min}}_{\text{max}}(P, S)$

Furthermore, the controllers in $\mathcal{L}^{\text{si}}_{\text{mm}}$ and $\mathcal{L}^{\text{si}}_{\text{um}}$ produce identical closed loops as the controllers in $\mathcal{L}_{\text{mm}}$ and $\mathcal{L}_{\text{um}}$.

A commonality of the four characterizations in Equations (2.6)–(2.9) is that they are based on characterizations of the closed loop and not on the controller. In Equations (2.10) and (2.11) we explain that the actual result as obtained by $\mathcal{L}^{\text{si}}_{\text{mm}}$ and $\mathcal{L}^{\text{si}}_{\text{um}}$ is the controller and the closed loop which basically follows from the adherence of the closed loop to the specification due to its intersection with the plant.

### 2.4.2 DES-*Based Characterization by Suprema*

Furthermore, the characterizations in Equations (2.6)–(2.9) do not properly reflect the execution of iterative solvers which modify both, the marked and the unmarked language (using operations from Figure 2.6). In our formal approach we handle the modifications to both languages explicitly, in contrast to [14] who focus on the modifications to the marked language exclusively, by including the statements on the unmarked (in the case of $\mathcal{L}_{\text{mm}}$) and marked language (in the case of $\mathcal{L}_{\text{um}}$) into the supremum statement using the lattice of DES.

**Proposition 2** $\mathcal{D}_{\text{mm}}(P, S) = \mathcal{D}_{\text{um}}(P, S) = \mathcal{L}_{\text{mm}}(P, S)$ □

The results on the set-based characterization can easily be transferred to the DES-based characterization. In the next section we decompose a supremum into a greatest fixed point of a composed operation.

## 2.5 SCP Characterizations via Greatest Fixed-Points

Usually, the desired controller is calculated by iterative application of a function $F : \text{DES} \to \text{DES}$. Let $\mathcal{F}$ denote the set of all these iterators ($\mathcal{F} = \text{DES}^{\text{DES}}$). Good iterators have the property that they do not skip beyond greatest fixed points which is achieved by including the last property (v) in the next definition. If $F \in \mathcal{G}(F_{\text{inp}}, F_{\text{fp}}, F_{\text{out}})$ in the following definition, then

(i) $F_{\text{inp}}$ specifies the set of DES to which $F$ should only be applied to,
(ii) $F_{\text{fp}}$ specifies the set of DES (from $F_{\text{inp}}$) for which $F$ returns its input, and
(iii) $F_{\text{out}}$ specifies the set of DES which are returned by $F$ (when executed on a DES from $F_{\text{inp}}$).

**Definition 18 (Good Iterator)**
*$F \in \mathcal{F}$ is a good iterator on $F_{\text{inp}}, F_{\text{fp}}, F_{\text{out}} \subseteq \text{DES}$ (written $F \in \mathcal{G}(F_{\text{inp}}, F_{\text{fp}}, F_{\text{out}})$) iff whenever $X, Y \in F_{\text{inp}}$ then*
*(i)* $F(X) \leq X$,
*(ii)* $X \in F_{\text{fp}}$ *iff* $F(X) = X$,
*(iii)* $F(X) \in F_{\text{out}}$,
*(iv) if* $X \leq Y$ *then* $F(X) \leq F(Y)$, *and*
*(v) if* $F(X) < X$, $Y < X$, $F(Y) = Y$, *then* $Y \leq F(X)$.

Good iterators are closed under unconditional $\circ$ and conditional $\rhd$ composition, used to obtain fixed point algorithms computing the desired solutions in Section 2.5.1.

**Lemma 3 (Composition of Good Iterators using $\circ$)**
*Let* $F \in \mathcal{G}(F_{\text{inp}}, F_{\text{fp}}, F_{\text{out}})$.
*Let* $G \in \mathcal{G}(G_{\text{inp}}, G_{\text{fp}}, G_{\text{out}})$.
*If* $F_{\text{out}} \subseteq G_{\text{inp}}$, *then* $G \circ F \in \mathcal{G}(F_{\text{inp}}, F_{\text{fp}} \cap G_{\text{fp}}, G_{\text{out}})$.

**Definition 19 (Conditional Composition)**
*Let* $F, G \in \mathcal{F}$.
*Then* $G \rhd F = \lambda x.$ if $F(x) = x$ then $x$ else $G(F(x))$.

**Lemma 4 (Composition of Good Iterators using $\rhd$)**
*Let* $F \in \mathcal{G}(F_{\text{inp}}, F_{\text{fp}}, F_{\text{out}})$.
*Let* $G \in \mathcal{G}(G_{\text{inp}}, G_{\text{fp}}, G_{\text{out}})$.
*If* $F_{\text{out}} \subseteq G_{\text{inp}}$ *and* $F_{\text{inp}} \subseteq G_{\text{fp}}$, *then* $G \rhd F \in \mathcal{G}(F_{\text{inp}}, \lambda X.$ if $F(X) = X$ then $X \in F_{\text{fp}} \cap F_{\text{inp}}$ else $X \in F_{\text{fp}} \cap G_{\text{fp}}, (F_{\text{inp}} \cap F_{\text{fp}} \cap F_{\text{out}}) \cup G_{\text{out}})$.

Good iterators do not skip beyond greatest fixed points.

**Corollary 3 (Good Iterators do not Skip a GFP)**
*Let* $F \in \mathcal{G}(\text{DES}, F_{\text{fp}}, \text{DES})$ *and* $Y \in \text{DES}$.
*Then* $\text{gfp}(\lambda X.F(X \times Y)) = \text{gfp}(\lambda X.F(X \times F(Y)))$.

The actual synthesis computation is then given by the following algorithm $\mathcal{U}$.

$$
\begin{aligned}
\mathrm{op}_{\mathrm{cw}}(w, U) &= \forall u \in U \,.\, w{\cdot}u \in \mathrm{L_{um}}(P) \rightarrow w{\cdot}u \in \mathrm{L_{um}}(C) \\
\mathrm{op}_{\mathrm{c}}(A, U) &= \forall w' \in A \,.\, \mathrm{op}_{\mathrm{cw}}(w', U, \mathrm{L_{um}}(P), \mathrm{L_{um}}(C)) \\
&\quad \text{where } A = \cup\{A \mid A = \overline{A} \wedge A \subseteq \mathrm{L_{um}}(D)\} \\
\mathrm{F_{bf}}(D) &= \langle\!| \overline{\mathrm{L_m}(D)}, \mathrm{L_m}(D) |\!\rangle \\
\mathrm{F_{c1}}(D) &= \langle\!| A, \mathrm{L_m}(D) \cap A |\!\rangle \\
&\quad \text{where } A = \{w \in \mathrm{L_{um}}(D) \mid \mathrm{op_c}(\overline{\{w\}}, \Sigma^*_{\mathrm{uc}})\} \\
\mathrm{F_{c2}}(D) &= \langle\!| A, \mathrm{L_m}(D) \cap A |\!\rangle \\
&\quad \text{where } A = \{w \in \mathrm{L_{um}}(D) \mid \mathrm{op_c}(\overline{\{w\}}, \Sigma_{\mathrm{uc}})\} \\
\mathrm{F_{c3}}(D) &= \langle\!| A, B |\!\rangle \\
&\quad \text{where } B = \{w \in \mathrm{L_m}(D) \mid \mathrm{op_c}(\overline{\{w\}}, \Sigma_{\mathrm{uc}})\} \\
&\quad \text{and } A = \{w \in \mathrm{L_{um}}(D) \mid \begin{pmatrix} \wedge \mathrm{op_c}(\overline{\{w\}} \setminus \{w\}, \Sigma_{\mathrm{uc}}) \\ \wedge (\neg\, \mathrm{op_{cw}}(w, \Sigma_{\mathrm{uc}}) \rightarrow w \notin \overline{B}) \end{pmatrix}\} \\
\mathrm{F_{spec}}(D) &= D \times S
\end{aligned}
$$

Figure 2.6.: The iterators $\mathrm{F_{bf}}$, $\mathrm{F_{c1}}$, $\mathrm{F_{c2}}$, $\mathrm{F_{c3}}$, and $\mathrm{F_{spec}}$ where the plant $P$ and the specification $S$ are omitted parameters.

**Definition 20 (Universal Computation)**
*Let* $F \in \mathcal{F}$ *and* $D \in \mathrm{DES}$.
*Then* $\mathcal{U}(F, D) = (\text{if } D = F(D) \text{ then } D \text{ else } \mathcal{U}(F, F(D)))$.

Then, greatest fixed points are calculated (assuming termination) as follows.

**Theorem 8 ($\mathcal{U}$ computes the GFP)**
*Let* $F \in \mathcal{G}(\mathrm{DES}, F_{\mathrm{fp}}, \mathrm{DES})$.
*If* $\mathcal{U}(F, \top)$ *terminates, then* $\mathcal{U}(F, \top) = \mathrm{Sup}(F_{\mathrm{fp}}) = \mathrm{gfp}(F)$.

We conclude that it is sufficient for concrete applications (e.g., DFA and DPDA) to verify the goodness of the used iterator to prove that the generated controller solves the SCP. Furthermore, the composition lemmas above give a proof-strategy by composition of a good iterator from multiple good iterators: this is exemplified in the next subsection where we introduce concrete iterators. As demonstrated in Section 2.5.1, it is occasionally advantageous to execute the universal algorithm on some value different from the $\top$ element.

**Theorem 9 (Initialized $\mathcal{U}$ computes the GFP)**
*Let* $F \in \mathcal{G}(\mathrm{DES}, F_{\mathrm{fp}}, F_{\mathrm{out}})$.
*If* $\mathcal{U}(F, S)$ *terminates,*
*then* $\mathcal{U}(F, S) = \mathrm{Sup}(\{X \mid X \in F_{\mathrm{fp}}, X \leq S\}) = \mathrm{gfp}(\lambda X.F(X \times S))$.

### 2.5.1 *Examples of Good Iterators*

The iterators to be discussed are given in Table 2.6.

▶*Blockfreeness:* The iterator $F_{bf}$ generates the least restrictive DES that is blockfree and contained in the input. An implementation of this iterator is presented in [17, Lemma 5.1] for DFA and in Chapter 4 for DPDA. Let $\Phi_{bf}(D) = L_{um}(D) \subseteq L_m(D)$.

**Lemma 5**
$F_{bf} \in \mathcal{G}(\text{DES}, \text{DES} \cap \Phi_{bf}, \text{DES} \cap \Phi_{bf})$.

▶$\star$*-Controllability:* The iterator $F_{c1}$ generates the least restrictive DES that is controllable and contained in the input. Let $\Phi_{cont}(D) = \text{LCont}(L_{um}(D), L_{um}(P), \Sigma_{uc})$.

**Lemma 6**
*Given a plant* $P \in \text{DES}$ *over the uncontrollable symbols* $\Sigma_{uc}$.
$F_{c1} \in \mathcal{G}(\text{DES}, \text{DES} \cap \Phi_{cont}, \text{DES} \cap \Phi_{cont})$.

▶1-*Controllability:* The difference between $\star$-controllability and the 1-controllability is that $F_{c1}$ removes controllability problems for every finite sequence of uncontrollable symbols whereas $F_{c2}$ only removes single-step controllability problems. Thereby, $F_{c2}$ may produce "new" controllability problems which are not removed in the single application of $F_{c2}$. Therefore, iterative application of $F_{c2}$ always produces $F_{c1}$: Formally, $\text{gfp}(\lambda X.F_{c2}(X \times Y)) = F_{c1}(Y)$. An implementation of this iterator is presented in [17, Lemma 5.1] for DFA.

**Lemma 7**
*Given a plant* $P \in \text{DES}$ *over the uncontrollable symbols* $\Sigma_{uc}$.
$F_{c2} \in \mathcal{G}(\text{DES}, \text{DES} \cap \Phi_{cont}, \text{DES})$.

▶*M-Controllability:* $F_{c3}$ requires a blockfree input to ensure the welldefinedness of the resulting DES. For $F_{c3}$, an unmarked word (which is not also a marked word) with a controllability problem (but where all strict prefixes are controllable) is not removed, if it cannot be extended to the created marked language. Therefore, $F_{c3}$ is equivalent to $F_{c2}$ up to blockfreeness: $F_{bf} \circ F_{c3} = F_{bf} \circ F_{c2}$. An implementation of this iterator is presented in Chapter 3 for DPDA.

**Example 2 (Differences between $F_{c1}$, $F_{c2}$ and $F_{c3}$)**
*Let* $D = \langle\!|\overline{\{aua\}}, \{aua\}|\!\rangle$ *and* $P$ *as given in Figure 2.1. Then* $F_{c1}(D) = \langle\!|\{\lambda\}, \emptyset|\!\rangle$, $F_{c2}(D) = \langle\!|\overline{\{a\}}, \emptyset|\!\rangle$, *and* $F_{c3}(D) = \langle\!|\overline{\{au\}}, \emptyset|\!\rangle$. *For* $F_{c3}$*: the word* $au$ *is not removed because all its strict prefixes are controllable and even though it is not controllable it can also not be extended into the created marked language* $\emptyset$.

**Lemma 8**
*Given a plant* $P \in \text{DES}$ *over the uncontrollable symbols* $\Sigma_{uc}$.
$F_{c3} \in \mathcal{G}(\text{DES} \cap \Phi_{bf}, \text{DES} \cap \Phi_{cont}, \text{DES})$.

▶*Satisfaction of the Specification:* Finally, the iterator $F_{spec}$ generates the least restrictive DES that is contained in the input and the specification. Let $\Phi_{spec}(D) = D \leq S$.

**Lemma 9**
*Given a specification* $S \in \text{DES}$.
*Then* $F_{spec} \in \mathcal{G}(\text{DES}, \text{DES} \cap \Phi_{spec}, \text{DES} \cap \Phi_{spec})$

Using the above properties on the individual good iterators, we can compose three good iterators.

**Theorem 10 (Computation of $\mathcal{D}^{\mathbf{si}}_{\mathbf{mm}}$)**
*Given a plant $P \in \mathrm{DES}$ and a specification $S \in \mathrm{DES}$.*
*Let $F_c \in \{\mathrm{F_{c1}}, \mathrm{F_{c2}}, \mathrm{F_{c3}}\}$.*
*Let $F = F_c \circ \mathrm{F_{bf}} \circ \mathrm{F_{spec}}$.*
*Then $F \in \mathcal{G}(\mathrm{DES}, \{\langle\!|\overline{A}, A|\!\rangle \mid \Phi_{\mathrm{mm}}(A, \mathrm{L_m}(S))\}, \mathrm{DES})$.*
*Finally, if $\mathcal{U}(F, \top)$ terminates, then $\mathcal{U}(F, \top) = \mathcal{D}^{\mathrm{si}}_{\mathrm{mm}}(P, S)$.*

That is, we have verified an approach, up to termination and fixed-point-detection, which synthesizes a least restrictive controller. However,

(i) while the operation $\mathrm{F_{spec}}$ is neutral from the second iteration onwards, this cannot be formally captured in the context of the complete lattice and therefore an implementation which follows the universal computation strictly has to re-execute $\mathrm{F_{spec}}$ in every iteration and
(ii) as $\mathrm{F_{c3}}$ requires a blockfree input, we are forced to operate with the input assumption of blockfree DES.

While there is a decision procedure ([15]) to determine whether our implementation of $\mathrm{F_{bf}}$ modifies its input, we are using the $\rhd$-composition in Theorem 11 to avoid the execution of such a computationally expensive equivalence test. We obtain the following result, which is also adequate for the DFA setting.

**Theorem 11 (DPDA-Computation of $\mathcal{D}^{\mathbf{si}}_{\mathbf{mm}}$)**
*Given a plant $P \in \mathrm{DES}$ and a specification $S \in \mathrm{DES}$.*
*Let $I = \mathrm{F_{bf}}(\mathrm{F_{spec}}(P))$ where the restricted specification $P \times S$ is used.*
*Let $F = \mathrm{F_{bf}} \rhd \mathrm{F_{c3}}$.*
*Then $F \in \mathcal{G}(\mathrm{DES} \cap \Phi_{\mathrm{bf}}, \mathrm{DES} \cap \Phi_{\mathrm{bf}} \cap \Phi_{\mathrm{cont}}, \mathrm{DES} \cap \Phi_{\mathrm{bf}})$.*
*Finally, if $\mathcal{U}(F, I)$ terminates, then $\mathcal{U}(F, I) = \mathcal{D}^{\mathrm{si}}_{\mathrm{mm}}(P, S)$.*

The last theorem states that the universal algorithm, started with the initial argument $I$ and iteratively executing the implementations of $\mathrm{F_{bf}}$ and $\mathrm{F_{c3}}$ presented in Chapter 4 and Chapter 3, determines (up to termination) the smallest maximal controller defined by the suprema-based characterization $\mathcal{D}^{\mathrm{si}}_{\mathrm{mm}}(P, S)$.

## 2.6 Conclusion

We have introduced a methodology for the extension of the SCP to domains broader than DFA. The technical problems resulting in such extensions are exemplified by considering DPDA-specifications which allow for $\lambda$-steps which are unobservable (when part of finite sequences) or observable (when part of infinite sequences). By means of nondeterminism and finalization (using an end-of-output marker $\diamond$) we have shown that the operational criteria (specifically operational blockfreeness) are more suitable than the trace-abstract criteria as they are strictly more demanding and adequate w.r.t. the task of characterization of the desired observable operational behavior (see Remark 2). While we have investigated how the operational criteria are satisfied by using certain encodings of the involved finite automata models, future research may also allow for the direct extension of the SCP by additional trace-abstract criteria. This work may also be extended by

(i) adding further operational criteria as for example worst-case-execution times and
(ii) by establishing initial results on size-optimal realizations of constructed controllers.

Finally, we intend to prove the results on the encodings in Sections 2.2 and 2.3 in the interactive theorem prover Isabelle/HOL [13] as it is already done for automata models from Section 2.1 and the results of Sections 2.4 and 2.5.

# 3. Enforcing Controllability Least Restrictively for DPDA

This chapter presents an algorithm to calculate the largest controllable marked sublanguage of a given deterministic context free language (DCFL) by least restrictively removing controllability problems in a DPDA realization of this DCFL.

## 3.1 Introduction

Ramadge and Wonham ([17]) presented an implementable fixed point algorithm to calculate the desired marked closed loop language $L_{\mathrm{clm}}$ using finite automaton representations of the involved languages and therefore restricting its applicability to regular plant and specification languages. This fixed point algorithm iteratively executes the following:

(i) it removes controllability problems, i.e., situations where the controller attempts to prevent uncontrollable events.

(ii) It resolves blocking issues, i.e., situations where the closed loop cannot reach a marking state.

Obviously, step (i) may generate new blocking issues, while step (ii) may lead to new controllability problems. The algorithm terminates iff no more controllability problems or blocking situations are present. In this chapter an algorithm which realizes step (i) for DPDA is given.

The chapter is structured as follows. After introducing all required notation in Section 3.2, we summarize the necessary parts of SCT in Section 3.3. In Section 3.4, we present an algorithm working on DPDA, realizing step (i) of the fixed point algorithm.

## 3.2 Preliminaries

Let $\Sigma$ be the external alphabet. Then $\Sigma^*$ denotes the set of all finite-length strings over symbols from $\Sigma$. Furthermore, we use the abbreviations $\Sigma^+ = \Sigma^* \setminus \{\lambda\}$ and $\Sigma^{\leq 1} = \Sigma \cup \{\lambda\}$, where $\lambda$ is the empty string. Throughout this chapter, we denote elements of the set $\Sigma^{\leq 1}$ by $\sigma$, i.e., $\sigma$ can also be the empty string. We denote the *projection* of a tuple or string $a$ on its $i$th element by $\pi_i(a)$ and the concatenation of two strings $w, w' \in \Sigma^*$ by $w{\cdot}w'$, meaning that $w'$ is appended to $w$. The *prefix relations* on strings are defined by $w \sqsubseteq w'$ if $\exists w'' \in \Sigma^* \,.\, w{\cdot}w'' = w'$ and by $w \sqsubset w'$ if $\exists w'' \in \Sigma^+ \,.\, w{\cdot}w'' = w'$. Any subset of $\Sigma^*$ is called a *language*. The *prefix closure* of a language $L$ is defined by $\overline{L} = \{w \in \Sigma^* \mid \exists w' \in L \,.\, w \sqsubseteq w'\}$. A language $L_1$ is *nonblocking* w.r.t. a language $L_2$ iff $L_1 \subseteq \overline{L_2}$. On an alphabet $\Sigma$ partitioned into controllable (in the sense of preventable) and uncontrollable events, i.e., $\Sigma = \Sigma_c \cup \Sigma_{uc}$, $\Sigma_c \cap \Sigma_{uc} = \emptyset$, a prefix-closed language $L_1 = \overline{L_1}$ is *controllable* w.r.t. a prefix-closed language $L_2 = \overline{L_2}$ iff $(L_1{\cdot}\Sigma_{uc}) \cap L_2 \subseteq L_1$. In

particular, a word $w \in L_1$ is controllable w.r.t. $L_2$ (written $\mathrm{ContW}(L_1, L_2, \Sigma_{uc}, w)$) iff

$$\forall \mu \in \Sigma_{uc} \,.\, w\mu \in L_2 \rightarrow w\mu \in L_1. \tag{3.1}$$

A *discrete event system* (DES) is a tuple $D = (\Sigma_c, \Sigma_{uc}, L_{um}, L_m)$, where $L_{um} = \overline{L_{um}} \subseteq \Sigma^*$ is the prefix closed unmarked language modeling the step by step evolution of the system while the marked language $L_m \subseteq L_{um}$ models only the satisfactory words. Since all DES in this chapter evolve on the same external alphabet $\Sigma = \Sigma_c \cup \Sigma_{uc}$, we can characterize $D$ by its marked and unmarked language only and write $D = (\!| L_{um}, L_m |\!)$. We say that $D$ is nonblocking, iff $L_{um}$ is nonblocking w.r.t. $L_m$.

A *pushdown automaton* (PDA) is a tuple

$$M := (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$$

s.t.

(i) the state set $Q$, the external alphabet $\Sigma$, the stack alphabet $\Gamma$, and the set of transitions $\delta$ are finite,
(ii) $\delta \subseteq Q \times \Sigma^{\leq 1} \times \Gamma \times \Gamma^* \times Q$,
(iii) the end-of-stack marker $\square$ is contained in $\Gamma$,
(iv) the set of marking states $F$ and the initial state $q_0$ are contained in $Q$ and
(v) $\square$ is never removed from the stack (i.e., $(q, \sigma, \square, s, q')$ implies[1] $\exists s' \in \Gamma^* \,.\, s = s' \cdot \square$).

**Example 3**
*Consider the external alphabet $\Sigma = \{a, b, c, d, u\}$, the stack alphabet $\Gamma = \{\square, \bullet\}$, the state set $Q = \{q_0, \ldots, q_5\}$ and the set of marking states $F = \{q_2, q_5\}$. Then $M_O$ in Figure 3.1 is a* PDA, *and the transition $(q, \sigma, \gamma, s, q') \in \delta$ is depicted by an edge from $q$ to $q'$ labeled by $\sigma, \gamma, s$, denoting that by taking this transition, $\sigma \in \Sigma^{\leq 1}$ is generated, $\gamma \in \Gamma$ (called "stack-top") is popped from the top of the stack, and $s \in \Gamma^*$ is pushed onto the stack (with the right-most symbol first). Two transitions with the same pre and post state are depicted by one edge with two labels.*

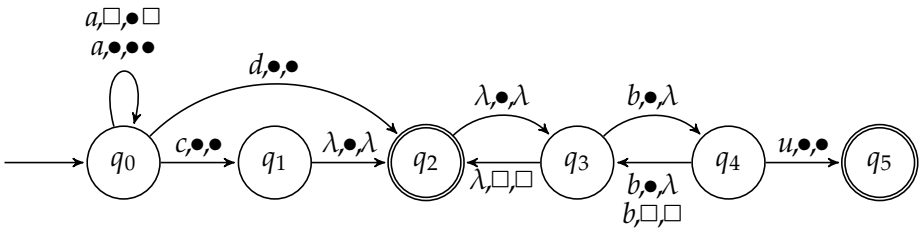



Figure 3.1.: PDA $M_O$ in Example 3.

Note that $M$ can do silent moves (called $\lambda$-transitions), possibly modifying the stack but not generating an external symbol (e.g., see Figure 3.1, from $q_1$ to $q_2$). We collect the starting states of all $\lambda$-transitions in the set

$$Q_\lambda(M) \triangleq \{q \in Q \mid (q, \lambda, \gamma, s, q') \in \delta\}.$$

A PDA $M$ is *life-lock free* (written $M \in \mathrm{LLF}$) iff there exists no reachable infinite sequence of $\lambda$-transitions. Since life-lock free PDA and PDA are equally expressive, we only consider $M \in \mathrm{LLF}$.

[1] $s$ is always pushed onto the stack with the right-most symbol first.

From a system theoretic point of view, a system state would be a pair $(q, s)$, where $q \in Q$ and $s \in \Gamma^*$ represents the current stack content. Hence, from this point of view, a PDA with $|Q| < \infty$ has infinite state space since the stack is not restricted. The pair $(q, s)$ is sometimes referred to as a configuration. For our purposes, it turns out to be convenient to add the string of generated external symbols to this pair. We therefore define the set of *configurations* of $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F) \in \text{PDA}$ by

$$\mathcal{C}(M) \triangleq Q \times \Sigma^* \times \Gamma^*$$

where $(q, w, s) \in \mathcal{C}(M)$, consists of a state $q$, a history variable $w$ (storing the external symbols generated), and a stack variable $s$ (storing the current stack content). The initial configuration is $(q_0, \lambda, \square)$. Now observe that a transition from one configuration to another occurs when $M$ takes a transition $e \in \delta$. To make this step visible we choose to include the taken transition $e$ into the definition of a transition from one configuration to another. The *single-step transition relation* $\vdash_M \subseteq ((\delta \cup \{\bot\}) \times \mathcal{C}(M))^2$ is therefore defined by

$$(e, (q, w, \gamma \cdot s')) \vdash_M (e', (q', w \cdot \sigma, s \cdot s')) \text{ for } e' = (q, \sigma, \gamma, s, q'),$$

where $\bot$ denotes an undefined[2] pre-transition.

The set $\mathcal{D}(M)$ contains all finite-length[3] *derivations* $f : \mathbf{N} \rightharpoonup ((\delta \cup \{\bot\}) \times \mathcal{C}(M))$ s.t.

$$\forall n < \max(\text{dom}(f)) \,.\, f(n) \vdash_M f(n+1)$$

with $\text{dom}(f)$ being the domain of $f$. The set $\mathcal{D}_{\text{I}}(M)$ contains all elements of $\mathcal{D}(M)$ starting with the initial element $f(0) = (\bot, (q_0, \lambda, \square))$. Furthermore, the set $\mathcal{D}_{max}(M, w)$ of maximal finite-length derivations of a word $w \in \Sigma^*$ is defined by

$$\mathcal{D}_{max}(M, w) := \left\{ f \in \mathcal{D}_{\text{I}}(M) \middle| \begin{pmatrix} \wedge \exists e, q, s \,.\, f(\max(\text{dom}(f))) = (e, (q, w, s)) \\ \wedge \nexists e', q', s' \,.\, (e, (q, w, s)) \vdash_M (e', (q', w, s')) \end{pmatrix} . \right\}$$

The set of reachable configurations is defined by

$$\mathcal{C}_{\text{reach}}(M) \triangleq \{c \in \mathcal{C}(M) | \exists d \in \mathcal{D}_{\text{I}}(M), e, n \,.\, d(n) = (e, c)\} \,.$$

Using this definition, we define the *marked and the unmarked languages* generated by $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F) \in \text{PDA}$ by

$$\text{L}_{\text{m}}(M) \triangleq \{w \mid (q, w, s) \in \mathcal{C}_{\text{reach}}(M) \wedge q \in F\} \text{ and} \tag{3.2}$$

$$\text{L}_{\text{um}}(M) \triangleq \{w \mid (q, w, s) \in \mathcal{C}_{\text{reach}}(M)\}, \tag{3.3}$$

respectively. The class of languages generated by PDA is the class of context free languages (CFL). We say that a PDA $M$ realizes a DES $D = \langle\!| L_{\text{um}}, L_{\text{m}} |\!\rangle$, iff $\text{L}_{\text{m}}(M) = L_{\text{m}}$ and $\text{L}_{\text{um}}(M) = L_{\text{um}}$ and, with some abuse of terminology, we say $D \in \text{CFL}$ iff $L_{\text{um}}, L_{\text{m}} \in \text{CFL}$.

[2] We use the dummy symbol $\bot$ to define the initial transition, and, occasionally, when the pre-transition is irrelevant for the context.

[3] since $M \in \text{LLF}$

A PDA $M$ is a *deterministic pushdown automaton* (DPDA) (written $M \in$ DPDA) iff distinct steps from a reachable configuration append distinct elements of $\Sigma$ to the history variable. Note that this implies that the existence of an outgoing $\lambda$-transition in state $q$ requiring stack-top $\gamma$ prevents other outgoing transition in $q$ requiring stack-top $\gamma$. The class of languages generated by DPDA is the class of deterministic context free languages (DCFL).

**Example 4**
*The* PDA $M_O$ *depicted in Figure 3.1 is also a* DPDA *since all required properties hold. Furthermore, the marked and unmarked language of* $M_O$ *are given by*

$$\begin{aligned} L_m(M_O) = \{&ac, aac, a^{2k+1}c(bb)^l, a^{2k+2}c(bb)^l, \\ &a^{2k+3}c(bb)^l bu, a^{2k+2}c(bb)^l bu, \\ &ad, a^{2k}d(bb)^l, a^{2k+1}d(bb)^l, a^{2k+2}d(bb)^l bu, \\ &a^{2k+1}d(bb)^l bu \mid k,l \in \mathbf{N}, k > 0, l \leq k-1\} \end{aligned}$$

*and* $L_{um}(M_O) = \overline{L_m(M_O)}$, *since no blocking situations occur in* $M_O$. *This can be verified in Figure 3.1*

*(i) by checking that in every non-marking state* $\tilde{q}$ *either*
  *(a) an outgoing transition exists for both stack-tops* $\square$ *and* $\bullet$, *or*
  *(b) the configuration* $(\tilde{q}, \cdot, \gamma)$ *is not reachable if no outgoing transition with stack-top* $\gamma$ *exists (e.g.,* $(q_1, \cdot, \square)$ *is not reachable) and*
*(ii) observing that no dead locks or infinite loops visiting only non-marking states exist.*

A *nondeterministic finite automaton* (NFA) can be viewed as a special PDA which does neither have $\lambda$-transitions, nor a stack (see Figure 3.2). Therefore, we can formally define a PDA $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$ to be an NFA (written $M \in$ NFA) iff whenever $(q, \sigma, \gamma, s, q') \in \delta$, then $\gamma = s = \square$ and $\sigma \in \Sigma$. Additionally, *deterministic finite automata* (DFA) are NFA which are deterministic. The class of languages generated by DFA is the class of regular languages (REG)[4].

**Example 5**
*Consider the input alphabet* $\Sigma = \{a, b, c, d, u\}$, *the state set* $Q = \{p_0, p_1, p_2, p_3\}$, *the set of marking states* $F = \{p_1, p_3\}$ *and the initial state* $p_0$. *Then the automaton* $M_P$ *in Figure 3.2 is a* DFA. *The transition* $(p, \sigma, \square, \square, p') \in \delta$ *is depicted by an edge from* $p$ *to* $p'$ *labeled by* $\sigma$.

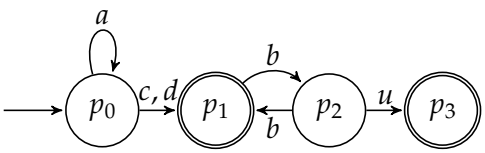



Figure 3.2.: DFA $M_P$ in Example 5.

*The marked and unmarked languages of* $M_P$ *are given by*[5]

$$L_m(M_P) = \{a^n(c+d), a^n(c+d)(bb)^m(bu+\lambda) \mid n, m \in \mathbf{N}\}$$

[4]Note that for every language $L_{um}$ generated by a NFA, there also exists a DFA generating $L_{um}$ (see [10, p.22]). However, this is not true for PDA and DPDA, implying DCFL $\subset$ CFL.
[5]The term $(x + y)$ denotes "$x$ or $y$".

*and* $\mathrm{L_{um}}(M_P) = \overline{\mathrm{L_m}(M_P)}$*, since no blocking situations occur in* $M_P$*.*

## 3.3 Supervisory Control Revisited

In the context of SCT, a controller $D_C = \langle\!| L_{\mathrm{Cum}}, L_{\mathrm{Cm}} |\!\rangle$ is a proper minimally restrictive supervisor for a plant $D_P = \langle\!| L_{\mathrm{Pum}}, L_{\mathrm{Pm}} |\!\rangle$ and a specification $D_S = \langle\!| L_{\mathrm{Sum}}, L_{\mathrm{Sm}} |\!\rangle$, if $D_C$

(i) is not preventing uncontrollable events (i.e., $((L_{\mathrm{Pum}} \cap L_{\mathrm{Cum}}) \cdot \Sigma_{\mathrm{uc}}) \cap L_{\mathrm{Pum}} \subseteq (L_{\mathrm{Pum}} \cap L_{\mathrm{Cum}})$),

(ii) generates a closed loop $D_{cl} = \langle\!| L_{\mathrm{clum}}, L_{\mathrm{clm}} |\!\rangle$, defined by $L_{\mathrm{clum}} = L_{\mathrm{Pum}} \cap L_{\mathrm{Cum}}$ and $L_{\mathrm{clm}} = L_{\mathrm{Pm}} \cap L_{\mathrm{Cm}}$, which contains as many words generated by the plant as possible while respecting the specification, and

(iii) always guides the system to a marking state (i.e., $L_{\mathrm{clum}} \subseteq \overline{L_{\mathrm{clm}}}$).

In this case $L_{\mathrm{clm}}$ is the so called marked supremal controllable (nonblocking) sublanguage of $L_{\mathrm{Pm}} \cap L_{\mathrm{Sm}}$ (see [17]). Since different controllers can generate the same closed loop, there exists no unique minimally restrictive supervisor for $D_P$ and $D_S$. Obviously, the closed loop itself is a proper supervisor, giving $D_C = \langle\!| L_{\mathrm{clum}}, L_{\mathrm{clm}} |\!\rangle$.

Ramadge and Wonham introduced the monotonic operator $\Omega : \Sigma^* \to \Sigma^*$ in [17, Lemma 2.1] defined by

$$\begin{aligned} \Omega(L_{\mathrm{m}}) &= \{w \in L_{\mathrm{m}} \mid \forall w' \sqsubseteq w \,.\, \mathrm{ContW}(\overline{L_{\mathrm{m}}}, L_{\mathrm{Pum}}, \Sigma_{\mathrm{uc}}, w')\} \\ &\subseteq L_{\mathrm{m}}. \end{aligned} \tag{3.4}$$

By iteratively applying $\Omega$, starting with $L_{\mathrm{Sm}} \cap L_{\mathrm{Pm}}$, one obtains the (unique) greatest fixed point $L_{\mathrm{clm}}$. Ramadge and Wonham showed in [17, Lemma 2.1] that for $L_{\mathrm{Sm}}, L_{\mathrm{Pm}} \in \mathrm{REG}$ an implementable algorithm working on DFA to calculate $L_{\mathrm{clm}} \in \mathrm{REG}$ exists.

Now consider plant $D_P = \langle\!| L_{\mathrm{Pum}}, L_{\mathrm{Pm}} |\!\rangle \in \mathrm{REG}$ and specification $D_S = \langle\!| L_{\mathrm{Sum}}, L_{\mathrm{Sm}} |\!\rangle \in \mathrm{DCFL}$. In this case we have $\langle\!| L_{\mathrm{Sum}} \cap L_{\mathrm{Pum}}, L_{\mathrm{Sm}} \cap L_{\mathrm{Pm}} |\!\rangle \in \mathrm{DCFL}$ (see [10, p.135]), realizable by a DPDA. Now observe the following: in one iteration of the fixed-point algorithm the language $\overline{L_{\mathrm{m}}}$ is needed to calculate $\Omega(L_{\mathrm{m}})$. When implementing this algorithm using a DPDA-realization of the DES $D = \langle\!| L_{\mathrm{um}}, L_{\mathrm{m}} |\!\rangle$, the language $\overline{L_{\mathrm{m}}}$ is (in general) not easily obtained. However, if $D$ is nonblocking, we can use $L_{\mathrm{um}} = \overline{L_{\mathrm{m}}}$. Therefore, to implement the iterative calculation of $\Omega$, starting with $L_{\mathrm{Sm}} \cap L_{\mathrm{Pm}} \in \mathrm{DCFL}$, one has to iteratively

(i) construct a DPDA $M'_O$ from a nonblocking DPDA $M_O$ s.t. $\Omega(\mathrm{L_m}(M_O)) = \mathrm{L_m}(M'_O)$ and

(ii) make $M'_O$ nonblocking.

The connection between the language characterization of the supervisory control problem and its automaton based construction using these two basic subfunctions is investigated in detail in the Chapter 2. There, explicit fixed-point constructions and soundness proofs are given and it is shown that the resulting controller can be realized by a DPDA. Note that the controller is only implementable if its stack is bounded, which is to be expected for many applications.

We show in the remainder of this chapter that there exists an algorithm working on DPDA which realizes step (i) and returns a DPDA. An algorithm realizing step (ii) is presented in Chapter 4.

**Remark 3**
*The implementable algorithm to calculate* $L_{\mathrm{clm}} \in \mathrm{REG}$ *presented in [17, Lemma 5.1] iteratively*

*calculates the unmarked controllable sublanguage of* $L_{\mathrm{um}} = \overline{L_{\mathrm{m}}}$*, i.e.,*

$$\Omega(L_{\mathrm{um}}) = \{w \in L_{\mathrm{um}} \mid \mathrm{ContW}(L_{\mathrm{um}}, L_{\mathrm{Pum}}, \Sigma_{\mathrm{uc}}, w)\},$$

*and its nonblocking sublanguage* $\overline{\Omega(\overline{L_{\mathrm{m}}}) \cap L_{\mathrm{Sm}} \cap L_{\mathrm{Pm}}}$*. In contrast, the algorithm introduced in the next section iteratively calculates the marked controllable sublanguage* $\Omega(L_{\mathrm{m}})$ *and its prefix closure.*

## 3.4 Computability of Ω for DCFL

In this section, using the DFA $M_P$ realizing $D_P = \langle\!| L_{\mathrm{Pum}}, L_{\mathrm{Pm}} |\!\rangle$, we derive a sequence of automaton manipulations starting with the DPDA $M_O$ and generating a DPDA $M'_O$ s.t. $\Omega(\mathrm{L_m}(M_O)) = \mathrm{L_m}(M'_O)$. We assume that $\mathrm{L_m}(M_O) \subseteq L_{\mathrm{Pm}}$. This is not a restriction of generality, as $\Omega$ is monotonic and its iterative application is initialized with $L_{\mathrm{Pm}} \cap L_{\mathrm{Sm}}$. We furthermore assume that $M_O$ is nonblocking, i.e., $\mathrm{L_{um}}(M_O) \subseteq \overline{\mathrm{L_m}(M_O)}$. If this assumption does not hold, the algorithm introduced in Chapter 4 is applied first.

The first task is to find all controllability problems in $M_O$. Intuitively, a controllability problem occurs when a word $w \in \Sigma^*$ is generated by $M_P$ and $M_O$ reaching states $p$ and $q$ (by using maximal derivations), respectively, and $M_P$ can generate an uncontrollable symbol $\mu \in \Sigma_{\mathrm{uc}}$ in $p$ while $M_O$ cannot generate this symbol in $q$. To obtain the states $p$ and $q$ reached when generating the same word $w \in \Sigma^*$, a product automaton is constructed, following, e.g., [10, p.135].

**Definition 21**
*Let* $M_P = (Q_P, \Sigma, \{\Box\}, \delta_P, q_{P0}, \Box, F_P) \in$ DFA.
*Let* $M_O = (Q_O, \Sigma, \Gamma, \delta_O, q_{O0}, \Box, F_O) \in$ DPDA.
*Then the product automaton* $M_\times = M_P \times M_O = (Q_\times, \Sigma, \Gamma, \delta_\times, q_{\times 0}, \Box, F_\times)$ *is given by* $Q_\times = Q_P \times Q_O$, $q_{\times 0} = (q_{P0}, q_{O0})$, $F_\times = F_P \times F_O$ *and*

$$\delta_\times = \left\{ ((p,q),\sigma,\gamma,s,(p',q')) \mid \begin{array}{l} \wedge \sigma \in \Sigma \\ \wedge (q,\sigma,\gamma,s,q') \in \delta_O \\ \wedge (p,\sigma,\Box,\Box,p') \in \delta_P \end{array} \right) \vee \left( \begin{array}{l} \wedge \sigma = \lambda \\ \wedge p = p' \\ \wedge (q,\lambda,\gamma,s,q') \in \delta_O \end{array} \right) \right\}.$$

**Lemma 10**
*Let* $M_P \in$ DFA *and* $M_O \in$ DPDA *s.t.* $\mathrm{L_m}(M_O) \subseteq \mathrm{L_m}(M_P)$ *and* $\mathrm{L_{um}}(M_O) \subseteq \overline{\mathrm{L_m}(M_O)}$.
*Then*
*(i)* $\mathrm{L_m}(M_P \times M_O) = \mathrm{L_m}(M_O)$,
*(ii)* $\mathrm{L_{um}}(M_P \times M_O) = \mathrm{L_{um}}(M_O)$,
*(iii)* $M_\times = M_P \times M_O \in$ DPDA, *and*
*(iv)* $\times$ *is implementable*[6].

**Proof 1**
*Since* $M_\times = M_P \times M_O$ *is a product automaton, as usual we have that* $\mathrm{L_m}(M_P \times M_O) = \mathrm{L_m}(M_P) \cap \mathrm{L_m}(M_O)$ *and* $\mathrm{L_{um}}(M_P \times M_O) = \mathrm{L_{um}}(M_P) \cap \mathrm{L_{um}}(M_O)$.

[6] We say an algorithm is implementable if it can be realized by a computer program.

*Then* $L_m(M_O) \subseteq L_m(M_P)$, $L_{um}(M_O) \subseteq \overline{L_m(M_O)}$, *and* $\overline{L_{um}(M_P)} = L_{um}(M_P)$ *are sufficient to conclude that* $L_{um}(M_O) \subseteq L_{um}(M_P)$, *which immediately proves (i) and (ii). (iii) and (iv) follow from [10, p.135].*

**Example 6**

*Consider the* DPDA $M_O$ *in Figure 3.1 and the* DFA $M_P$ *in Figure 3.2. It can be easily verified that* $L_m(M_O) \subseteq L_m(M_P)$, *since the state and transition structures of* $M_O$ *and* $M_P$ *are identical and the usage of a stack only prevents certain transitions. Furthermore,* $L_{um}(M_O) \subseteq \overline{L_m(M_O)}$ *from Example 4. The (accessible part of the) product automaton* $M_\times = M_P \times M_O$ *is depicted in Figure 3.3 and does obviously generate the same marked and unmarked language as* $M_O$.

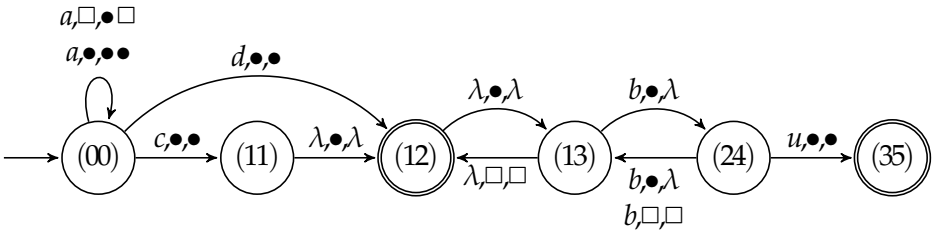



Figure 3.3.: DPDA $M_\times$ in Example 6, where $(ij) \triangleq (p_i, q_j)$.

Unfortunately, in contrast to the DFA algorithm in [17, Lemma 5.1], it is not possible to remove controllability problems in $M_O$ in a minimally restrictive fashion by deleting states or edges in $M_\times$. This is due to the following observations:

(i) it is possible that controllability problems occur in one state for a subset of possible stack-tops only (e.g., if $u \in \Sigma_{uc}$, $M_\times$ in Figure 3.3 has a controllability problem in $(p_2, q_4)$ for stack-top $\square$, only). Therefore, removing this state may falsely delete controllable words.

(ii) It is generally not possible to prevent a certain stack-top symbol in a given state by removing certain pre-transitions as one incoming transition can generate more than one stack-top[7] (e.g., the transition $((p_1, q_3), b, \bullet, \lambda, (p_2, q_4))$ in Figure 3.3 generates as stack-top the symbol which is currently in the stack underneath $\bullet$).

(iii) Controllability problems are not easily observable in states $(\tilde{p}, \tilde{q}) \in Q_\lambda$ (for example, $(p_1, q_1)$ and $(p_1, q_2)$ in Figure 3.3) as it is not possible to determine in $(\tilde{p}, \tilde{q})$ whether an uncontrollable event $\mu \in \Sigma_{uc}$ generated by $M_P$ in $\tilde{p}$ is also generated by $M_\times$ after a (finite[8]) sequence of $\lambda$-transitions starting in $(\tilde{p}, \tilde{q})$. If such a controllability problem occurs, it will be resolved at the final state $(\tilde{p}, \tilde{q}')$ of the $\lambda$-transition sequence. However, observe that $(\tilde{p}, \tilde{q}) \in F_\times$ and $(\tilde{p}, \tilde{q}') \notin F_\times$ imply that the word $\tilde{w} \in L_m(M_\times)$ with $((\tilde{p}, \tilde{q}), \tilde{w}, s), ((\tilde{p}, \tilde{q}'), \tilde{w}, s') \in \mathcal{C}_{reach}(M_\times)$ (which has a controllability problem) is not removed from the marked language by removing $(\tilde{p}, \tilde{q}')$ in $M_\times$. To remove words $w \in L_m(M_\times)$ with a controllability problem from the marked language, we will ensure that *only* its maximal derivation $f \in \mathcal{D}_{max}(M_\times, w)$ ends in a marking state.

[7] That this is the reason why the algorithm presented in [8] does not give a minimally restrictive controller.

[8] As $M_O \in$ LLF.

We therefore split states and redirect transitions in a particular way, such that deleting states with a controllability problem deletes *all* words $w \in \mathrm{L_m}(M_\times)$ *(and only those)* having a controllability problem, as required by $\Omega$ in (3.4). For this purpose we introduce four new state types: *regular* $(\cdot_r)$ and *special* $(\cdot_s)$ *main* states

$$\mathcal{M}_r(Q) \triangleq \{\langle q\rangle_r \mid q \in Q\}$$
$$\mathcal{M}_s(Q) \triangleq \{\langle q\rangle_s \mid q \in Q\}$$

and *regular* $(\cdot_r)$ and *special* $(\cdot_s)$ *auxiliary* states

$$\mathcal{A}_r(Q,\Gamma) \triangleq \{\langle q,\gamma\rangle_r \mid q \in Q \wedge \gamma \in \Gamma\}$$
$$\mathcal{A}_s(Q,\Gamma) \triangleq \{\langle q,\gamma\rangle_s \mid q \in Q \wedge \gamma \in \Gamma\}$$

where $\mathcal{M}(Q) = \mathcal{M}_r(Q) \cup \mathcal{M}_s(Q)$ and $\mathcal{A}(Q,\Gamma) = \mathcal{A}_r(Q,\Gamma) \cup \mathcal{A}_s(Q,\Gamma)$ are the sets of all main and all auxiliary states, respectively. Hence, every state is split into two entities consisting of $|\Gamma|+1$ states each. Using these new states, we define a function splitting states and redirecting transitions.

**Definition 22**
*Let* $M = (Q,\Sigma,\Gamma,\delta,q_0,\square,F)$*. Then the split automaton* $M_{Sp} = \mathrm{SPLIT}(M)$ *is defined by* $M_{Sp} = (Q_{Sp},\Sigma,\Gamma,\delta_{Sp},q_{Sp0},\square,F_{Sp})$ *with* $Q_{Sp} = \mathcal{M}(Q) \cup \mathcal{A}(Q,\Gamma)$*,* $q_{Sp0} = \langle q_0\rangle_r$*,* $F_{Sp} = (\mathcal{A}_r(F,\Gamma) \cup \mathcal{A}_s(Q,\Gamma)) \setminus Q_\lambda(M_{Sp})$ *and*

$$\delta_{Sp} = \left(\begin{array}{l} \{(\langle p\rangle_r,\lambda,\gamma,\gamma,\langle p,\gamma\rangle_r) | p \in Q,\gamma \in \Gamma\} \\ \cup\{(\langle p\rangle_s,\lambda,\gamma,\gamma,\langle p,\gamma\rangle_s) | p \in Q,\gamma \in \Gamma\} \\ \cup\left\{(\langle p,\gamma\rangle_r,\sigma,\gamma,s,\langle p'\rangle_r) \middle| \begin{pmatrix}\wedge(p,\sigma,\gamma,s,p') \in \delta \\ \wedge(p \notin F \vee \sigma \neq \lambda)\end{pmatrix}\right\} \\ \cup\{(\langle p,\gamma\rangle_r,\lambda,\gamma,s,\langle p'\rangle_s) | (p,\lambda,\gamma,s,p') \in \delta \wedge p \in F\} \\ \cup\{(\langle p,\gamma\rangle_s,\lambda,\gamma,s,\langle p'\rangle_s) | (p,\lambda,\gamma,s,p') \in \delta\} \\ \cup\{(\langle p,\gamma\rangle_s,\sigma,\gamma,s,\langle p'\rangle_r) | (p,\sigma,\gamma,s,p') \in \delta \wedge \sigma \neq \lambda\} \end{array}\right)$$

**Example 7**
*Consider the product automaton* $M_\times$ *in Example 6 depicted in Figure 3.3. By using the abbreviation in Figure 3.4, its split version* $M_{Sp} = \mathrm{SPLIT}(M_\times)$ *is depicted in Figure 3.5, where all "obviously useless" entities (i.e., main with corresponding auxiliary states that are not connected by a path (ignoring the labels) to the initial and a marking state) were removed.*

Intuitively, the newly introduced auxiliary states work as a stack-top observer, separating outgoing transitions by their required stack-top, as shown in Example 7. Therefore, if a controllability problem occurs for one stack-top, we can delete the respective auxiliary state without falsely deleting controllable words. Furthermore, observe that all main states belong to $Q_\lambda(M_{Sp})$ while, due to determinism, auxiliary states can either belong to the set $Q_\lambda(M_{Sp})$ or do not have outgoing $\lambda$-transitions at all. This uniquely defines the subset of states (i.e., $\mathcal{A}(Q,\Gamma) \setminus Q_\lambda(M_{Sp})$) for which controllability can and must efficiently be tested. Finally, the new special states are used to ensure that only the states reached by maximal derivations of words $w \in \mathrm{L_m}(M_O)$ are marking . Observe that the

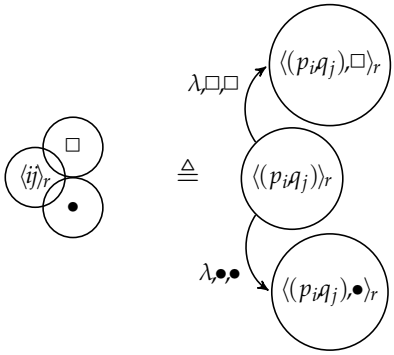



Figure 3.4.: Graphical abbreviation of one entity.

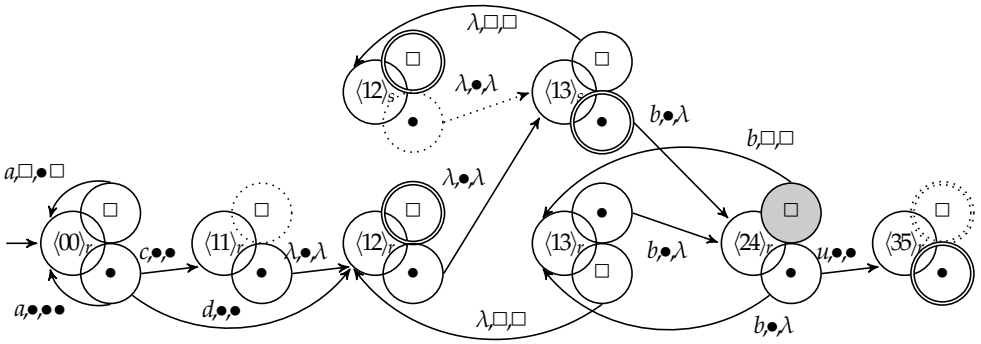



Figure 3.5.: DPDA $M_{Sp}$ in Example 7. Non-accessible states and transitions are dotted. The uncontrollable state NCS $= \{\langle(p_2, q_4), \square\rangle_r\}$ is indicated in gray.

construction in SPLIT ensures, that final states of maximal derivations always end in the set $\mathcal{A}(Q,\Gamma) \setminus Q_\lambda(M_{Sp})$, formally,

$$\begin{array}{l} \forall\, w \in \mathrm{L_{um}}(M_{Sp}), f \in \mathcal{D}_{max,M_{Sp}}(w)\ . \\ \quad \hookrightarrow\ \pi_1(\pi_2(f(\max(\mathrm{dom}(f))))) \in \big(\mathcal{A}(Q,\Gamma) \setminus Q_\lambda(M_{Sp})\big)\ . \end{array} \tag{3.5}$$

Therefore, shifting the marking into those states ensures that uncontrollable words can be removed from the marked language by removing states in the set $\mathcal{A}(Q,\Gamma) \setminus Q_\lambda(M_{Sp})$. Before removing states with controllability problems we show that SPLIT does not change the marked and unmarked language of its input.

**Lemma 11**
*Let* $M = (Q,\Sigma,\Gamma,\delta,q_0,\square,F)$ *and* $M_{Sp} = \mathrm{SPLIT}(M)$. *Then*
*(i)* $\mathrm{L_{um}}(M_{Sp}) = \mathrm{L_{um}}(M)$,
*(ii)* $\mathrm{L_m}(M_{Sp}) = \mathrm{L_m}(M)$,
*(iii)* $\mathrm{SPLIT}(M) \in \mathrm{DPDA}$, *and*
*(iv)* SPLIT *is implementable.*

**Proof 2**
*To simplify notation, we collect all states* $q$, *reachable by a finite sequence of* $\lambda$*-transitions from a configuration* $(\tilde{q},w,\tilde{s})$ *with* $\tilde{q} \in F$ *in the set*

$$Q_{rlt}(M,w) \triangleq \left\{ q \in Q \;\middle|\; \begin{array}{l} \exists\, f \in \mathcal{D}_{\mathrm{I}}(M), n \in \mathbf{N}, \tilde{q} \in F\ . \\ \quad \hookrightarrow \begin{pmatrix} \wedge f(n) = (\tilde{e},(\tilde{q},w,\tilde{s})) \\ \wedge \exists n' > n\ .\ f(n') = (e,(q,w,s)) \end{pmatrix} \end{array} \right\}.$$

***(i)*** $\mathrm{L_{um}}(M_{Sp}) = \mathrm{L_{um}}(M)$: *Let* $q \in Q$, $w \in \Sigma^*$, $\gamma \in \Gamma$, $s \in \Gamma^*$ *and* $(q,w,\gamma{\cdot}s)$ *be an arbitrary configuration reachable by the initial derivation* $f \in \mathcal{D}_{\mathrm{I}}(M)$, *implying* $w \in \mathrm{L_{um}}(M)$. *Observe, that the state* $q$ *is mapped to the states* $\{\langle q\rangle_r, \langle q,\gamma\rangle_r \mid \gamma \in \Gamma\}$ *if* $q \notin Q_{rlt}(M,w)$ *and to* $\{\langle q\rangle_s, \langle q,\gamma\rangle_s \mid \gamma \in \Gamma\}$ *if* $q \in Q_{rlt}(M,w)$. *Using this, we can easily show that the single-step relation*

$$(e,(q,w,\gamma{\cdot}s)) \vdash_M ((q,\sigma,\gamma,s',q'),(q',w{\cdot}\sigma,s'{\cdot}s)) \tag{3.6}$$

*is mimicked by a sequence of two single-step relations in* $M_{Sp}$ *in four different ways*
*(a)* $(q \notin Q_{rlt}(M,w) \wedge (\sigma \neq \lambda \vee q \notin F))$:

$$\begin{array}{l} (\check{e},(\langle q\rangle_r,w,\gamma{\cdot}s)) \vdash_{M_{Sp}} ((\langle q\rangle_r,\lambda,\gamma,\gamma,\langle q,\gamma\rangle_r),(\langle q,\gamma\rangle_r,w,\gamma{\cdot}s))\ . \\ \quad \hookrightarrow \vdash_{M_{Sp}} ((\langle q,\gamma\rangle_r,\sigma,\gamma,s',\langle q'\rangle_r),(\langle q'\rangle_r,w{\cdot}\sigma,s'{\cdot}s)) \end{array}$$

*(b)* $(q \in F \wedge \sigma = \lambda)$:

$$\begin{array}{l} (\check{e},(\langle q\rangle_r,w,\gamma{\cdot}s)) \vdash_{M_{Sp}} ((\langle q\rangle_r,\lambda,\gamma,\gamma,\langle q,\gamma\rangle_r),(\langle q,\gamma\rangle_r,w,\gamma{\cdot}s))\ . \\ \quad \hookrightarrow \vdash_{M_{Sp}} ((\langle q,\gamma\rangle_r,\sigma,\gamma,s',\langle q'\rangle_s),(\langle q'\rangle_s,w{\cdot}\sigma,s'{\cdot}s)) \end{array}$$

*(c)* $(q \in Q_{rlt}(M,w) \wedge \sigma = \lambda)$*:*

$$(\check{e},(\langle q\rangle_s,w,\gamma{\cdot}s)) \vdash_{M_{Sp}} ((\langle q\rangle_s,\lambda,\gamma,\gamma,\langle q,\gamma\rangle_s),(\langle q,\gamma\rangle_s,w,\gamma{\cdot}s)) \bullet$$
$$\hookrightarrow \vdash_{M_{Sp}} ((\langle q,\gamma\rangle_s,\sigma,\gamma,s',\langle q'\rangle_s),(\langle q'\rangle_s,w{\cdot}\sigma,s'{\cdot}s))$$

*(d)* $(q \in Q_{rlt}(M,w) \wedge \sigma \neq \lambda)$*:*

$$(\check{e},(\langle q\rangle_s,w,\gamma{\cdot}s)) \vdash_{M_{Sp}} ((\langle q\rangle_s,\lambda,\gamma,\gamma,\langle q,\gamma\rangle_s),(\langle q,\gamma\rangle_s,w,\gamma{\cdot}s)) \bullet$$
$$\hookrightarrow \vdash_{M_{Sp}} ((\langle q,\gamma\rangle_s,\sigma,\gamma,s',\langle q'\rangle_r),(\langle q'\rangle_r,w{\cdot}\sigma,s'{\cdot}s))$$

*Therefore, using induction, we can always construct a derivation* $f' \in \mathcal{D}_{\mathrm{I}}(M_{Sp})$, $n' \in \mathbf{N}$ *s.t.* $f'(n') = (e',(q',w,\gamma{\cdot}s))$ *giving* $w \in \mathrm{L_{um}}(M_{Sp})$ *and therefore,* $\mathrm{L_{um}}(M_{Sp}) \subseteq \mathrm{L_{um}}(M)$*. Using the fact that the relations in case (a)-(d) only occur, iff there exists a matching relation* (3.6) *in* $M$ *and derivations in* $M_{Sp}$ *can only be concatenated iff this is possible in* $M$*, the construction of single-step relations in* $M$ *matching single-step relations in* $M_{Sp}$ *gives the same cases as before, implying* $\mathrm{L_{um}}(M) \subseteq \mathrm{L_{um}}(M_{Sp})$*.*
***(ii)*** *Show* $\mathrm{L_m}(M_{Sp}) = \mathrm{L_m}(M)$*: Let* $w \in \mathrm{L_m}(M)$ *and* $f \in \mathcal{D}_{\mathrm{I}}(M)$, $n \in \mathbf{N}$, $q \in F$ *s.t.* $f(n) = (e,(q,w,\gamma{\cdot}s))$ *is the starting configuration in* (3.6)*. If* $\sigma \neq \lambda$ *in* (3.6)*, we know from part (i) and cases (a) and (d) that there exists a derivation* $f' \in \mathcal{D}_{\mathrm{I}}(M_{Sp})$, $n' \in \mathbf{N}$ *s.t.* $f'(n') = (\check{e},(\langle q,\gamma\rangle_r,w,\gamma{\cdot}s))$ *with* $\langle q,\gamma\rangle_r \in F_{Sp}$ *(since* $q \in F$ *and* $\langle q,\gamma\rangle_r \notin Q_\lambda(M_{Sp})$*), giving* $w \in \mathrm{L_m}(M_{Sp})$*. Now let* $\sigma = \lambda$*. Since* $M \in$ LLF*, there exists* $\sigma' \neq \lambda$ *and a finite chain of singe-step relations, s.t.*

$$(\check{e},(q,w,\gamma{\cdot}s)) \vdash_M^* (\tilde{e},(\tilde{q},w,\tilde{\gamma}{\cdot}\tilde{s})) \vdash_M ((\tilde{q},\sigma',\tilde{\gamma},\tilde{s}',\tilde{q}'),(\tilde{q}',w{\cdot}\sigma',\tilde{s}'{\cdot}\tilde{s})) \quad \textit{or} \tag{3.7}$$
$$(\check{e},(q,w,\gamma{\cdot}s)) \vdash_M^* (\hat{e},(\hat{q},w,\hat{s}{\cdot}s)) \nvdash_M . \tag{3.8}$$

*Then we can combine case (b),(c) and (d) from the proof of part (i) to mimic* (3.7) *by the finite chain*

$$(\check{e},(\langle q\rangle,w,\gamma{\cdot}s)) \vdash_{M_{Sp}}^* (\tilde{h},(\langle\tilde{q},\tilde{\gamma}\rangle_s,w,\tilde{\gamma}{\cdot}\tilde{s})) \vdash_{M_{Sp}} \bullet$$
$$\hookrightarrow ((\langle\tilde{q},\tilde{\gamma}\rangle_s,\sigma',\tilde{\gamma},\tilde{s}',\langle\tilde{q}'\rangle_r),(\langle\tilde{q}'\rangle_r,w\sigma',\tilde{s}'{\cdot}\tilde{s}))$$

*and* (3.8) *by the finite chain*

$$(\bot,(\langle q\rangle,w,\gamma{\cdot}s)) \vdash_{M_{Sp}}^* (\hat{h},(\langle\hat{q},\hat{\gamma}\rangle_s,w,\hat{\gamma}{\cdot}s)) \nvdash_{M_{Sp}} .$$

*Now observe, that all states in* $\mathcal{A}_s(Q,\Gamma) \cap Q_\lambda(Q_{Sp})$ *can* only *have outgoing* $\lambda$*-transitions, due to the determinism of* $Q$ *that is preserved in* $Q_{Sp}$*. Therefore, we have* $\langle\tilde{q},\tilde{\gamma}\rangle_s \cup \langle\hat{q},\hat{\gamma}\rangle_s \not\subseteq Q_\lambda(M_{Sp})$ *and therefore* $\langle\tilde{q},\tilde{\gamma}\rangle_s \cup \langle\hat{q},\hat{\gamma}\rangle_s \in F_{Sp}$ *by definition. This implies that there exists a derivation* $f' \in \mathcal{D}_{\mathrm{I}}(M_{Sp})$, $n'' \in \mathbf{N}$ *s.t.* $f'(n') = (h,(p,w,r))$ *s.t.* $p \in F_{Sp}$*, implying* $w \in \mathrm{L_m}(M_{Sp})$ *and therefore* $\mathrm{L_m}(\mathrm{SPLIT}(M)) \subseteq \mathrm{L_m}(M)$*. For the proof of* $\mathrm{L_m}(M) \subseteq \mathrm{L_m}(\mathrm{SPLIT}(M))$ *observe, that if* $w$ *is accepted by a state* $\langle q,\gamma\rangle_r \in F_{Sp}$*, it follows from (i) and the construction of* $F_{Sp}$ *that* $w$ *is accepted by* $q \in F$ *in* $M$*. Now let* $w$ *be accepted by a state* $\langle q,\gamma\rangle_s \in F_{Sp}$*. Then it follows from case (b) in the proof of (i) that there exists some state* $p \in F$ *that accepts* $w$*, since otherwise,* $\langle q,\gamma\rangle_s \in F_{Sp}$ *is not reachable by* $w$*. This again gives* $w \in \mathrm{L_m}(M_{Sp})$*.*
*Since we only split states, redirect existing transitions and add unique* $\lambda$*-transitions,* ***(iii)*** *and* ***(iv)*** *follow immediately from the construction.*

Technically, we are now ready, to delete all states that have a controllability problem. However, both $\times$ and SPLIT introduce non-accessible states and transitions. Trimming the automaton $M_{Sp}$ prior to the removal of controllability problems has the advantage that termination of the fixed point algorithm over $\Omega$ can be easily verified, since no states are removed, if the automaton is trim, nonblocking and no further controllability problems are present. Therefore, following [7, Thm.4.1], we introduce an algorithm to remove non-accessible states and transitions in DPDA. Here, in contrast to DFA, transitions can be non-accessible even if they connect accessible states, since the required stack-top might not be available at its pre-state.

**Definition 23**
*Let $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$. Then the accessible part $\mathrm{AC}(M)$ of $M$ is defined by $M_{Ac} = (Q_{Ac}, \Sigma, \Gamma, \delta_{Ac}, q_0, \square, F_{Ac})$ s.t. $Q_{Ac} = Q \setminus \mathrm{Q}_{\mathrm{na}}(M)$, $F_{Ac} = F \setminus \mathrm{Q}_{\mathrm{na}}(M)$ and $\delta_{Ac} = \delta \setminus \delta_{\mathrm{na}}(M)$, where*

$$\mathrm{Q}_{\mathrm{na}}(M) \triangleq \{q \in Q \setminus \{q_0\} \mid \mathrm{L_m}((Q, \Sigma, \Gamma, \delta, q_0, \square, \{q\})) = \emptyset\}$$

*are the non accessible states and*

$$\delta_{\mathrm{na}}(M) \triangleq \left\{ e = (q, x, \gamma, s, q') \in \delta \;\middle|\; \begin{array}{l} \forall\, r \notin Q, \delta' = (\delta \cup \{(q, \lambda, \gamma, \gamma, r)\}) \setminus \{e\} \;. \\ \hookrightarrow \mathrm{L_m}((Q \cup \{r\}, \Sigma, \Gamma, \delta', q_0, \square, \{r\})) = \emptyset \end{array} \right\}$$

*are the non accessible transitions.*

**Example 8**
*The accessible part of the automaton $M_{Sp}$ in Example 7, depicted in Figure 3.5, is obtained by removing the dotted states, giving $M_{Ac} = \mathrm{AC}(M_{Sp})$.*

Trimming a DPDA does not change its marked and unmarked languages.

**Lemma 12**
*Let $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$. Then*
*(i)* $\mathrm{L_m}(\mathrm{AC}(M)) = \mathrm{L_m}(M)$,
*(ii)* $\mathrm{L_{um}}(\mathrm{AC}(M)) = \mathrm{L_{um}}(M)$,
*(iii)* $\mathrm{AC}(M) \in$ DPDA, *and*
*(iv)* AC *is implementable.*

**Proof 3**
*Observe that $\mathrm{Q}_{\mathrm{na}}(M)$ is the set of states unreachable by derivations of words $w \in \mathrm{L_{um}}(M)$ and $\delta_{\mathrm{na}}(M)$ is the set of transitions that are not part of derivations of words $w \in \mathrm{L_{um}}(M)$. Therefore removing these states and transitions does not change the unmarked language, implying (ii). Since accessible marking states are preserved, also the marked language (as a subset of the unmarked language) is left unchanged, implying (i). Since $M \in$ DPDA, (iii) immediately follows as we only remove states and transitions. (iv) follows, as the emptiness of a* CFL *is decidable*[9] *(see [10, p.137]).*

[9] The algorithm testing the emptiness of a CFL is computationally very involving. The more efficient algorithm used in our `libFAUDES` implementation is presented in Chapter 4.

As the final step of our algorithm, we will now identify auxiliary states that have a controllability problem with respect to the plant, and remove them. This is done in analogy to the DFA algorithm in [17], while ignoring all states in $Q_\lambda$.

**Definition 24**
*Let* $\Sigma_{uc} \subseteq \Sigma$, $M_P \in$ DFA,
$M_O \in$ DPDA,
$M_\times = M_P \times M_O = (Q_\times, \Sigma, \Gamma, \delta_\times, q_{\times 0}, \square, F_\times)$, *and*
$M_{Ac} = \mathrm{AC}(\mathrm{SPLIT}(M_\times)) = (Q_{Ac}, \Sigma, \Gamma, \delta_{Ac}, q_0, \square, F_{Ac})$.
*Then the automaton with removed non-controllable auxiliary states is defined by* $\mathrm{RNCS}(M_{Ac}) = (Q_R, \Sigma, \Gamma, \delta_R, q_0, \square, F_R)$, *where* $Q_R = Q_{Ac} \setminus \mathrm{NCS}$, $F_R = F_{Ac} \setminus \mathrm{NCS}$ *and* $\delta_R = \{(q, x, \gamma, w, q') \in \delta' | q, q' \in Q_R\}$, *where*

$$\mathrm{NCS} := \left\{ \begin{array}{l} \langle (p,q), \gamma \rangle \in ((Q_{Ac} \cap \mathcal{A}(Q_\times, \Gamma)) \setminus Q_\lambda(M_{Ac})) \mid \bullet \\ \hookrightarrow \exists \mu \in \Sigma_{uc} \,.\, \begin{pmatrix} \wedge \exists p' \,.\, (p, \mu, \square, \square, p') \in \delta_P \\ \wedge \forall s, r \,.\, (\langle (p,q), \gamma \rangle, \mu, \gamma, s, r) \notin \delta_{Ac} \end{pmatrix} \end{array} \right\}$$

**Example 9**
*Consider the accessible split automaton* $M_{Ac}$ *in Example 8 depicted in Figure 3.5 and let* $\Sigma_{uc} = \{u\}$. *Then the set of uncontrollable auxiliary states of* $M_{Ac}$ *is given by* $\mathrm{NCS} = \{\langle (p_2, q_4), \square \rangle_r\}$, *indicated in gray in Figure 3.5. Removing this state and all ingoing and outgoing transitions generates the automaton* $M'_O = \mathrm{RNCS}(M_{Ac})$. *Observe that* $M'_O$ *has a blocking situation in* $\langle (p_2, q_4) \rangle_r$ *for all words* $w$ *whose derivations generate stack-top* $\square$ *in* $\langle (p_2, q_4) \rangle_r$, *which are* $w \in \{a^{2k+1} c (bb)^k, a^{2k} d (bb)^k \mid k > 0\}$. *Note that removing these words (and their extensions) from* $\mathrm{L_{um}}(M'_O)$, *i.e., making* $M'_O$ *nonblocking, requires, depending on the number of occurrences of* $a$ *and* $b$ *in the past, the restriction of possible future steps. This implies, that the nonblocking version of* $M'_O$ *has to be structurally different from* $M_O$ *in a non-obvious manner. This makes the problem of* automatically *removing blocking situations in* DPDA *challenging. This problem is investigated in Chapter 4.*

As our main result, we now prove that the introduced sequence of automaton manipulations removes those (and only those) marked words which have a prefix which is uncontrollable w.r.t. the plant.

**Theorem 12**
*Let* $\Sigma = \Sigma_c \cup \Sigma_{uc}$ *s.t.* $\Sigma_c \cap \Sigma_{uc} = \emptyset$.
*Let* $M_P \in$ DFA.
*Let* $M_O \in$ DPDA *s.t.* $\mathrm{L_m}(M_O) \subseteq \mathrm{L_m}(M_P)$ *and* $\mathrm{L_{um}}(M_O) \subseteq \overline{\mathrm{L_m}(M_O)}$.
*Then* $\mathrm{L_m}(\mathrm{RNCS}(\mathrm{AC}(\mathrm{SPLIT}(M_P \times M_O)))) = \Omega(\mathrm{L_m}(M_O))$.

**Proof 4**
*Using* $M_{Ac} = \mathrm{AC}(\mathrm{SPLIT}(M_P \times M_O))$ *and* $M'_O = \mathrm{RNCS}(M_{Ac})$, *we have the following observations:*

***(A)*** *From Lemma 10, 11 and 12, follows that* $\mathrm{L_{um}}(M_{Ac}) = \mathrm{L_{um}}(M_O)$ *and* $\mathrm{L_m}(M_{Ac}) = \mathrm{L_m}(M_O)$.

***(B)*** *Pick an arbitrary* $w \in \mathrm{L_{um}}(M_O)$ *satisfying* $\neg \mathrm{ContW}(\mathrm{L_{um}}(M_O), L_{\mathrm{Pum}}, \Sigma_{uc}, w)$. *Using Equation (3.5), we can fix* $f_w \in \mathcal{D}_{max, M_{Ac}}(w)$ *and the final state of* $f_w$ $\langle (p,q), \gamma \rangle \in$

$((Q_{Ac} \cap \mathcal{A}(Q_\times, \Gamma)) \setminus Q_\lambda(M_{Ac}))$. *Using Def 21 (p. 30) this implies that* $M_P$ *uniquely accepts* $w$ *in* $p$. *Now Equations (3.1) and (3.3) and A imply that there exists* $\mu \in \Sigma_{uc}$ *s.t.* $(p, \mu, \square, \square, p') \in \delta_P$ *and* $\forall s, r\ .\ (\langle (p,q), \gamma \rangle, \mu, \gamma, s, r) \notin \delta_{Ac}$, *giving that* $\langle (p,q), \gamma \rangle \in$ NCS.

**(C)** *It follows from the construction in Def 22 (p. 32) that there exist derivations generating* $w$ *and ending in a main state* $\langle (p,q) \rangle$ *and possibly also in other preceding states in* $Q_\lambda(M_{Ac})$. *Observe, that all states reached by generating* $w$, *except of* $\langle (p,q), \gamma \rangle$, *are in* $Q_\lambda(M_{Ac})$ *and therefore, by construction, not in* $F_{Ac}$.

**(D)** *Def 24 (p. 37) implies that* $w \in \mathrm{L_{um}}(M'_O)$ *iff there exists a (possibly non-maximal) derivation* $f' \in \mathcal{D}(M_{Ac})$ *with final configuration* $(r, w, s)$ *s.t. for all* $n \in \mathrm{dom}(f')$ *preceeding* $\max(\mathrm{dom}(f'))$: $\pi_2(f'(n)) \notin$ NCS. $M_{Ac} \in$ DPDA *implies that for all prefixes* $w' \sqsubset w$ *(but not for* $w$ *itself) holds that their maximal derivations are a prefix of* $f'$. *Therefore, using B implies*

$$w \in \mathrm{L_{um}}(M'_O) \rightarrow \big(\forall w' \sqsubset w\ .\ \mathrm{ContW}(\mathrm{L_{um}}(M_O), L_{\mathrm{P}um}, \Sigma_{uc}, w')\big) \quad \textit{and}$$
$$\big(\forall w' \sqsubseteq w\ .\ \mathrm{ContW}(\mathrm{L_{um}}(M_O), L_{\mathrm{P}um}, \Sigma_{uc}, w')\big) \rightarrow w \in \mathrm{L_{um}}(M'_O).$$

**(E)** *Now assume* $\neg\,\mathrm{ContW}(\mathrm{L_{um}}(M_O), L_{\mathrm{P}um}, \Sigma_{uc}, w)$ *and all* $w' \sqsubset w$ *are controllable. From B follows that* $\langle (p,q), \gamma \rangle \in$ NCS *is removed. This has two consequences:*

**(a)** *we still have* $w \in \mathrm{L_{um}}(M'_O)$ *from C, D, and*

**(b)** *if* $w \in \mathrm{L_m}(M_O)$ *we have* $w \notin \mathrm{L_m}(M'_O)$ *from C, D.*

*Combining A, D and E(b) implies*

$$w \in \mathrm{L_m}(\mathrm{RNCS}(\mathrm{AC}(\mathrm{SPLIT}(M_P \times M_O)))) \leftrightarrow \bullet$$
$$\hookrightarrow \forall w' \sqsubseteq w\ .\ \mathrm{ContW}(\mathrm{L_{um}}(M_O), L_{\mathrm{P}um}, \Sigma_{uc}, w'),$$

*which proves the statement.*

**Remark 4**

*Observation E(a) in the proof of Theorem 12 is the reason why we do not implement the removal of all uncontrollable* unmarked *words, i.e.,* $\Omega(\mathrm{L_{um}}(M_O))$, *used in [17] as discussed in Remark 3.*

## 3.5 Conclusion

In this chapter, we have presented a first step towards extending SCT to situations, where the plant is realized by a DFA, but the specification is modeled as a DPDA, i.e., the specification language is a DCFL. In particular, we have presented an algorithm consisting of a sequence of automaton manipulations which, starting with a nonblocking DPDA, removes all (and only those) marked words having a prefix which is uncontrollable w.r.t. the plant. This algorithm was implemented as a plug-in in [12]. The remaining essential part of a procedure to obtain a proper minimally restrictive supervisor, namely an algorithm removing blocking situations in DPDA, is presented in Chapter 4. The connection between a language-theoretic characterization and the automata-based implementation is investigated in detail in Chapter 2.

# 4. Enforcing Operational Properties including Blockfreeness for DPDA

We present an algorithm which modifies a deterministic pushdown automaton (DPDA) such that

(i) the marked language is preserved,
(ii) lifelocks are removed,
(iii) deadlocks are removed,
(iv) all states and edges are accessible, and
(v) operational blockfreeness is established (i.e., coaccessibility in the sense that every initial derivation can be continued to a marking configuration).

This problem can be trivially solved for deterministic finite automata (DFA) but is not solvable for standard petri net classes. The algorithm is required for an operational extension of the supervisory control problem (SCP) to the situation where the specification in modeled by a DPDA.

## 4.0 Introduction

We are introducing an algorithm to transform a DPDA such that its observable operational behavior is restricted to its desired fragment. The algorithm decomposes the problem into three steps: transformation of the DPDA into a Context Free Grammar (CFG) while preserving the operational behavior, restricting the CFG to enforce operational blockfreeness, and the transformation of the resulting CFG via Parsers to DPDA while preserving and establishing the relevant criteria on the operational behavior. The algorithm presented here is an essential part for the effective solution of the supervisory control problem for DFA plants and DPDA specifications which is reduced (in Chapter 2) to the effective implementability of ensuring blockfreeness (solved in this chapter) and ensuring controllability (solved in Chapter 3).

In Section 4.1 we define abstract transition systems (ATS) as a basis for the systems involved in the algorithm and give a formal problem statement to be solved for DPDA. In Section 4.2 we define the concrete transition systems appearing in the algorithm as instantiations of ATS. In Section 4.3 we present the extensive algorithm due to space restrictions mostly informally using a running example before we discuss the formal verification and possible improvements of the approach. The formal constructions of the algorithm are contained in Appendix A.2. We summarize our results in Section 4.4 and outline our next steps in Section 4.5.

## 4.1 Abstract Transition Systems

The concrete systems used in this chapter (including DPDA, CFG, and Parsers) are instantiations of the subsequently defined class of Abstract Transition Systems (ATS).

Thus, they will inherit the uniform definitions of derivations, languages, and the problem to be solved from the ATS definitions.

Throughout the chapter we use the following notations.

**Notation 2**
*Let $A$ be an alphabet and let $B$ be a set. Then*
*(i)* $A^*$ *denotes the set of all finite words over* $A$,
*(ii)* $A^{\leq 1} = A \cup \{\lambda\}$,
*(iii)* $A^{\omega *}$ *denotes the set of all finite and infinite words over* $A$,
*(iv)* $\cdot$ *is the (sometimes omitted) concatenation operation on words (and languages),*
*(v)* $\sqsubseteq$ *is the prefix relation,*
*(vi)* $\overline{A}$ *is the prefix-closure of* $A$,
*(vii)* $\sqsupseteq$ *is the suffix relation, and*
*(viii)* $k{:}w$ *denotes the k-Prefix of* $w \in A^*$ *which is defined by* (if $w = \alpha{\cdot}w' \wedge k > 0$ then $\alpha{\cdot}((k-1){:}w')$ else $\lambda$), *and*
*(ix)* $✠(A, B)$ *denotes* $(A \cup \{\bot\}) \times B$ *where* $\bot$ *represents undefinedness.*

**Definition 25 (Abstract Transition System)**
$\mathcal{S} = (E, C, S, \pi_S, R, c_0, A, O, o_m, o_{um}) \in$ ATS *iff*
*(i)* $E$ *is a set of step-edges,*
*(ii)* $C$ *is a set of configurations,*
*(iii)* $S$ *is a set of states,*
*(iv)* $\pi_S$ *maps each configuration to at most one state,*
*(v)* $R$ *is a binary step-relation on* $✠(E, C)$,
*(vi)* $c_0 \in C$ *is the initial configuration,*
*(vii)* $A$ *is the marking subset of* $C$,
*(viii)* $O$ *is the set of outputs, and*
*(ix)* $o_{um} : C \to 2^O$ *and* $o_m : A \to 2^O$ *define the unmarked and marked outputs for configurations.*

For these ATS we define their derivations, generated languages, and subsequently the properties to be enforced.

**Definition 26 (Semantics of ATS)**
*(i) the set of derivations* $\mathcal{D}(\mathcal{S})$ *contains all elements from* $✠(E, C)^{\omega *}$ *starting in a configuration of the form* $(\bot, c)$ *where all adjacent* $(c_1, e_1), (c_2, e_2) \in ✠(E, C)$ *satisfy* $(c_1, e_1)\ R\ (c_2, e_2)$,
*(ii) the set of initial derivations* $\mathcal{D}_{\mathrm{I}}(\mathcal{S})$ *contains all elements of* $\mathcal{D}(\mathcal{S})$ *starting with* $(\bot, c_0)$,
*(iii) the reachable configurations* $\mathcal{C}_{\mathrm{reach}}(\mathcal{S})$ *are defined by* $\{c \in C \mid \exists d \in \mathcal{D}_{\mathrm{I}}(\mathcal{S})\ .\ d(n) = (e, c)\}$,
*(iv) the marked language* $\mathrm{L_m}(\mathcal{S})$ *is defined by* $\cup o_m(F \cap \mathcal{C}_{\mathrm{reach}}(\mathcal{S}))$, *and*
*(v) the unmarked language* $\mathrm{L_{um}}(\mathcal{S})$ *is defined by* $\cup o_{um}(\mathcal{C}_{\mathrm{reach}}(\mathcal{S}))$.
*The concatenation of derivations* $d_1, d_2 \in \mathcal{D}(\mathcal{S})$ *is given by* $(d_1 \cdot_n d_2)(i) =$ (if $i \leq n$ then $d_1(i)$ else $d_2(i-n)$).

**Definition 27 (Properties of ATS)**
*(i)* $\mathcal{S}$ *has a deadlock iff for some finite* $d \in \mathcal{D}_{\mathrm{I}}(\mathcal{S})$ *of length* $n \in \mathbf{N}$ *which is not marking (i.e., for all* $k$, $d(k) = (e, c)$ *implies* $c \notin A$*) there is no* $c'$ *such that* $d(n)\ R\ c'$,
*(ii)* $\mathcal{S}$ *has a lifelock iff for some infinite* $d \in \mathcal{D}_{\mathrm{I}}(\mathcal{S})$ *there is an* $N \in \mathbf{N}$ *such that the unmarked language of* $d$ *is constant after* $N$ *(i.e., for all* $k \geq N$, $o_{um}(d(N)) = o_{um}(d(k))$*),*

*(iii)* $\mathcal{S}$ *is accessible iff for each* $p \in S$ *there is* $c \in \mathcal{C}_{\text{reach}}(\mathcal{S})$ *such that* $\pi_S(c) = p$ *and for each* $e \in E$ *there is* $d \in \mathcal{D}_{\text{I}}(\mathcal{S})$ *such that* $d(n) = (e, c)$*, and*

*(iv)* $\mathcal{S}$ *is operational blockfree iff for any finite* $d_i \in \mathcal{D}_{\text{I}}(\mathcal{S})$ *of length* $n \in \mathbf{N}$ *ending in* $d_i(n) = (e, c)$ *there is a continuation* $d_c \in \mathcal{D}(\mathcal{S})$ *such that* $d_i \cdot_n d_c$ *is a marking derivation and* $d_i$ *and* $d_c$ *match at the gluing point* $n$ *(i.e.,* $d_c(0) = (\bot, c)$*).*

By definition, for operational blockfree ATS the absence of deadlocks is guaranteed. Finally, we present the problem of enforcing the desired properties on an ATS, which will be solved for DPDA by the algorithm presented in Section 4.3.

**Definition 28 (Problem Statement for ATS)**
*Let* $\mathcal{S} \in$ ATS*. How to find* $\mathcal{S}' \in$ ATS *such that*

*(i)* $\text{L}_\text{m}(\mathcal{S}) = \text{L}_\text{m}(\mathcal{S}')$*,*
*(ii)* $\mathcal{S}'$ *is accessible,*
*(iii)* $\mathcal{S}'$ *has no deadlocks,*
*(iv)* $\mathcal{S}'$ *has no lifelocks, and*
*(v)* $\mathcal{S}'$ *is operational blockfree?*

In the DFA-setting: lifelocks can not occur and the other aspects of the problem are solved by simple and efficient graph-traversal algorithms pruning out states which are either not reachable from the initial state or from which no marking state can be reached[1].

## 4.2 Concrete Transition Systems

Every deterministic context free language can be properly represented by at least three different types of finite models: a deterministic EPDA, a context free grammar (CFG) satisfying the LR(1) determinism property, and a deterministic Parser. These three types occur at intermediate steps of our algorithm which solves the problem stated in Definition 28. Therefore, the following subsections contain their definitions as instantiations of the ATS. In each of the three cases we proceed in three steps:

1. definition of EPDA, CFG, and Parser as tuples,
2. instantiation of the ATS-scheme by defining each of the ten components, and
3. characterization of the determinism conditions.

**Remark 5**
*We provide the slightly nonstandard branching semantics*[2] *for* EPDA *and* Parsers *which utilize a history variable in the configurations to greatly simplify the definition of the operational-blockfreeness from Definition 27. Furthermore, this branching semantics corresponds to the intuition that the finite state realizations are generators rather than acceptors of languages, as it is customary in the context of supervisory control theory.*

### 4.2.1 EPDA *and* DPDA

We introduce EPDA, which are NFA enriched with a variable on which the stack-operations top, pop, and, push can be executed.

[1] The trivial handling of an ATS with empty marked language obtained at some point of the calculation is kept implicit in this chapter (in this case, no solution exists and the calculation can be aborted).

[2] The branching interpretation is already the standard for CFG.

| | EPDA | PDA | DPDA | NFA | DFA |
|---|---|---|---|---|---|
| 1-popping | | ✓ | ✓ | ✓ | ✓ |
| deterministic | | | ✓ | | ✓ |
| $\lambda$-step-free | | | | ✓ | ✓ |
| stack-free | | | | ✓ | ✓ |

Table 4.1.: Subclasses of EPDA.

**Definition 29 (Extended Pushdown Automata (EPDA))**
$M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F) \in$ EPDA *iff*
*(i)* *the states $Q$, the output alphabet $\Sigma$, the stack alphabet $\Gamma$, and the set of edges $\delta$ are finite ($Q$, $Q^*$, $\Sigma$, $\Sigma^*$, $\Gamma$, $\Gamma^*$ range over $p$, $\tilde{p}$, $\alpha$, $w$, $\gamma$, $s$, respectively),*
*(ii)* $\delta : Q \times \Sigma^{\leq 1} \times \Gamma^* \times \Gamma^* \times Q$,
*(iii)* *the end-of-stack marker $\square$ is contained in $\Gamma$,*
*(iv)* *the marking states $F$ and the initial state $p_0$ are contained in $Q$, and*
*(v)* *$\square$ is never removed from the stack (i.e., $(p, \sigma, s, s', p') \in \delta$ and $s \sqsupseteq \square$ imply $s' \sqsupseteq \square$).*

We proceed with the ATS instantiation for EPDA.

**Definition 30 (EPDA—ATS Instantiation)**
*An* EPDA $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$ *instantiates the* ATS *scheme* $(E, C, S, \pi_S, R, c_0, A, O, o_m, o_{um})$ *via:*
*(i)* $\boxed{E}\ \delta$
*(ii)* $\boxed{C}\ \mathcal{C}(M) \triangleq Q \times \Sigma^* \times \Gamma^*$ *where $(p, w, s) \in \mathcal{C}(M)$ consists of a state $p$, a history variable $w$ (storing the symbols generated), and the stack-variable $s$*
*(iii)* $\boxed{S}\ Q$
*(iv)* $\boxed{\pi_S(p, w, s)}\ p$
*(v)* $\boxed{R} \vdash_M : \maltese(\delta, \mathcal{C}(M))^2$ *defined by* $(e, (p, w, s' \cdot s)) \vdash_M ((p, \sigma, s', s'', p'), p', w \cdot \sigma, s'' \cdot s)$
*(vi)* $\boxed{c_0}\ (p_0, \lambda, \square)$
*(vii)* $\boxed{A}\ \{(p, w, s) \in \mathcal{C}(M) \mid p \in F\}$
*(viii)* $\boxed{O}\ \Sigma^*$
*(ix)* $\boxed{o_m(p, w, s), o_{um}(p, w, s)}\ \{w\}$

The well known sub-classes of EPDA having one or more of the properties below are defined in Table 4.1.

**Definition 31 (Sub-classes of EPDA)**
*An* EPDA *is 1-popping iff every edge pops precisely one element from $\Gamma$ from the stack. An* EPDA *is deterministic iff for every reachable configuration all two distinct steps append distinct elements of $\Sigma$ to the history variable*[3]*. An* EPDA *is $\lambda$-step-free iff no edge is of the form $(p, \lambda, s, s', p')$. An* EPDA *is stack-free iff every edge is of the form $(p, \alpha, \square, \square, p')$.*

### 4.2.2 CFG *and* LR(1)

A CFG (e.g., defined in [4]) is a term-replacement system replacing a nonterminal with a word over output symbols[4] and nonterminals.

[3] Thus, $\lambda$-steps may not be enabled simultaneously with other steps.
[4] The output symbols of a CFG are usually called terminals.

**Definition 32 (Context-Free Grammars (CFG))**
$G = (N, \Sigma, P, S) \in \text{CFG}$ *iff*
*(i)* *the nonterminals $N$ (ranging over $A, B$), the output alphabet $\Sigma$, and the productions $P$ are finite*
*(ii)* $P : N \times (N \cup \Sigma)^*$, *and*
*(iii)* *the initial nonterminal $S$ is contained in $N$.*
$N \cup \Sigma$ *and* $(N \cup \Sigma)^*$ *range over* $\kappa$ *and* $v$, *respectively. Productions* $(A, v)$ *are written* $A \to v$.

**Definition 33 (CFG—ATS Instantiation)**
*A* CFG $G = (N, \Sigma, P, S)$ *instantiates the* ATS *scheme* $(E, C, S, \pi_S, R, c_0, A, O, o_m, o_{um})$ *via:*
*(i)* $\boxed{E}\ P$
*(ii)* $\boxed{C}\ \mathcal{C}(G) = (N \cup \Sigma)^*$
*(iii)* $\boxed{S}\ N$
*(iv)* $\boxed{\pi_S}$ *take the first nonterminal (if present) of the configuration*
*(v)* $\boxed{R} \vdash_G : \text{✠}(P, \mathcal{C}(G))^2$ *given by* $(e, (v_1 \cdot A \cdot v_2)) \vdash_G ((A, v), v_1 \cdot v \cdot v_2)$
*(vi)* $\boxed{c_0}\ S$
*(vii)* $\boxed{A}\ \Sigma^*$
*(viii)* $\boxed{O}\ \Sigma^*$
*(ix)* $\boxed{o_m(v)}\ \{v\}$
*(x)* $\boxed{o_{um}(v)}\ \overline{\{v\}}$

The LR(1)-condition below, which corresponds to the determinism property of EPDA, depends on the restriction of the step-relation to the replacement of the right-most nonterminal which will be denoted by the index rm.

**Definition 34 (LR(1)-Condition)**
*According to [16] (page 52)*[5], LR(1) *is the set of all* CFG *for which (assuming* $x \in \Sigma^*$*)*
*(i)* $(\bot, S) \vdash_G^{\text{rm}^*} (e_1, v_1' \cdot A_1 \cdot w_1) \vdash_G^{\text{rm}} ((A_1, v_1), v_1' \cdot v_1 \cdot w_1)$,
*(ii)* $(\bot, S) \vdash_G^{\text{rm}^*} (e_2, v_2' \cdot A_2 \cdot w_2) \vdash_G^{\text{rm}} ((A_2, v_2), v_2' \cdot v_2 \cdot w_2)$,
*(iii)* $v_2' \cdot v_2 = v_1' \cdot v_1 \cdot x$, *and*
*(iv)* $1{:}w_1 = 1{:}(x \cdot w_2)$, *imply*
*(v)* $v_1' = v_2'$, $A_1 = A_2$, *and* $v_1 = v_2$.

Intuitively, if a parser for a CFG has generated the shorter prefix $v_1' \cdot v_1$ it must be able to decide by fixing the next symbol ($1{:}w_1$ and $1{:}(x \cdot w_2)$, respectively) whether $(A_1, v_1)$ is to be applied backwards or whether for $x \neq \lambda$ another symbol of $x$ should be generated or for $x = \lambda$ the production $(A_2, v_2)$ is to applied backwards[6].

### 4.2.3 Parser

Intuitively, a Parser is an EPDA with mild modifications[7]:
1. the parser may fix the next output-symbol (without generating it) and
2. the parser may terminate the generation of symbols (by fixing the end-of-output marker $\diamond$).

[5] The here relevant section 6.6 of the monograph [16] is based primarily on the work of Knuth [11] which was later extended in [1].
[6] E.g., $(\{S, A, B\}, \{a\}, \{(S, A), (S, B), (A, a), (B, a)\}, S) \notin \text{LR}(1)$.
[7] An equivalent linear/scheduled definition of Parser is given in [16].

**Definition 35 (Parser)**
$M = (N, \Sigma, S, F, P, \diamond) \in$ Parser *iff*

*(i)* *the stack alphabet $N$, the output alphabet $\Sigma$, the marking stack-tops $F$, and the rules $P$ are finite, ($N$, $N^*$, $\Sigma$, $\Sigma^*$, range over $p$, $\tilde{p}$, $\alpha$, $w$, respectively)*

*(ii)* $P : N^+ \times \Sigma^{\leq 1} \times N^+ \times \Sigma^{\leq 1}$,

*(iii)* *the initial stack symbol $S$ and the marking stack-tops $F$ are contained in $N$,*

*(iv)* *the end-of-output marker $\diamond$ is contained in $\Sigma$,*

*(v)* *the parser may not modify the output (i.e., $(s{\cdot}p, w, s'{\cdot}p', w') \in P$ implies $w \sqsupseteq w'$ (i.e., $w$ ends with $w'$)), and*

*(vi)* *the end-of-output marker $\diamond$ may not be generated (i.e., $(s{\cdot}p, w{\cdot}w', s'{\cdot}p', w') \in P$ and $w \sqsupseteq \diamond$ imply $w' \sqsupseteq \diamond$).*

*Rules $(s{\cdot}p, w{\cdot}w', s'{\cdot}p', w')$ are written $s{\cdot}p|w{\cdot}w' \to s'{\cdot}p'|w'$.*

Intuitively, a rule $s{\cdot}p|w{\cdot}w' \to s'{\cdot}p'|w'$ is changing the state from $p$ to $p'$, pops $s$ from the stack, pushes $s'$ to the stack, fixes the output $w'$, and generates $w$ to the output.

**Definition 36 (Parser—ATS Instantiation)**
*A* Parser $M = (N, \Sigma, S, F, P, \diamond)$ *instantiates the* ATS *scheme* $(E, C, S, \pi_S, R, c_0, A, O, o_m, o_{um})$ *via:*

*(i)* $\boxed{E}\ P$

*(ii)* $\boxed{C}\ \mathcal{C}(M) \triangleq N^+ \times \Sigma^* \times \Sigma^*$ *where* $(s{\cdot}p, w, f) \in \mathcal{C}(M)$ *contains the stack fragment $s$, the current state $p$, a history variable $w$, and the* fixed *part $f \in \Sigma^{\leq 1}$ which the parser fixed without generating it.*

*(iii)* $\boxed{S}\ \{\overline{p} \in N \mid (s{\cdot}p, w{\cdot}w', s'{\cdot}p', w') \in P \wedge \overline{p} \in \{p, p'\}\}$

*(iv)* $\boxed{\pi_S(s{\cdot}p, w, f)}\ \{p\}$

*(v)* $\boxed{R} \vdash_M : \maltese(P, \mathcal{C}(M)) \times \maltese(P, \mathcal{C}(M))$ *given by*
$(e, (s{\cdot}s_1{\cdot}p, w, f)) \vdash_M ((s_1{\cdot}p, w_1, s_2{\cdot}p', w_2), s{\cdot}s_2{\cdot}p', w', f')$ *where*
*(a)* $w_1 \sqsubseteq f \vee f \sqsubseteq w_1$,
*(b)* $w' = w{\cdot}\mathrm{drop}(|f|, \mathrm{delbot}(\square, w_1))$,[8] *and finally*
*(c)* $f' = w_2{\cdot}\mathrm{drop}(|w_1|, f)$.

*(vi)* $\boxed{c_0}\ (S, \lambda, \lambda)$

*(vii)* $\boxed{A}\ \{(s{\cdot}p, w, f) \mid f \in \{\lambda, \diamond\} \wedge p \in F\}$

*(viii)* $\boxed{O}\ \Sigma^*$

*(ix)* $\boxed{o_m(s{\cdot}p, w, f)}\ \{w\}$

*(x)* $\boxed{o_{um}(s{\cdot}p, w, f)}\ \{w\}$.

A Parser is deterministic iff for all reachable configuration all two distinct steps
(i) append distinct symbols to the history variable, or
(ii) one step adds a symbol to the history variable and the other step completes the output-generation by fixing the end-of-output marker $\diamond$[9].

## 4.3 Approach

▶*Motivation::* For example, the DPDA $G_0$ in Figure 4.1 exhibits

[8] Here $\mathrm{delbot}(\square, \overline{w})$ removes a potential $\diamond$ from the end of $\overline{w}$ and $\mathrm{drop}(n, \overline{w})$ drops the first $n$ symbols from $\overline{w}$.

[9] A parser may (depending on the other rules) be deterministic if $(p_1, w{\cdot}\alpha, \lambda)$ and $(p_2, w, \diamond)$ are successors of the same reachable configuration $(p, w, \lambda)$.

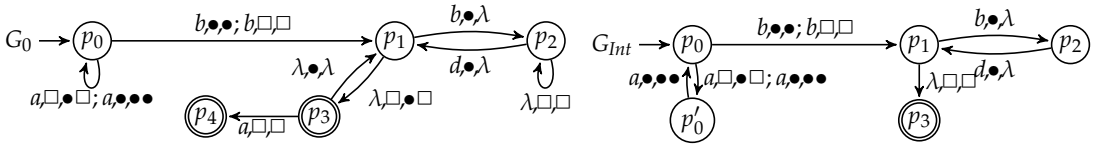



Figure 4.1.: DPDA $G_0$ and $G_{Int}$ generating $\{a^{2n}b(bd)^n \mid n \in \mathbf{N}\}$.

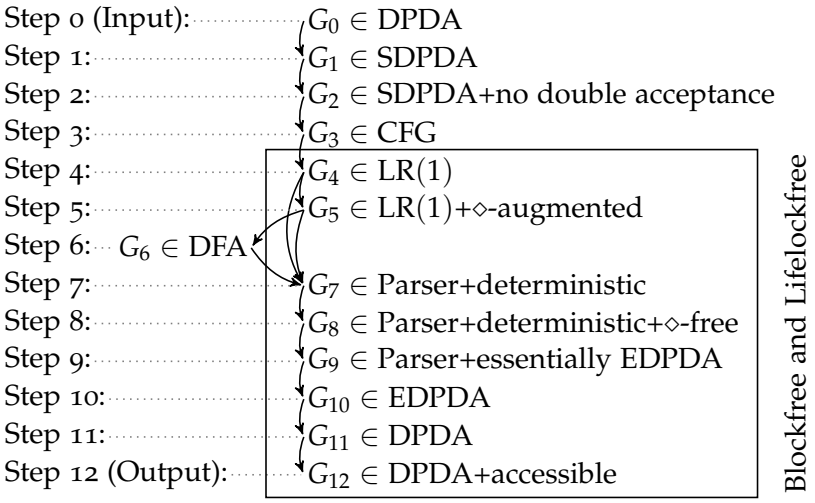



Figure 4.2.: Visualization of the algorithm.

- a lifelock generating the output $b$ reaching $p_1, p_3$ arbitrarily often,
- a lifelock (and blocking situations) generating the output $abb$ reaching $p_2$ arbitrarily often,
- a non accessible state $p_4$ (along with the edge leading to it), but
- no deadlock.

Observe that the cause ($G_0$ does not properly distinguish between an even or odd number of generated $a$s) is structurally separated from the lifelock at $p_2$. Thus, the intuitive solution $G_{Int}$ (see Figure 4.1) is obtained by splitting the state $p_0$ and by removing junk. Any formal construction must

- detect the states with a deadlock, a lifelock, or a blocking situation,
- determine the cause of that problem, and
- make a decision on how to fix the problem.

▶*Solution::* In Figure 4.2 we have depicted our approach in the subsequently explained 12 steps. The basic idea is to

- (Steps 1–3) transform the DPDA $G_0$ into a CFG $G_3$,
- (Step 4) obtain an LR(1) grammar $G_4$ by restricting $G_3$ to establish operational blockfreeness and absence of lifelocks,
- (Steps 5–11) transform the LR(1) grammar into a DPDA $G_{11}$ preserving the desired properties, and finally
- (Step 12) remove all inaccessible states and edges.

Steps 1–4 and 7–12 preserve the marked language. Steps 1–3 and 7–12 preserve the unmarked language while step 4 restricts the unmarked language to the prefix closure of the marked language. Steps 5 and 6 are not meant to preserve the (un)marked language

as they are only intermediate results of the translation in Step 7.

### 4.3.1 *Approximating Accessibility*

Throughout the following presentation we omit states and edges which are obviously inaccessible: such states and edges are detected by overapproximating the possible $\leq k$-length prefixes of stacks in reachable configurations. The $k$-overapproximation $\mathcal{R} : Q \to Q \to 2^{\Gamma^{\leq k}}$ is the least function satisfying the following rules:

(i) initial configuration: $k{:}\Box \in \mathcal{R}(p_0, p_0)$,
(ii) closure under steps: if $\gamma s \in \mathcal{R}(p, p)$ and $(p, \sigma, \gamma, s', p') \in \delta$ then $k{:}(s' \cdot s) \in \mathcal{R}(p, p')$ and $k{:}(s' \cdot s) \in \mathcal{R}(p', p')$, and
(iii) transitivity: if $s \in \mathcal{R}(p, p')$ and $s' \in \mathcal{R}(p', p'')$ then $s' \in \mathcal{R}(p, p'')$.[10]

For example, in $G_0$ the state $p_4$ is obviously inaccessible because the set of all $\leq$1-length prefixes of stacks of reachable configurations with state $p_4$ is empty. However, we would obtain $\lambda$ to be a $\leq$0-length prefix of a reachable configuration with state $p_4$; i.e., by increasing the parameter for the length of the calculated prefixes a better result may be obtained. For DFA and $k = 0$ the standard DFA-accessibility-operation is obtained. For arbitrary DPDA step 12 alone enforces accessibility.

Applying this approximation implicitly in the running example, we now describe the steps of the algorithm solving the problem stated in Definition 28.

### 4.3.2 *Step 1*

We transform the DPDA into a simple DPDA (called SDPDA subsequently) such that every edge is of one of three forms: a generating edge $(p, a, \gamma, \gamma, p')$, a pop edge $(p, \lambda, \gamma, \lambda, p')$, or a push edge $(p, \lambda, \gamma, \gamma'\gamma, p')$. The operation consists of four steps:

(i) split each edge of the form $(p, a, \gamma, s, p')$ into the two edges $(p, a, \gamma, \gamma, p'')$ and $(p'', \lambda, \gamma, s, p')$,
(ii) split each neutral edge of the form $(p, \lambda, \gamma, \gamma, p')$ into the two edges $(p, \lambda, \gamma, \circ\gamma, p'')$ and $(p'', \lambda, \circ, \lambda, p')$ for a unique fresh stack symbol $\circ \in \Gamma$,
(iii) split each rule of the form $(p, \lambda, \gamma, s\gamma', p')$ with $\gamma \neq \gamma'$ into $(p, \lambda, \gamma, \lambda, p'')$ and $(p'', \lambda, \gamma'', s\gamma'\gamma'', p')$ for every $\gamma'' \in \Gamma$, and
(iv) split every rule of the form $(p, \lambda, \gamma, s\gamma, p')$ into $|s|$ steps which push a single symbol of $s$ in each step.

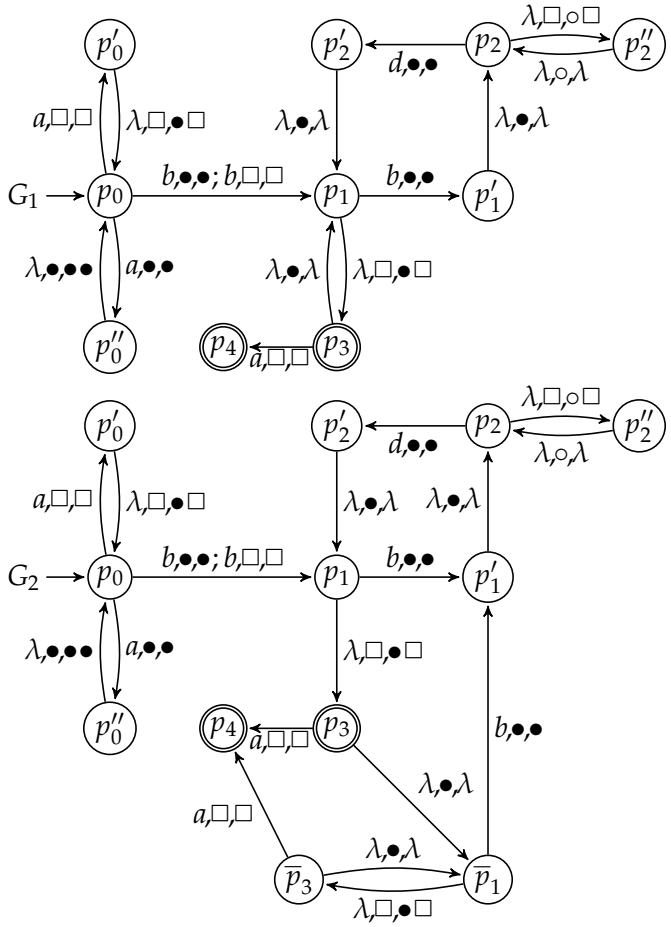
Note that the fresh states to be used in each of the four steps contain the edge for which they have been constructed (i.e., $p''$ in the first step is $(p, \sigma, \gamma, s, p')$). The operation has been adapted from [11] by

1. correcting the handling of neutral edges involving the $\Box$ symbol (for example, the self loop at $p_2$ in $G_0$ would have been handled incorrectly), and by
2. logging the involved edges in the fresh states as explained before.

For the DPDA $G_0$ from Figure 4.1 the SDPDA $G_1$ in Figure 4.3 results (up to renaming of the states).

---

[10] Without using the transitivity rule we obtain the 0- and 1-overapproximations $\mathcal{R}_0$ and $\mathcal{R}_1$ of $G_0$ (where we omit empty sets):
$\mathcal{R}_0 = \{p_0 \mapsto \{p_1 \mapsto \{\lambda\}, p_0 \mapsto \{\lambda\}\}, p_1 \mapsto \{p_2 \mapsto \{\lambda\}, p_1 \mapsto \{\lambda\}, p_3 \mapsto \{\lambda\}\}, p_2 \mapsto \{p_2 \mapsto \{\lambda\}, p_1 \mapsto \{\lambda\}\}, p_3 \mapsto \{p_1 \mapsto \{\lambda\}, p_4 \mapsto \{\lambda\}, p_3 \mapsto \{\lambda\}\}, \mathbf{p_4 \mapsto \{p_4 \mapsto \{\lambda\}\}}\}$
$\mathcal{R}_1 = \{p0 \mapsto \{p_1 \mapsto \{\Box, \bullet\}, p0 \mapsto \{\Box, \bullet\}\}, p_1 \mapsto \{p_2 \mapsto \{\lambda\}, p_1 \mapsto \{\lambda, \Box, \bullet\}, p_3 \mapsto \{\bullet\}\}, p_2 \mapsto \{p_2 \mapsto \{\lambda, \Box\}, p_1 \mapsto \{\lambda\}\}, p_3 \mapsto \{p_1 \mapsto \{\lambda\}, p_3 \mapsto \{\bullet\}\}\}$




$G_4$ with initial symbol $L_{p_0,\square}$

| | |
|---|---|
| $1{:}L_{p_0,\square} \to a \cdot L_{p_0',\square}$ | $8 :L_{p_0,\square} \to b \cdot L_{p_1,\square}$ |
| $2{:}L_{p_0',\square} \to L_{p_0,\bullet,p_1} \cdot L_{p_1,\square}$ | $9 :L_{p_0,\bullet,p_1} \to a \cdot L_{p_0'',\bullet,p_1}$ |
| $3{:}L_{p_1,\square} \to L_{p_3,\bullet}$ | $10{:}L_{p_3,\bullet} \to \lambda$ |
| $4{:}L_{p_0'',\bullet,p_1} \to L_{p_0,\bullet,p_2} \cdot L_{p_2,\bullet,p_1}$ | $11{:}L_{p_0,\bullet,p_2} \to a \cdot L_{p_0'',\bullet,p_2}$ |
| $5{:}L_{p_0,\bullet,p_2} \to b \cdot L_{p_1,\bullet,p_2}$ | $12{:}L_{p_1,\bullet,p_2} \to b \cdot L_{p_1',\bullet,p_2}$ |
| $6{:}L_{p_1',\bullet,p_2} \to \lambda$ | $13{:}L_{p_0'',\bullet,p_2} \to L_{p_0,\bullet,p_1} \cdot L_{p_1,\bullet,p_2}$ |
| $7{:}L_{p_2,\bullet,p_1} \to d \cdot L_{p_2',\bullet,p_1}$ | $14{:}L_{p_2',\bullet,p_1} \to \lambda$ |

renamed $G_4$ with initial symbol $S$

| | | | | |
|---|---|---|---|---|
| $1 :S \to aA$ | $2 :A \to BC$ | $3 :C \to D$ | $4 :E \to FJ$ | $5 :F \to bG$ |
| $6 :H \to \lambda$ | $7 :J \to dK$ | $8 :S \to bC$ | $9 :B \to aE$ | $10{:}D \to \lambda$ |
| $11{:}F \to aI$ | $12{:}G \to bH$ | $13{:}I \to BG$ | $14{:}K \to \lambda$ | |

Figure 4.3.: The simple DPDA $G_1$, the simple DPDA $G_2$ not exhibiting double marking, and the LR(1)-grammar $G_4$.

### 4.3.3 *Step 2*

We transform the SDPDA $G_1$ into an SDPDA $G_2$ such that once the SDPDA $G_2$ has generated an output, it has to generate another symbol before entering a marking state again. For the example automaton $G_1$ this means that the lifelock at $p_2, p_3$ is problematic. We are reusing the construction from [11]: Every state is duplicated (the duplicated states are neither initial nor marking). Then, the edges are defined such that the automaton $G_2$ operates on the original states until it reaches a marking state. Once this happens, the automaton either remains in the original states by using a generating edge or it switches to the duplicated states. The automaton remains in the duplicated states until switching to the original states using any generating edge. For the SDPDA $G_1$ from Figure 4.3 the SDPDA $G_2$ in the same figure results. Note, the lifelock in $p_1, p_3$ has been removed by the cost of another lifelock in $\overline{p}_1, \overline{p}_3$ generating the same output $b$.

### 4.3.4 *Step 3 & Step 4*

We transform the SDPDA $G_2$ in step 3 into the CFG $G_3$ using a construction from [11]. We restrict the CFG $G_3$ in step 4 to the LR(1) grammar $G_4$ (see Figure 4.3) by removing all productions from $G_3$ which do not appear in any marking derivation of $G_3$. That is, the accessible and coaccessible part is constructed using a fixed-point algorithm in each case. For the accessible part: the *accessible* nonterminals are the least set of nonterminals $\mathcal{A}$ such that the initial nonterminal is contained in $\mathcal{A}$ and for any production $A \to v$: if $A \in \mathcal{A}$, then the nonterminals of $v$ are contained in $\mathcal{A}$. For the coaccessible part: the *coaccessible* nonterminals are the least set of nonterminals $\mathcal{A}$ such that for any production $A \to v$: if the nonterminals of $v$ are contained in $\mathcal{A}$ then $A \in \mathcal{A}$. The equivalence of $G_2$ and $G_4$ w.r.t. the marked language can best be understood by comparing the derivations in Figure 4.4. The following three properties explain the correctness of the construction:

(i) The nonterminals of the form $L_{p,A}$ (for example $L_{p_1,\square}$) guarantee a marking derivation of the SDPDA starting in $p$ not modifying the stack starting with $A$.

(ii) The nonterminals of the form $L_{p,A,p'}$ (for example $L_{p_0,\bullet,p_2}$) guarantee a derivation of the SDPDA starting in $p$ not modifying the stack starting with $A$ and reaching a configuration in which the $A$ is removed and the state $p'$ is reached.

(iii) For any configuration $(p_1, w, \gamma_1 \dots \gamma_n)$ there are $p_2 \dots p_n$ such that the configuration $(p, w, \gamma_1 \dots \gamma_n)$ is reachable by $G_2$ iff $w \cdot L_{p_1,\gamma_1,p_2} \dots L_{p_{n-1},\gamma_{n-1},p_n} L_{p_n,\gamma_n}$ is reachable by $G_4$.

*Once step 4 has been completed, for the given* DPDA *a marked language equivalent* CFG *has been constructed which is lifelockfree, accessible, and operational blockfree (and by that deadlockfree).*

| SDPDA $G_2$ | | LR(1) $G_4$ | |
|---|---|---|---|
| | $(p_0, \lambda, \square)$ | | $L_{p_0,\square}$ |
| $\vdash_{G_2}$ | $(p'_0, a, \square)$ | $\vdash_{G_4}$ | $a \cdot L_{p'_0,\square}$ |
| $\vdash_{G_2}$ | $(p_0, a, \bullet\square)$ | $\vdash_{G_4}$ | $a \cdot L_{p_0,\bullet,p_1} \cdot L_{p_1,\square}$ |
| $\vdash_{G_2}$ | $(p''_0, aa, \bullet\square)$ | $\vdash_{G_4}$ | $aa \cdot L_{p''_0,\bullet,p_1} \cdot L_{p_1,\square}$ |
| $\vdash_{G_2}$ | $(p_0, aa, \bullet\bullet\square)$ | $\vdash_{G_4}$ | $aa \cdot L_{p_0,\bullet,p_2} \cdot L_{p_2,\bullet,p_1} \cdot L_{p_1,\square}$ |
| $\vdash_{G_2}$ | $(p_1, aab, \bullet\bullet\square)$ | $\vdash_{G_4}$ | $aab \cdot L_{p_1,\bullet,p_2} \cdot L_{p_2,\bullet,p_1} \cdot L_{p_1,\square}$ |
| $\vdash_{G_2}$ | $(p'_1, aabb, \bullet\bullet\square)$ | $\vdash_{G_4}$ | $aabb \cdot L_{p'_1,\bullet,p_2} \cdot L_{p_2,\bullet,p_1} \cdot L_{p_1,\square}$ |
| $\vdash_{G_2}$ | $(p_2, aabb, \bullet\square)$ | $\vdash_{G_4}$ | $aabb \cdot L_{p_2,\bullet,p_1} \cdot L_{p_1,\square}$ |
| $\vdash_{G_2}$ | $(p'_2, aabbd, \bullet\square)$ | $\vdash_{G_4}$ | $aabbd \cdot L_{p'_2,\bullet,p_1} \cdot L_{p_1,\square}$ |
| $\vdash_{G_2}$ | $(p_1, aabbd, \square)$ | $\vdash_{G_4}$ | $aabbd \cdot L_{p_1,\square}$ |
| $\vdash_{G_2}$ | $(p_3, aabbd, \bullet\square)$ | $\vdash_{G_4}$ | $aabbd \cdot L_{p_3,\bullet}$ |
| | | $\vdash_{G_4}$ | $aabbd$ |

Figure 4.4.: Corresponding initial derivations of the SDPDA $G_2$ and the LR(1) grammar $G_4$.

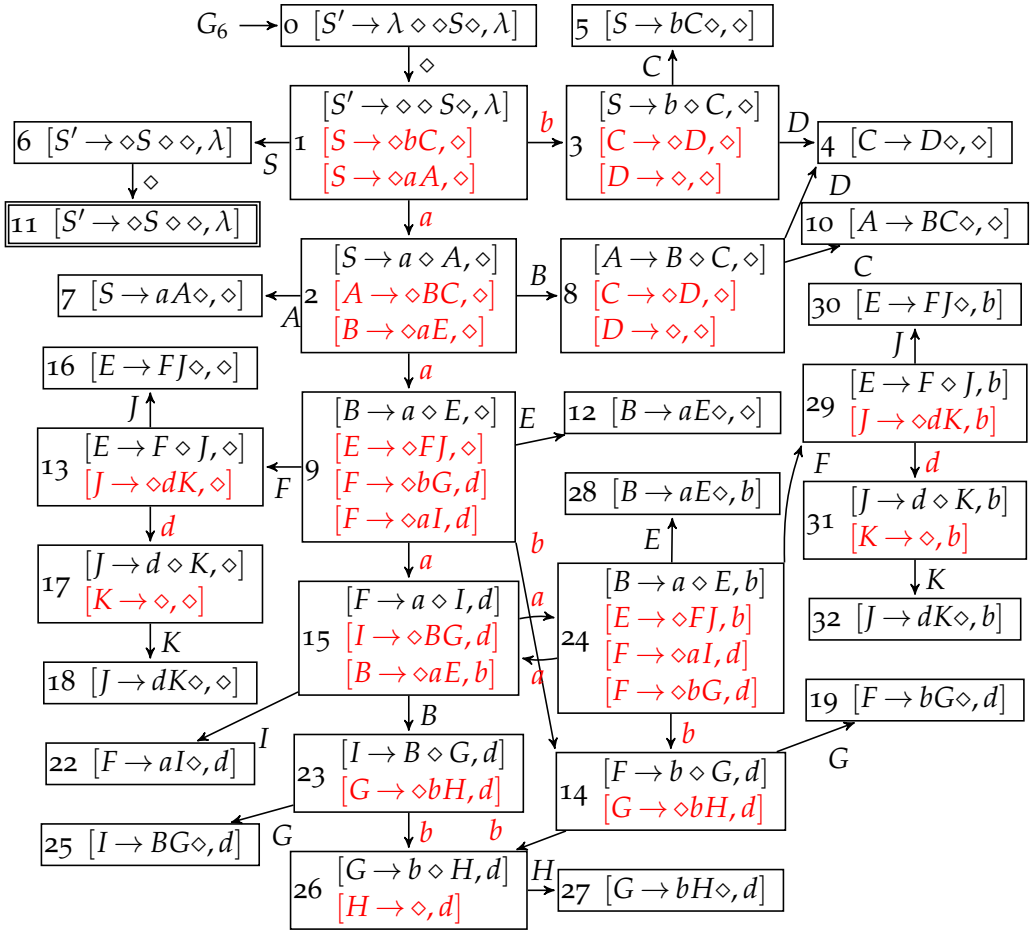



Figure 4.5.: The LR(1)-machine $G_6$. Edges generating terminals (relevant for shift-rules) and items with marker $\diamond$ at the beginning of the right hand side (relevant for reduce rules) are printed in red.

### 4.3.5 *Step 5 & Step 6 & Step 7*

In these steps we are following, with some modifications, the constructions in [16].

In step 5 we are constructing the $\diamond$-augmented version $G_5$ of $G_4$: A new initial nonterminal $S'$ and the production $S' \to \diamond S \diamond$ are added where $S$ is the old nonterminal. This modification allows for a simpler construction procedure of the LR(1)-machine and the LR(1)-parser in steps 6 and 7.

In step 6 we are constructing the LR(1)-machine $G_6$ (depicted in Figure 4.5) for the LR(1)-grammar $G_5$. The output alphabet of the DFA $G_6$ is the union of the output alphabet and the nonterminals of $G_5$. The steps of the parser (between two states $p, p'$ of $G_6$) will depend on the elements of $p$: these elements are called items which are formally four-tuples containing a production with a marker splitting the right hand side of the production and a lookahead symbol. The DFA $G_6$ has two kinds of edges: the edges labeled with an output symbol $\alpha$ represent the action where the parser generates $\alpha$, the edges labeled with nonterminals are required for the actions where the parser concludes (based on its stack and the lookahead of the items) that it has generated a word derivable

| Reduce Rules | | | Shift Rules |
|---|---|---|---|
| $1{\cdot}3{\cdot}5\|\diamond\rightarrow\ 1{\cdot}6\|\diamond$ | $1{\cdot}2{\cdot}7\|\diamond\rightarrow\ 1{\cdot}6\|\diamond$ | | $1\|a\rightarrow\ 1{\cdot}2\|$ |
| $2{\cdot}8{\cdot}10\|\diamond\rightarrow\ 2{\cdot}7\|\diamond$ | $2{\cdot}9{\cdot}12\|\diamond\rightarrow\ 2{\cdot}8\|\diamond$ | | $1\|b\rightarrow\ 1{\cdot}3\|$ |
| $3{\cdot}4\|\diamond\rightarrow\ 3{\cdot}5\|\diamond$ | $3\|\diamond\rightarrow\ 3{\cdot}4\|\diamond$ | | $2\|a\rightarrow\ 2{\cdot}9\|$ |
| $8{\cdot}4\|\diamond\rightarrow\ 8{\cdot}10\|\diamond$ | $8\|\diamond\rightarrow\ 8{\cdot}4\|\diamond$ | | $9\|a\rightarrow\ 9{\cdot}15\|$ |
| $9{\cdot}13{\cdot}16\|\diamond\rightarrow\ 9{\cdot}12\|\diamond$ | $13{\cdot}17{\cdot}18\|\diamond\rightarrow 13{\cdot}16\|\diamond$ | | $9\|b\rightarrow\ 9{\cdot}14\|$ |
| $17\|\diamond\rightarrow 17{\cdot}18\|\diamond$ | $9{\cdot}14{\cdot}19\|d\rightarrow\ 9{\cdot}13\|d$ | | $13\|d\rightarrow 13{\cdot}17\|$ |
| $9{\cdot}15{\cdot}22\|d\rightarrow\ 9{\cdot}13\|d$ | $14{\cdot}26{\cdot}27\|d\rightarrow 14{\cdot}19\|d$ | | $14\|b\rightarrow 14{\cdot}26\|$ |
| $15{\cdot}23{\cdot}25\|d\rightarrow 15{\cdot}22\|d$ | $15{\cdot}24{\cdot}28\|b\rightarrow 15{\cdot}23\|b$ | | $15\|a\rightarrow 15{\cdot}24\|$ |
| $23{\cdot}26{\cdot}27\|d\rightarrow 23{\cdot}25\|d$ | | | $23\|b\rightarrow 23{\cdot}26\|$ |
| $24{\cdot}29{\cdot}30\|b\rightarrow 24{\cdot}28\|b$ | $24{\cdot}15{\cdot}22\|d\rightarrow 24{\cdot}29\|d$ | | $24\|a\rightarrow 24{\cdot}15\|$ |
| $24{\cdot}14{\cdot}19\|d\rightarrow 24{\cdot}29\|d$ | $26\|d\rightarrow 26{\cdot}27\|d$ | | $24\|b\rightarrow 24{\cdot}14\|$ |
| $29{\cdot}31{\cdot}32\|b\rightarrow 29{\cdot}30\|b$ | $31\|b\rightarrow 31{\cdot}32\|b$ | | $29\|d\rightarrow 29{\cdot}31\|$ |

Figure 4.6.: The rules of the LR(1)-parser $G_7$ with initial state 1 and marking set $\{6\}$.

by a nonterminal.

Every edge $(p, \kappa, p')$ in $G_6$ satisfies that $p'$ is the least set satisfying the following conditions: $p'$ contains all items of $p$ where the marker $\diamond$ has been shifted over $\kappa$. Furthermore, if an item of the form $[A \rightarrow v \diamond B{\cdot}v', \sigma]$ is obtained, then the so-called "first"-symbols $\sigma'$ are determined[11] for which there is a $w$ satisfying $v'{\cdot}\sigma \vdash^*_{G_5} w$ with $\sigma' = 1{:}w$ and for all such (possibly empty) $\sigma'$ and all productions of the form $B \rightarrow v''$, the item $[B \rightarrow \diamond v'', \sigma']$ is contained in $p'$[12].

For example (in Figure 4.5), the $a$-successor of state 9 is state 15: $[F \rightarrow a \diamond I, d] \in 15$ is the result of the shifting of the $\diamond$ over the $a$ in the item $[F \rightarrow \diamond aI, d] \in 9$; $[I \rightarrow \diamond BG, d] \in 15$ because $[F \rightarrow a \diamond I, d] \in 15$ and $d$ is (trivially) derivable to $d$; $[B \rightarrow \diamond aE, b] \in 15$ because $[I \rightarrow \diamond BG, d] \in 15$ and $Gd$ is derivable to $bd$.

In step 7 we are constructing the LR(1)-parser $G_7$ (depicted in Figure 4.6) for $G_5$ and $G_6$. The parser consists of shift rules (generating a symbol and changing the stack and state) and reduce rules (which only modify the stack and state). The shift rules are obtained from the LR(1)-machine by selecting the edges in $G_6$ which are labeled with an output symbol: an edge $(p, \alpha, p')$ would result in the shift rule $p|\alpha \rightarrow p{\cdot}p'|\lambda$ (e.g., the edge $(1, a, 2)$ results in the rule $1|a \rightarrow 1{\cdot}2|\lambda$). The reduce rules are constructed for every item of the form $[A \rightarrow \diamond v, \alpha] \in p$ (i.e., the marker $\diamond$ is at the beginning of the right hand side): let $\tilde{p}$ (by construction $\tilde{p}$ is also a word over the stack alphabet of $G_7$) be the sequence of states visited by generating $v$ starting in $p$ in $G_6$ and let $p'$ be the state reached by generating $A$ in $p$ in $G_6$. Then the reduce rule $p{\cdot}\tilde{p}|\lambda \rightarrow p{\cdot}p'|\lambda$ is added to the parser (e.g., the item $[J \rightarrow \diamond dK, b] \in 29$ results in the rule $29{\cdot}31{\cdot}32|b \rightarrow 29{\cdot}30|b$).

**Remark 6**

*According to [16], the parser $G_7$ is a* correct prefix parser. *However, that is a too weak assertion: their definition of the unmarked language considers a symbol the parser has fixed but not generated not to be part of the generated unmarked word. Since the mode of operation we are interested*

[11] While in [16] no effective algorithm is presented for this operation we have been able to verify such a construction.

[12] The state with no items has been removed from the visualization in Figure 4.5.

| Reduce Rules | | Shift Rules |
|---|---|---|
| $9{\cdot}14{\cdot}19\vert d\rightarrow\ 9{\cdot}13\vert d$ | $24{\cdot}14{\cdot}19\vert d\rightarrow 24{\cdot}29\vert d$ | $1\vert a\rightarrow\ 1{\cdot}2\vert$ |
| $9{\cdot}15{\cdot}22\vert d\rightarrow\ 9{\cdot}13\vert d$ | $24{\cdot}15{\cdot}22\vert d\rightarrow 24{\cdot}29\vert d$ | $1\vert b\rightarrow\ 1{\cdot}3\vert$ |
| $15{\cdot}23{\cdot}25\vert d\rightarrow 15{\cdot}22\vert d$ | $26\vert d\rightarrow 26{\cdot}27\vert d$ | $2\vert a\rightarrow\ 2{\cdot}9\vert$ |
| $14{\cdot}26{\cdot}27\vert d\rightarrow 14{\cdot}19\vert d$ | $23{\cdot}26{\cdot}27\vert d\rightarrow 23{\cdot}25\vert d$ | $9\vert a\rightarrow\ 9{\cdot}15\vert$ |
| $15{\cdot}24{\cdot}28\vert b\rightarrow 15{\cdot}23\vert b$ | $24{\cdot}29{\cdot}30\vert b\rightarrow 24{\cdot}28\vert b$ | $9\vert b\rightarrow\ 9{\cdot}14\vert$ |
| $31\vert b\rightarrow 31{\cdot}32\vert b$ | $29{\cdot}31{\cdot}32\vert b\rightarrow 29{\cdot}30\vert b$ | $13\vert d\rightarrow 13{\cdot}17\vert$ |
| | | $14\vert b\rightarrow 14{\cdot}26\vert$ |
| | | $15\vert a\rightarrow 15{\cdot}24\vert$ |
| | | $23\vert b\rightarrow 23{\cdot}26\vert$ |
| | | $24\vert a\rightarrow 24{\cdot}15\vert$ |
| | | $24\vert b\rightarrow 24{\cdot}14\vert$ |
| | | $29\vert d\rightarrow 29{\cdot}31\vert$ |

Figure 4.7.: The rules of the LR(1)-parser $G_8$ with initial state 1 and marking set $\{3, 17\}$ (the nonterminals $\{4, 5, 7, 8, 10, 12, 16, 18\}$ are no longer reachable)

*(control of (embedded) discrete event systems), we had to find new proofs to verify that our stronger condition is also satisfied by the generated parser $G_7$.*

### *4.3.6 Step 8*

Since DPDA are not capable of terminating the generation by fixing an end-of-output marker, we are modifying the parser $G_7$ by removing all rules involving the end-of-output marker $\diamond$ and by changing the set of marking states such that $G_8$ (depicted in Figure 4.7) marks in $(s{\cdot}p, w, f)$ iff some edge $s'{\cdot}p\vert\diamond\rightarrow s''\vert\diamond$ has been removed. While it is not mentioned in [16], we discovered that this drastic removal of rules preserves the (un)marked language because the parser reaches a configuration in which such an edge is enabled if and only if the stack can be entirely reduced by subsequently executed reduce rules. This optimization also speeds up the parsing process using the presented construction in any other context (e.g., parsing of programming languages for which it has originally been designed).

| Reduce Rules | |
|---|---|
| $9{\cdot}14{\cdot}(19,\lambda)\vert d\rightarrow\ 9{\cdot}(13,d)\vert\lambda$ | $9{\cdot}14{\cdot}(19,d)\vert\lambda\rightarrow\ 9{\cdot}(13,d)\vert\lambda$ |
| $24{\cdot}14{\cdot}(19,\lambda)\vert d\rightarrow 24{\cdot}(29,d)\vert\lambda$ | $24{\cdot}14{\cdot}(19,d)\vert\lambda\rightarrow 24{\cdot}(29,d)\vert\lambda$ |
| $9{\cdot}15{\cdot}(22,\lambda)\vert d\rightarrow\ 9{\cdot}(13,d)\vert\lambda$ | $9{\cdot}15{\cdot}(22,d)\vert\lambda\rightarrow\ 9{\cdot}(13,d)\vert\lambda$ |
| $24{\cdot}15{\cdot}(22,\lambda)\vert d\rightarrow 24{\cdot}(29,d)\vert\lambda$ | $24{\cdot}15{\cdot}(22,d)\vert\lambda\rightarrow 24{\cdot}(29,d)\vert\lambda$ |
| $15{\cdot}23{\cdot}(25,\lambda)\vert d\rightarrow 15{\cdot}(22,d)\vert\lambda$ | $15{\cdot}23{\cdot}(25,d)\vert\lambda\rightarrow 15{\cdot}(22,d)\vert\lambda$ |
| $(26,\lambda)\vert d\rightarrow 26{\cdot}(27,d)\vert\lambda$ | $(26,d)\vert\lambda\rightarrow 26{\cdot}(27,d)\vert\lambda$ |
| $14{\cdot}26{\cdot}(27,\lambda)\vert d\rightarrow 14{\cdot}(19,d)\vert\lambda$ | $14{\cdot}26{\cdot}(27,d)\vert\lambda\rightarrow 14{\cdot}(19,d)\vert\lambda$ |
| $23{\cdot}26{\cdot}(27,\lambda)\vert d\rightarrow 23{\cdot}(25,d)\vert\lambda$ | $23{\cdot}26{\cdot}(27,d)\vert\lambda\rightarrow 23{\cdot}(25,d)\vert\lambda$ |
| $15{\cdot}24{\cdot}(28,\lambda)\vert b\rightarrow 15{\cdot}(23,b)\vert\lambda$ | $15{\cdot}24{\cdot}(28,b)\vert\lambda\rightarrow 15{\cdot}(23,b)\vert\lambda$ |
| $24{\cdot}29{\cdot}(30,\lambda)\vert b\rightarrow 24{\cdot}(28,b)\vert\lambda$ | $24{\cdot}29{\cdot}(30,b)\vert\lambda\rightarrow 24{\cdot}(28,b)\vert\lambda$ |
| $(31,\lambda)\vert b\rightarrow 31{\cdot}(32,b)\vert\lambda$ | $(31,b)\vert\lambda\rightarrow 31{\cdot}(32,b)\vert\lambda$ |
| $29{\cdot}31{\cdot}(32,\lambda)\vert b\rightarrow 29{\cdot}(30,b)\vert\lambda$ | $29{\cdot}31{\cdot}(32,b)\vert\lambda\rightarrow 29{\cdot}(30,b)\vert\lambda$ |

| Shift Rules | |
|---|---|
| $(1,\lambda)\vert a\rightarrow\ \ 1{\cdot}(2,\lambda)\vert\lambda$ | $(1,a)\vert\lambda\rightarrow\ \ 1{\cdot}(2,\lambda)\vert\lambda$ |
| $(1,\lambda)\vert b\rightarrow\ \ 1{\cdot}(3,\lambda)\vert\lambda$ | $(1,b)\vert\lambda\rightarrow\ \ 1{\cdot}(3,\lambda)\vert\lambda$ |
| $(2,\lambda)\vert a\rightarrow\ \ 2{\cdot}(9,\lambda)\vert\lambda$ | $(2,a)\vert\lambda\rightarrow\ \ 2{\cdot}(9,\lambda)\vert\lambda$ |
| $(9,\lambda)\vert a\rightarrow\ 9{\cdot}(15,\lambda)\vert\lambda$ | $(9,a)\vert\lambda\rightarrow\ 9{\cdot}(15,\lambda)\vert\lambda$ |
| $(9,\lambda)\vert b\rightarrow\ 9{\cdot}(14,\lambda)\vert\lambda$ | $(9,b)\vert\lambda\rightarrow\ 9{\cdot}(14,\lambda)\vert\lambda$ |
| $(13,\lambda)\vert d\rightarrow 13{\cdot}(17,\lambda)\vert\lambda$ | $(13,d)\vert\lambda\rightarrow 13{\cdot}(17,\lambda)\vert\lambda$ |
| $(14,\lambda)\vert b\rightarrow 14{\cdot}(26,\lambda)\vert\lambda$ | $(14,b)\vert\lambda\rightarrow 14{\cdot}(26,\lambda)\vert\lambda$ |
| $(15,\lambda)\vert a\rightarrow 15{\cdot}(24,\lambda)\vert\lambda$ | $(15,a)\vert\lambda\rightarrow 15{\cdot}(24,\lambda)\vert\lambda$ |
| $(23,\lambda)\vert b\rightarrow 23{\cdot}(26,\lambda)\vert\lambda$ | $(23,b)\vert\lambda\rightarrow 23{\cdot}(26,\lambda)\vert\lambda$ |
| $(24,\lambda)\vert a\rightarrow 24{\cdot}(15,\lambda)\vert\lambda$ | $(24,a)\vert\lambda\rightarrow 24{\cdot}(15,\lambda)\vert\lambda$ |
| $(24,\lambda)\vert b\rightarrow 24{\cdot}(14,\lambda)\vert\lambda$ | $(24,b)\vert\lambda\rightarrow 24{\cdot}(14,\lambda)\vert\lambda$ |
| $(29,\lambda)\vert d\rightarrow 29{\cdot}(31,\lambda)\vert\lambda$ | $(29,d)\vert\lambda\rightarrow 29{\cdot}(31,\lambda)\vert\lambda$ |

Figure 4.8.: The rules of the LR(1)-parser $G_9$ with initial state 1 and marking set $\{(3,\lambda),(17,\lambda)\}$.

### 4.3.7 Step 9

Since DPDA are not capable of fixing output symbols without generating them, we add the fixed output component of a configuration into the state of the configuration. For every shift rule of the form $p|\alpha \to p \cdot p'|\lambda$ the rules $(p,\lambda)|\alpha \to p \cdot (p',\lambda)|\lambda$ and $(p,\alpha)|\lambda \to p \cdot (p',\lambda)|\lambda$ are used. For every reduce rule of the form $s \cdot p|\alpha \to s' \cdot p'|\alpha$ the rules $s \cdot (p,\lambda)|\alpha \to s' \cdot (p',\alpha)|\lambda$ and $s \cdot (p,\alpha)|\lambda \to s' \cdot (p',\alpha)|\lambda$ are used. The resulting parser $G_9$ is depicted in Figure 4.8.

*It is then possible to verify, that all reachable configurations of the resulting parser $G_9$ have an empty fixed output component. We call the parser $G_9$ essentially* EDPDA *because it uses none of the extra capabilities of the parser formalism.*

### 4.3.8 Step 10

The essentially EDPDA parser $G_9$ can be translated into the EDPDA $G_{10}$ (depicted in Figure 4.9) by using for every rule of the form $s \cdot p|\sigma \to s' \cdot p'|\lambda$ the edge $(p, \sigma, s^{-1}, s'^{-1}, p')$. Marking and initial states of $G_{10}$ are taken from $G_9$.

### 4.3.9 Step 11

Since DPDA are not capable of popping strictly *more* than one symbol from the stack, we split such edges into multiple edges to obtain the DPDA $G_{11}$. To preserve determinism, the splitting of edges with the same source entails the merging of partially identical edges until the recursive split identifies their distinctness. For example, the edges $(p, \sigma, s \cdot s', s_1, p_1)$ and $(p, \sigma, s \cdot s'', s_2, p_2)$ share a common prefix $s$ on the popping component.

Since DPDA are not capable of popping strictly *less* than one symbol from the stack, we modify the automaton by replacing any edge $(p, \sigma, \lambda, s, p')$ with $(p, \sigma, \gamma, s \cdot \gamma, p)$ for any $\gamma$ of the stack alphabet of $G_{10}$. For soundness, recall that the stack-bottom-marker can never be removed from the stack.

### 4.3.10 Step 12

Finally, accessibility of states and edges can be enforced by reusing the already presented steps 1–4. For a DPDA we are executing steps 1–4. From the productions obtained by step 4 we can determine by executing the steps 1–3 backwards (which are by our construction injective in the sense that for each constructed production/edge $x$ a unique edge $e$ can be determined for which $x$ has been constructed). Using this backwards computation, we are able to determine the accessible edges of a DPDA. The accessible states are the sources and edges of any of the accessible edges. The inaccessible states and edges are then removed to obtain the DPDA $G_{12}$ from Figure 4.10.

We are not aware of comparable constructions ensuring accessibility of DPDA, however, using the decidability of emptiness from [10] it is possible to test a single (and by that every) edge for accessibility; this approach has been used in [7]. Our approach is superior as we are executing a single test on all edges simultaneously.

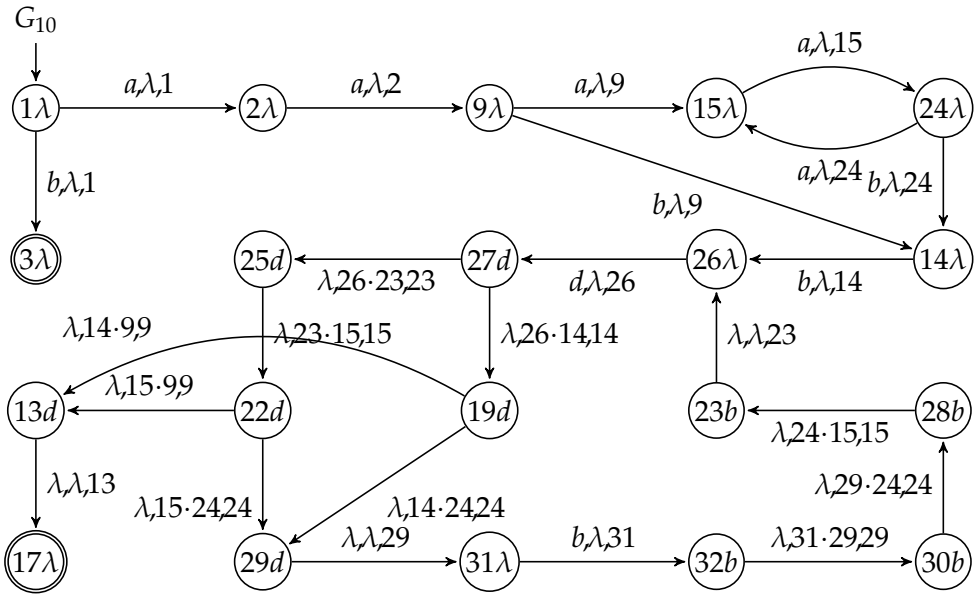



Figure 4.9.: The resulting EDPDA $G_{10}$ where obviously unreachable states have been removed.

### 4.3.11 *Verification*

The soundness of the presented algorithm (w.r.t. the problem Definition 28) has been verified in the interactive theorem prover Isabelle/HOL [13] apart from the following steps for which only pen-and-paper proofs exist yet and which are to be completed in Isabelle/HOL in the near future:

(i) the CFG obtained in step 4 is an LR(1) grammar (satisfied according to [11]),
(ii) the Parser obtained in step 7 is deterministic if $G_5$ is an LR(1) grammar (satisfied according to [16]),
(iii) step 11, and
(iv) step 12.

From these tasks however, only the first appears to be complicated.

### 4.3.12 *Testing*

The presented algorithm has been implemented in Java for rapid prototyping and in C++ as a plugin to the libFAUDES tool [12]. The implementations have been used successfully for many examples including the running example of this chapter.

### 4.3.13 *Optimizations*

The algorithm can be optimized in different ways.

(i) The runtime of the algorithm depends primarily on the steps 3 and 4 because $G_3$ would have an enormous amount of productions. We can greatly restrict the set of productions to be generated by exploiting the structure of the input DPDA using the reachability overapproximation presented on page 46.

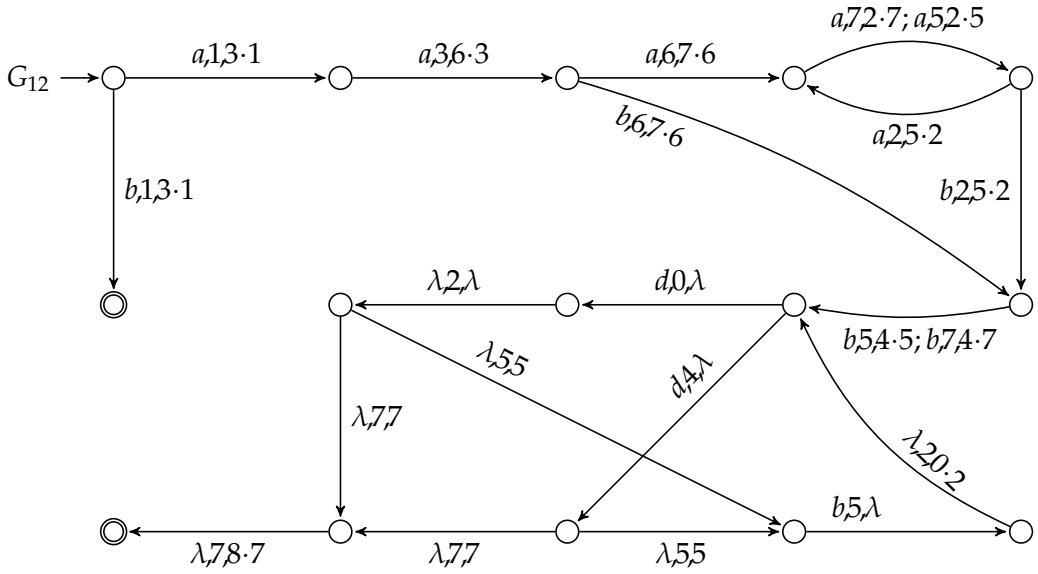



Figure 4.10.: The resulting DPDA $G_{12}$.

(ii) Furthermore, steps 3 and 4 can be merged such that only productions are generated which are coaccessible. This alternative trades runtime for space-requirements (the size of $G_4$ is usually not much greater than $G_1$ but the runtime is increased by the length of the longest derivation necessary in $G_2$ to reach all states).

(iii) Another optimization merges adjacent edges in EDPDA which are intermediate results. This optimization decreases the runtime of the subsequently executed operations.

The formal definition and verification in Isabelle/HOL of such intermediate operations is left for future work.

## 4.4 Conclusion

The algorithm presented in this chapter optimizes the behavior of a DPDA whilst preserving its marked language by first translating the DPDA into another model (LR(1) grammars) in which the desired properties can be enforced using simple constructions and by translating the obtained solution back into DPDA while preserving the desired properties.

The algorithm guarantees accessibility (every state and every edge is required for some marking derivation), lifelockfreeness (there is no initial derivation executing infinitely many steps without generating an output symbol), deadlockfreeness (non-extendable initial derivations are ending in marking states), and finally the operational blockfreeness (every initial derivation can be extended into a marking derivation).

The operational blockfreeness is sufficient to conclude that the unmarked language is the prefix closure of the marked language of the resulting DPDA.

The algorithm does not minimize the size of the automaton, in fact, the size of the resulting DPDA is usually increased and is growing according to [3] in some cases exponentially.

The algorithm presented here is a crucial part of the presented solution of the supervisory control problem for DFA plants and DPDA specifications which is reduced (in Chapter 2) to the effective implementability of ensuring blockfreeness (solved in this chapter) and ensuring controllability (solved in Chapter 3).

## 4.5 Future Work

**Petri nets** Since the problem of establishing blockfreeness is unsolvable for standard Petri net classes [5, 6], we intend to determine Petri net classes $\mathcal{P}$ that can be translated (preserving the marked language) into a DPDA $G$ such that the DPDA generated by our algorithm $G'$ can be translated back into a Petri net from $\mathcal{P}$ to solve the problem for such a Petri net class.

**Visibly Pushdown Tree Automata (VPTA)** VPTA introduced in [2] are the greatest known subclass of DPDA which are closed under intersection. For the context of the Supervisory Control Theory we intend to determine an algorithm which solves the problem from Definition 28 for VPTA because

(i) plant and controller can then be generated by VPTA, while this decreases the expressiveness for the controller language it also increases the expressiveness for the plant language, and

(ii) the closed loop is again a VPTA, which allows for the iterative restriction of a plant language by horizontal composition of controllers.

The algorithm presented here may be reusable: the output of the algorithm, when executed on a VPTA, may be (convertible) into a VPTA. Therefore, when using VPTA for plants, specifications, and controllers, the supervisory controller synthesis can be extended to yet another domain.

**Nondeterminism** For the context of the Supervisory Control Theory there is no reason to restrict oneself to deterministic controllers. However, for these systems the desired property of operational blockfreeness is not guaranteed for language blockfree controllers. Therefore, when extending the domain of the algorithm to PDA the proofs will become more complex as the preservation of marked and unmarked language is no longer sufficient for the preservation of the operational blockfreeness as discussed in Chapter 2.

# Bibliography


[1] Alfred V. Aho and Jeffrey D. Ullman. *The theory of parsing, translation, and compiling*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1972.

[2] Jacques Chabin and Pierre Réty. Visibly pushdown languages and term rewriting. In Boris Konev and Frank Wolter, editors, *FroCoS*, volume 4720 of *Lecture Notes in Computer Science*, pages 252–266. Springer, 2007.

[3] Matthew M. Geller, Harry B. Hunt III, Thomas G. Szymanski, and Jeffrey D. Ullman. Economy of descriptions by parsers, dpda's, and pda's. In *FOCS*, pages 122–127. IEEE Computer Society, 1975.

[4] Seymour Ginsburg and Sheila A. Greibach. Deterministic context free languages. *Information and Control*, 9(6):620–648, 1966.

[5] A. Giua and F. DiCesare. Blocking and controllability of petri nets in supervisory control. *IEEE Transactions on Automatic Control*, 39(4):818–823, 1994.

[6] A. Giua and F. DiCesare. Decidability and closure properties of weak petri net languages in supervisory control. *IEEE Transactions on Automatic Control*, 40(5):906–910, 1995.

[7] C. Griffin. A note on deciding controllability in pushdown systems. *IEEE Transactions on Automatic Control*, 51(2):334 – 337, feb. 2006.

[8] C. Griffin. A note on the properties of the supremal controllable sublanguage in pushdown systems. *IEEE Transactions on Automatic Control*, 53(3):826 –829, apr. 2008.

[9] Christopher Griffin. *Decidability and optimality in pushdown control systems: A new approach to discrete event control*. PhD thesis, The Pensylvania State University, 2007.

[10] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, languages and computation*. Addison-Wesley Publishing company, 1979.

[11] Donald E. Knuth. On the translation of languages from left to rigth. *Information and Control*, 8(6):607–639, 1965.

[12] libFAUDES. Software library for discrete event systems., 2006-2013.

[13] Larry Paulson, Tobias Nipkow, and Makarius Wenzel. Isabelle/HOL, 2011.

[14] P.J. Ramadge and W.M. Wonham. Supervisory control of a class of discrete event processes. In A. Bensoussan and J.L. Lions, editors, *Analysis and Optimization of Systems*, volume 63 of *Lecture Notes in Control and Information Sciences*, pages 475–498. Springer Berlin Heidelberg, 1984.

[15] Géraud Sénizergues. The equivalence problem for deterministic pushdown automata is decidable. In Pierpaolo Degano, Roberto Gorrieri, and Alberto Marchetti-Spaccamela, editors, *ICALP*, volume 1256 of *Lecture Notes in Computer Science*, pages 671–681. Springer, 1997.

[16] Seppo Sippu and Eljas Soisalon-Soininen. *Parsing Theory*, volume II: LR($k$) and LL($k$) Parsing of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1990.

[17] W. M. Wonham and P. J. Ramadge. On the supremal controllable sublanguage of a given language. In *SIAM Journal on Control and Optimization*, volume 25, pages 637–659, 1987.

# A. Appendix

## A.1 Counterexample

In this section, the algorithm presented in [8, p.827] and [9, p.64] is applied to an example. It will be shown that this represents a counterexample to [8, Theorem 3.5] and [9, Theorem 5.2.5] since the final DPDA does not realize the supremal controllable sublanguage of the given prefix closed deterministic context free specification language (which is required to be a subset of the given prefix closed regular plant language). Therefore we claim that the problem of automatically calculating a supremal controllable sublanguage of a DCFL was not solved by [9, 8].
The algorithm is initialized with a DFA $G$ and a DPDA $M$ realizing the plant and the specification, respectively, s.t. $L_m(G) = L_{um}(G)$, $L_m(M) = L_{um}(M)$ and $L_m(M) \subseteq L_m(G)$. Observe that the DFA $G$ and the DPDA $M$ depicted in Figure A.1 satisfy these requirements since their languages are given by

$$L_m(G) = L_{um}(G) = \{a^n, a^n b, a^n bu | n \in \mathbf{N}\} \quad \text{and}$$
$$L_m(M) = L_{um}(M) = \left\{a^n, a^m b, a^k bu \,\middle|\, n, m, k \in \mathbf{N}, m>0, k>1\right\}.$$


Figure A.1.: DFA $G$ (left) and DPDA $M$ (right) realizing plant and specification, respectively.

Using these automata, the construction follows seven steps.

1. Construct $M'$, depicted in Figure A.2, by making $M$ scan its entire input, using the algorithm by [10, Lem.10.3.].

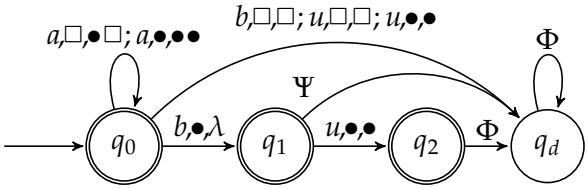



Figure A.2.: DPDA $M'$, with $\Psi = \Phi \setminus u,\bullet,\bullet$ and $\Phi = \{a,\square,\square; a,\bullet,\bullet; b,\square,\square; b,\bullet,\bullet; u,\square,\square; u,\bullet,\bullet\}$.

2. Construct a DPDA $M''$ that accepts the complement of $L_m(M)$ using the algorithm by [10], Thm.10.1. Here $M''$ is identical to $M'$ in Figure A.2 but with exchanged non-marking and marking states, i.e., $F'' = \{q_d\}$.
3. Construct a DPDA $M'''$ that accepts $L_m^c(M) \cap L_m(G)$, i.e., calculate the cross product of $G$ and $M''$ using the algorithm by [10], Thm.6.5.
4. Construct $M_1$, depicted in Figure A.3, as the accessible part of $M'''$, using the algorithm by [7], Thm.4.1.

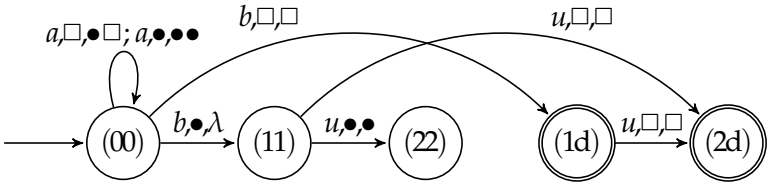



Figure A.3.: DPDA $M_1$, with $(ij) \triangleq (p_i, q_j)$

5. Construct a predicting machine to observe so called $\mu$-reverse paths using the algorithm by [10], p.240. Here, the construction simply defines an additional stack symbol $\mu_\gamma$, $\forall \gamma \in \Gamma$ s.t.

$$\mu_\gamma := \left\{ q \in Q_1 \backslash F_1 \;\middle|\; \begin{array}{l} \exists\, q' \in F_1, v \in \Sigma_{uc}^* \,. \\ \hookrightarrow (\bot, (q, w, \gamma \cdot s)) \vdash_{M_1}^* (\bot, (q', w \cdot v, s')) \end{array} \right\}$$

which denotes the set of unmarked states in $M_1$ from which a derivation starting with stack-top $\gamma$, generating a sequence of uncontrollable symbols $v \in \Sigma_{uc}^*$ and reaching a marking state $q'$ (i.e., a so called $\mu$-reverse path), exists. For $M_1$, depicted in Figure A.3, this gives $\mu_\square = \{(p_1, q_1)\}$ and $\mu_\bullet = \emptyset$. The predicting machine $M_1^\mu$, depicted in Figure A.4, is then identical to $M_1$ but uses pairs $[\gamma, \mu_\gamma]$ as stack.

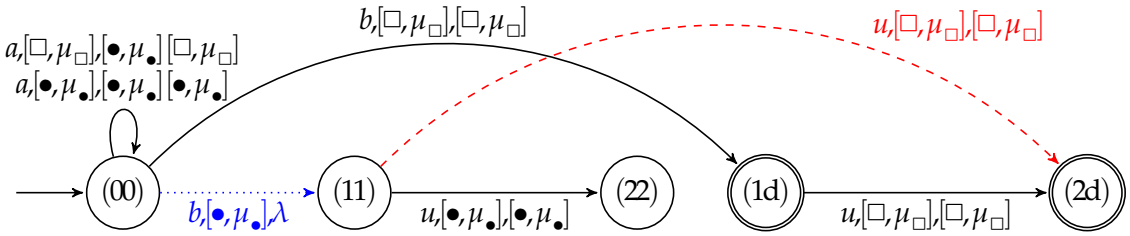



Figure A.4.: DPDA $M_1^\mu$ with $\mu_\square = \{q_2\}$ and $\mu_\bullet = \emptyset$. The set of $\mu$-reverse paths is depicted in red (dashed) while the set of edges in $\delta_{cp}$ is depicted in blue (dotted).

6. Construct $M_2$, depicted in Figure A.5, by deleting all transitions

$$\delta_{cp} \triangleq \{(q, \sigma, \gamma, \gamma' \cdot s, q') \in \delta_M \mid \begin{pmatrix} \wedge (q, \sigma, \gamma, [\gamma', \mu_{\gamma'}] \cdot s, q') \in \delta_{M_1^\mu} \\ \wedge \sigma \in \Sigma_c \wedge q' \in \mu_{\gamma'} \end{pmatrix}\}$$

in $M$ which produce a stack-top in $q'$ which enables a $\mu$-reverse path starting in $q'$. For $M$ and $M_1^\mu$, depicted in Figure A.1 and A.4, respectively, observe that $e =$

$((p_0, q_0), b, [\bullet, \mu_\bullet], \lambda, (p_1, q_1)) \in \delta_{M_1^\mu}$ is the only ingoing transition to $(p_1, q_1)$ (where the only $\mu$-reverse path starts for stack-top $\square$, since $\mu_\square = \{(p_1, q_1)\}$) and, since $M_1$ is trim, eventually leads to the stack-top $\square$ in $(p_1, q_1)$. Using the corresponding transition to $e$ in $M$, this gives $\delta_{cp} = \{(q_0, b, \bullet, \lambda, q_1)\}$. By deleting $\delta_{cp}$ in $M$, we obtain $M_2$, depicted in Figure A.5.

7. Construct $M_3$, depicted in Figure A.6, as the accessible part of $M_2$, using the algorithm by [7], Thm.4.1. If $\delta_{cp}$ in step 6 is empty, the algorithm terminates. Otherwise, the algorithm is restarted with $M = M_3$.

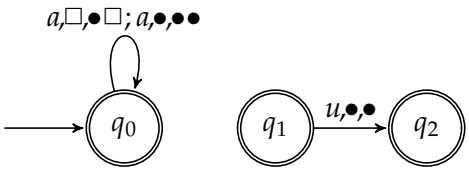



Figure A.5.: DPDA $M_2$

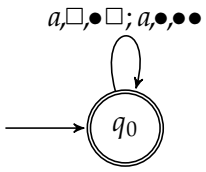



Figure A.6.: DPDA $M_3$

Obviously, $M_3$ does not have further controllability problems. Therefore, the algorithm would redo steps 1-6 and then return $M_3$.
Now observe that the specification language $\mathrm{L_m}(M)$ restricts the plant language $\mathrm{L_m}(G)$ such that $u$ cannot occur after exactly one $a$. This generates a controllability for the word $ab$ only. Using (3.4) in Section 3.3, the supremal controllable sublanguage of $\mathrm{L_m}(M)$ for this example is given by

$$\begin{aligned} L_{\mathrm{clm}} &= \{w \in \mathrm{L_m}(M) \mid \forall w' \sqsubseteq w \,.\, w' \neq ab\} \\ &= \mathrm{L_m}(M) \setminus \{ab\} \\ &= \left\{ a^n, a^m b, a^k bu \,\middle|\, n, m, k \in \mathbf{N}, m, k > 1 \right\} \end{aligned} \tag{A.1}$$

implying that $\mathrm{L_m}(M_3) = \{a^n \mid n \in \mathbf{N}\}$ is a strict subset of $L_{\mathrm{clm}}$ which is an obvious contradiction to [8, Thm.3.5]. Furthermore, $L_{\mathrm{clm}}$ in (A.1) cannot be realized using the state and transition structure of $M$ and only deleting existing transitions.
The automatic synthesis of a DPDA realizing $L_{\mathrm{clm}}$ for this example is provided as an example within our pushdown-plug-in for [12].

## A.2 Formal Definitions for the Constructions

We present the formal definitions of the constructions which constitute the synthesis algorithm as an output from Isabelle/HOL.
**theory** *FUNCTION-definitions*

**imports**
*I-epda-lemmas*
*INTER-CFGRM*
*I-kparser-lemmas*
*Fundamentals*
*LaTeXsugar*

**begin**

### *A.2.1 Split Read*

**datatype** $('q,'a,'b)$ *SDPDA1State* =
*old* $'q$
| *new* $('q,'a,'b)$*EDGE*

**definition** *FUNSR-E1* :: $('q,'a,'b)$*EDGE*
⇒ $(('q,'a,'b)$ *SDPDA1State*$,'a,'b)$*EDGE*
**where**
*FUNSR-E1 e* ≡
(|*edge-src*=*old* (*edge-src e*),
*edge-read*=*edge-read e*,
*edge-pop*=*edge-pop e*,
*edge-push*=*edge-pop e*,
*edge-trg*=*new e*|)

**definition** *FUNSR-E2* :: $('q,'a,'b)$*EDGE*
⇒ $(('q,'a,'b)$ *SDPDA1State*$,'a,'b)$*EDGE*
**where**
*FUNSR-E2 e* ≡
(|*edge-src*=*new e*,
*edge-read*=*None*,
*edge-pop*=*edge-pop e*,
*edge-push*=*edge-push e*,
*edge-trg*=*old* (*edge-trg e*)|)

**definition** *FUNSR-read* :: $('q,'a,'b)$*EDGE*
⇒ $(('q,'a,'b)$ *SDPDA1State*$,'a,'b)$*EDGE set*
**where**
*FUNSR-read e* ≡ {*FUNSR-E1 e*,*FUNSR-E2 e*}

**definition** *FUNSR-E* :: $('q,'a,'b)$*EDGE*
⇒ $(('q,'a,'b)$ *SDPDA1State*$,'a,'b)$*EDGE*
**where**
*FUNSR-E e* ≡
(|*edge-src*=*old* (*edge-src e*),
*edge-read*=*edge-read e*,

```
  edge-pop=edge-pop e,
  edge-push=edge-push e,
  edge-trg=old (edge-trg e)|)

definition FUNSR-else :: ('q,'a,'b)EDGE
  ⇒ (('q,'a,'b) SDPDA1State,'a,'b)EDGE set
  where
  FUNSR-else e ≡ {FUNSR-E e}

definition MixedReadEdge :: ('a,'b,'c)EDGE
  ⇒ bool
  where
  MixedReadEdge e ≡ edge-read e≠None ∧ (edge-pop e≠edge-push e)

definition FUNSR :: ('q,'a,'b)EPDA
  ⇒ (('q,'a,'b) SDPDA1State,'a,'b)EPDA
  where
  FUNSR G ≡
  (|epda-states=old ' (epda-states G) ∪ new ' (epda-delta G),
  epda-sigma=epda-sigma G,
  epda-gamma=epda-gamma G,
  epda-delta=⋃(λe. if (MixedReadEdge e) then FUNSR-read e
    else FUNSR-else e)'(epda-delta G),
  epda-initial=old (epda-initial G),
  epda-box=epda-box G,
  epda-final=old ' (epda-final G)|)
```

### A.2.2 Remove No Operation

```
definition FUNRNoOp-else :: ('q,'a,'b)EDGE
  ⇒ (('q,'a,'b) SDPDA1State,'a,'b)EDGE where
  FUNRNoOp-else e ≡
  (|edge-src=old (edge-src e),
  edge-read=edge-read e,
  edge-pop=edge-pop e,
  edge-push=edge-push e,
  edge-trg=old (edge-trg e)|)

definition FUNRNoOp-ON :: ('q,'a,'b)EDGE
  ⇒ 'b
  ⇒ (('q,'a,'b) SDPDA1State,'a,'b)EDGE
  where
  FUNRNoOp-ON e PB ≡
  (|edge-src=old (edge-src e),
  edge-read=None,
  edge-pop=edge-pop e,
  edge-push=[PB] @ (edge-pop e),
  edge-trg=new e|)

definition FUNRNoOp-NO :: ('q,'a,'b)EDGE
  ⇒ 'b
  ⇒ (('q,'a,'b) SDPDA1State,'a,'b)EDGE
  where
```

```
FUNRNoOp-NO e PB ≡
(|edge-src=new e,
edge-read=None,
edge-pop=[PB],
edge-push=[],
edge-trg=old(edge-trg e)|)

definition FUNRNoOp-NoOp :: ('q,'a,'b)EDGE
 ⇒ 'b
 ⇒ (('q,'a,'b) SDPDA1State,'a,'b)EDGE set
 where
FUNRNoOp-NoOp e PB ≡ {FUNRNoOp-ON e PB,FUNRNoOp-NO e PB}

definition FUNRNoOp :: ('q,'a,'b)EPDA
 ⇒ 'b
 ⇒ (('q,'a,'b) SDPDA1State,'a,'b)EPDA
 where
FUNRNoOp G PB ≡
(|epda-states=old ' (epda-states G) ∪ new ' (epda-delta G),
epda-sigma=epda-sigma G,
epda-gamma=epda-gamma G ∪ {PB},
epda-delta=⋃(λe. if NoOpEdge e then FUNRNoOp-NoOp e PB
 else {FUNRNoOp-else e})'(epda-delta G),
epda-initial=old (epda-initial G),
epda-box=epda-box G,
epda-final=old ' (epda-final G)|)
```

### A.2.3 Split Push Pop

```
definition FUNSPP-ON :: ('q,'a,'b)EDGE
 ⇒ (('q,'a,'b) SDPDA1State,'a,'b)EDGE
 where
FUNSPP-ON e ≡
(|edge-src=old (edge-src e),
edge-read=None,
edge-pop=edge-pop e,
edge-push=[],
edge-trg=new e|)

definition FUNSPP-NO :: ('q,'a,'b)EDGE
 ⇒ 'b set
 ⇒ (('q,'a,'b) SDPDA1State,'a,'b)EDGE set
 where
FUNSPP-NO e S ≡ {
(|edge-src=new e,
edge-read=None,
edge-pop=[X],
edge-push=edge-push e @ [X],
edge-trg=old(edge-trg e)|)| X.
X∈S}

definition FUNSPP-NoOp :: ('q,'a,'b)EDGE
 ⇒ 'b set
```

```
 ⇒ (('q,'a,'b) SDPDA1State,'a,'b)EDGE set
 where
 FUNSPP-NoOp e S ≡ {FUNSPP-ON e} ∪ FUNSPP-NO e S

definition PushPopEdge :: ('q,'a,'b)EDGE
 ⇒ 'b ⇒ bool
 where
 PushPopEdge e BOX ≡ (edge-read e = None ∧ (∃w b. edge-push e = w@[b] ∧ edge-pop e ≠ [b]))

definition FUNSPP-else :: ('q,'a,'b)EDGE
 ⇒ (('q,'a,'b) SDPDA1State,'a,'b)EDGE
 where
 FUNSPP-else e ≡
 (|edge-src=old (edge-src e),
 edge-read=edge-read e,
 edge-pop=edge-pop e,
 edge-push=edge-push e,
 edge-trg=old (edge-trg e)|)

definition FUNSPP :: ('q,'a,'b)EPDA
 ⇒ (('q,'a,'b) SDPDA1State,'a,'b)EPDA
 where
 FUNSPP G ≡
 (|epda-states=old ' (epda-states G) ∪ new ' (epda-delta G),
 epda-sigma=epda-sigma G,
 epda-gamma=epda-gamma G,
 epda-delta=⋃(λe. if PushPopEdge e (epda-box G) then FUNSPP-NoOp e (epda-gamma G)
  else {FUNSPP-else e})`(epda-delta G),
 epda-initial=old (epda-initial G),
 epda-box=epda-box G,
 epda-final=old ' (epda-final G)|)
```

### A.2.4 Remove Multiple Push

```
datatype ('q,'a,'b) SDPDA2State =
 old 'q
 | new ('q,'a,'b)EDGE nat

definition FUNRMP-else :: ('q,'a,'b)EDGE
 ⇒ (('q,'a,'b) SDPDA2State,'a,'b)EDGE
 where
 FUNRMP-else e ≡
 (|edge-src=old (edge-src e),
 edge-read=edge-read e,
 edge-pop=edge-pop e,
 edge-push=edge-push e,
 edge-trg=old (edge-trg e)|)

definition FUNRMP-state :: ('q,'a,'b)EDGE
 ⇒ nat
 ⇒ ('q,'a,'b) SDPDA2State option
 where
 FUNRMP-state e n ≡
```

```
 (if n=o then Some (old (edge-src e))
 else (if Suc n<length(edge-push e) then Some(new e n)
 else (if Suc n=length(edge-push e) then Some(old (edge-trg e))
 else None)))

definition FUNRMP-steps :: ('q,'a,'b)EDGE
 ⇒ (('q,'a,'b) SDPDA2State,'a,'b)EDGE set
 where
 FUNRMP-steps e ≡ ⋃(λi. {
 (|edge-src=the(FUNRMP-state e i),
 edge-read=None,
 edge-pop=[(rev(edge-push e))!i],
 edge-push=[(rev(edge-push e))!(Suc i)]@[(rev(edge-push e))!i],
 edge-trg=the(FUNRMP-state e (Suc i))|)
 }) ‘ {i. o≤i ∧ Suc i<length(edge-push e)}

definition FUNRMP :: ('q,'a,'b)EPDA
 ⇒ (('q,'a,'b) SDPDA2State,'a,'b)EPDA
 where
 FUNRMP G ≡
 (|epda-states=(old ‘ (epda-states G))
  ∪ ⋃((λe. (λi. the(FUNRMP-state e i))
  ‘ {i. o≤i ∧ Suc i≤length(edge-push e)})
  ‘ (epda-delta G)),
 epda-sigma=epda-sigma G,
 epda-gamma=epda-gamma G,
 epda-delta=⋃(λe. if MultiPushEdge e then FUNRMP-steps e
  else {FUNRMP-else e})‘(epda-delta G),
 epda-initial=old (epda-initial G),
 epda-box=epda-box G,
 epda-final=old ‘ (epda-final G)|)
```

### A.2.5 DPDA to SDPDA

```
definition FUN-2SDPDA :: ('q,'a,'b symbol)EPDA
  ⇒ (((((('q, 'a, 'b symbol) SDPDA1State, 'a, 'b symbol) SDPDA1State, 'a, 'b symbol) SDPDA2State, 'a, 'b symbol) EPDA
 where
 FUN-2SDPDA G ≡
 (let G2 = FUNSR G;
 G3 = FUNRNoOp G2 (FUNfreshSymbol (epda-gamma G2));
 G4 = FUNSPP G3;
 G5 = FUNRMP G4 in G5)
```

### A.2.6 No Double Acceptance

```
datatype 'q NDAState =
 old 'q
 | new 'q

definition FUNNDA-EAnn :: ('q,'a,'b)EDGE
 ⇒ ('q⇒'q NDAState)
 ⇒ ('q⇒'q NDAState)
```

```
 ⇒ ('q NDAState,'a,'b)EDGE
 where
 FUNNDA-EAnn e s t ≡
 (|edge-src=s(edge-src e),
 edge-read=edge-read e,
 edge-pop=edge-pop e,
 edge-push=edge-push e,
 edge-trg=t(edge-trg e)|)

definition FUNNDA-Edges :: ('q,'a,'b)EDGE
 ⇒ 'q set
 ⇒ ('q NDAState,'a,'b)EDGE set
 where
 FUNNDA-Edges e FS ≡
 if ReadEdge e then {FUNNDA-EAnn e old old,FUNNDA-EAnn e new old}
 else ({FUNNDA-EAnn e new new}∪(
   if edge-src e ∈ FS then {FUNNDA-EAnn e old new}
   else {FUNNDA-EAnn e old old}))

definition FUNNDA :: ('q,'a,'b)EPDA
 ⇒ ('q NDAState,'a,'b)EPDA
 where
 FUNNDA G ≡
 (|epda-states=old ' (epda-states G) ∪ new ' (epda-states G),
 epda-sigma=epda-sigma G,
 epda-gamma=epda-gamma G,
 epda-delta=⋃(λe. FUNNDA-Edges e (epda-final G))'(epda-delta G),
 epda-initial=old (epda-initial G),
 epda-box=epda-box G,
 epda-final=old ' (epda-final G)|)
```

### A.2.7 SDPDA to LR1

```
datatype ('q,'b) LR1State =
 l2 'q 'b
 | l3 'q 'b 'q

definition FUN2LR1-EAnnRead1 :: ('q,'a,'b)EDGE
 ⇒ 'q set
 ⇒ (('q,'b) LR1State,'a)PRODUCTION set
 where
 FUN2LR1-EAnnRead1 e Q ≡ {
 (|prod-lhs=l3 (edge-src e) ((edge-pop e)!0) qt,
 prod-rhs=[beB (the(edge-read e))]@[beA(l3 (edge-trg e) ((edge-pop e)!0) (qt))]|)
 | qt. qt ∈ Q}

definition FUN2LR1-EAnnPop1 :: ('q,'a,'b)EDGE
 ⇒ (('q,'b) LR1State,'a)PRODUCTION set
 where
 FUN2LR1-EAnnPop1 e ≡ {
 (|prod-lhs=l3 (edge-src e) ((edge-pop e)!0) (edge-trg e),
 prod-rhs=[]|)
 }
```

```
definition FUN2LR1-EAnnPush1 :: ('q,'a,'b)EDGE
 ⇒ 'q set
 ⇒ (('q,'b) LR1State,'a)PRODUCTION set
 where
 FUN2LR1-EAnnPush1 e Q ≡ {
 (|prod-lhs=l3 (edge-src e) ((edge-pop e)!o) (qt),
 prod-rhs=[beA(l3 (edge-trg e) ((edge-push e)!o) (qs))]@[beA(l3 (qs) ((edge-pop e)!o) (qt))]|)
 | qs qt. qs ∈ Q ∧ qt ∈ Q}

definition FUN2LR1-Edges1 :: ('q,'a,'b)EPDA
 ⇒ (('q,'b) LR1State,'a)PRODUCTION set
 where
 FUN2LR1-Edges1 G ≡
 ⋃(λe. case edge-read e of None ⇒ {} |
  Some A ⇒ FUN2LR1-EAnnRead1 e (epda-states G))`(epda-delta G)
 ∪ ⋃(λe. case edge-push e of a#y ⇒ {} | [] ⇒ FUN2LR1-EAnnPop1 e)`(epda-delta G)
 ∪ ⋃(λe. case edge-push e of [] ⇒ {} |
    a#y ⇒ (case edge-read e of Some A ⇒ {} |
     None ⇒ FUN2LR1-EAnnPush1 e (epda-states G)))`(epda-delta G)

definition FUN2LR1-EAnnRead2 :: ('q,'a,'b)EDGE
 ⇒ 'q set
 ⇒ (('q,'b) LR1State,'a)PRODUCTION set
 where
 FUN2LR1-EAnnRead2 e Q ≡ {
 (|prod-lhs=l2 (edge-src e) ((edge-pop e)!o),
 prod-rhs=[beB (the(edge-read e))]@[beA(l2 (edge-trg e) ((edge-pop e)!o))]|)}

definition FUN2LR1-EAnnFinal2 :: 'q set
 ⇒ 'b set
 ⇒ (('q,'b) LR1State,'a)PRODUCTION set
 where
 FUN2LR1-EAnnFinal2 F S ≡ {(|prod-lhs=l2 i A,prod-rhs=[]|)| i A. i ∈ F ∧ A ∈ S}

definition FUN2LR1-EAnnPush2 :: ('q,'a,'b)EDGE
 ⇒ 'q set
 ⇒ (('q,'b) LR1State,'a)PRODUCTION set
 where
 FUN2LR1-EAnnPush2 e Q ≡ {
 (|prod-lhs=l2 (edge-src e) ((edge-pop e)!o),
 prod-rhs=[beA(l2 (edge-trg e) ((edge-push e)!o))]|)
 }∪{
 (|prod-lhs=l2 (edge-src e) ((edge-pop e)!o),
 prod-rhs=[beA(l3 (edge-trg e) ((edge-push e)!o) (qs))]@[beA(l2 (qs) ((edge-pop e)!o))]|)
 | qs. qs ∈ Q}

definition FUN2LR1-Edges2 :: ('q,'a,'b)EPDA
 ⇒ (('q,'b) LR1State,'a)PRODUCTION set
 where
 FUN2LR1-Edges2 G ≡
 ⋃(λe. case edge-read e of None ⇒ {} |
  Some A ⇒ FUN2LR1-EAnnRead2 e (epda-states G))`(epda-delta G)
```

```
 ∪ (FUN2LR1-EAnnFinal2 (epda-final G) (epda-gamma G))
 ∪ ⋃(λe. case edge-push e of [] ⇒ {} |
   a#y ⇒ (case edge-read e of Some A ⇒ {} |
         None ⇒ FUN2LR1-EAnnPush2 e (epda-states G)))`(epda-delta G)

definition FUN2LR1 :: ('q,'a,'b)EPDA
 ⇒ (('q,'b) LR1State,'a)CFG
 where
 FUN2LR1 G ≡
 (|cfg-nonterms={l2 q A| q A. q ∈ epda-states G ∧ A ∈ epda-gamma G}
 ∪{l3 q1 A q2| q1 A q2. q1 ∈ epda-states G ∧ q2 ∈ epda-states G ∧ A ∈ epda-gamma G},
 cfg-sigma=epda-sigma G,
 cfg-initial=l2 (epda-initial G) (epda-box G),
 cfg-prods=FUN2LR1-Edges1 G ∪ FUN2LR1-Edges2 G|)
```

### A.2.8 Dollar Augmentation

```
definition FUNDollarAugment :: ('a,'b)CFG
 ⇒ 'a
 ⇒ 'b
 ⇒ ('a,'b)CFG
 where
 FUNDollarAugment G S' Do ≡
 (|cfg-nonterms=cfg-nonterms G∪{S'},
 cfg-sigma=cfg-sigma G∪{Do},
 cfg-initial=S',
 cfg-prods=cfg-prods G∪{(|prod-lhs=S',prod-rhs=[beB Do,beA (cfg-initial G),beB Do]|)}|)
```

### A.2.9 Valid-Operations

```
definition FUNdesc11 :: ('a, 'b) CFG
 ⇒ nat
 ⇒ ('a, 'b)ITEM
 ⇒ ('a, 'b)ITEM set
 where
 FUNdesc11 G k I = {x. ∃B w z β. x =
 (|item-lhs = B,
 item-rhs1 = [],
 item-rhs2 = w,
 item-la = z|)
 ∧ (|prod-lhs = B,prod-rhs = w|) ∈ cfg-prods G
 ∧ item-rhs2 I = beA B # β
 ∧ (z ∈ FIRST G k (β @ (Set2Conv (item-la I))))}
 ∪{I}

definition FUNdesc1 :: ('a, 'b) CFG
 ⇒ nat
 ⇒ ('a, 'b)ITEM set
 ⇒ ('a, 'b)ITEM set
 where
 FUNdesc1 G k  S = {I. ∃x ∈ S. I ∈ FUNdesc11 G k x}

function (domintros) FUNdesc :: ('a, 'b) CFG
```

$\Rightarrow$ *nat*
$\Rightarrow$ ('*a*, '*b*)*ITEM set*
$\Rightarrow$ ('*a*, '*b*)*ITEM set*
**where**
*FUNdesc G k S* = (
*if*(*FUNdesc1 G k S* = *S*)*then S*
*else FUNdesc G k* (*FUNdesc1 G k S*))
**by** *pat-completeness auto*

**definition** *FUNpassesX* :: ('*a*,'*b*)*biElem*
$\Rightarrow$ ('*a*,'*b*)*ITEM*
$\Rightarrow$ ('*a*,'*b*)*ITEM*
$\Rightarrow$ *bool*
**where**
*FUNpassesX X I1 I2* $\equiv$
*item-lhs I1* = *item-lhs I2*
$\wedge$ *item-la I1* = *item-la I2*
$\wedge$ ($\exists \beta$. *item-rhs2 I1* = *X* # $\beta$ $\wedge$ *item-rhs1 I2* = *item-rhs1 I1* @ [*X*] $\wedge$ *item-rhs2 I2* = $\beta$)

**definition** *FUNBASIS* :: ('*a*,'*b*)*biElem*
$\Rightarrow$ ('*a*,'*b*)*ITEM set*
$\Rightarrow$ ('*a*,'*b*)*ITEM set*
**where**
*FUNBASIS X S* $\equiv$ {*I2*. $\exists$ *I1*$\in$*S*. *FUNpassesX X I1 I2*}

**definition** *FUNGOTO* :: ('*a*,'*b*)*CFG*
$\Rightarrow$ *nat*
$\Rightarrow$ ('*a*,'*b*)*biElem*
$\Rightarrow$ ('*a*,'*b*)*ITEM set*
$\Rightarrow$ ('*a*,'*b*) *ITEM set*
**where**
*FUNGOTO G k X S* $\equiv$ *FUNdesc G k* (*FUNBASIS X S*)

**definition** *FUNdescInitial* :: ('*a*,'*b*)*CFG*
$\Rightarrow$ ('*a*,'*b*)*ITEM set*
**where**
*FUNdescInitial G* $\equiv$ {*I*. $\exists$ *p*$\in$*cfg-prods G*.
*prod-lhs p*=*cfg-initial G*
$\wedge$ *item-lhs I*=*cfg-initial G*
$\wedge$ *item-rhs1 I*=[]
$\wedge$ *item-rhs2 I* = *prod-rhs p*
$\wedge$ *item-la I* = []}

**definition** *FUNValidEmpty* :: ('*a*,'*b*)*CFG*
$\Rightarrow$ *nat*
$\Rightarrow$ ('*a*,'*b*)*ITEM set*
**where**
*FUNValidEmpty G k* $\equiv$ *FUNdesc G k* (*FUNdescInitial G*)

### *A.2.10 LR1-Machine*

**definition** *FUNLRM-once* :: ('*a*,'*b*)*CFG*
$\Rightarrow$ *nat*

⇒ (('a,'b)ITEM set)set
⇒ (('a,'b)ITEM set,('a,'b)biElem,nat)EDGE set
**where**
FUNLRM-once G k S ≡ (λ(q,X).
(|edge-src=q,
edge-read=Some X,
edge-pop=[o::nat],
edge-push=[o::nat],
edge-trg=FUNGOTO G k X q|)
) ' (S×(biElemSet (cfg-nonterms G) (cfg-sigma G)))

**function** (domintros) FUNLRM-loop :: ('a,'b)CFG
⇒ nat
⇒ (('a,'b)ITEM set,('a,'b)biElem,nat)EDGE set
⇒ (('a,'b)ITEM set)set ⇒ (('a,'b)ITEM set)set
⇒ ((('a,'b)ITEM set)set × (('a,'b)ITEM set,('a,'b)biElem,nat)EDGE set)
**where**
FUNLRM-loop G k E V S = (
if FUNLRM-once G k S = {} then (V,E)
else FUNLRM-loop G k
(E∪(FUNLRM-once G k S))
(V∪S)
((edge-trg ' (FUNLRM-once G k S))−(V∪S)))
**by** pat-completeness auto

**definition** FUNLRM :: ('a,'b)CFG
⇒ nat
⇒ (('a,'b)ITEM set,('a,'b)biElem ,nat)EPDA
**where**
FUNLRM G k ≡
(|epda-states=fst(FUNLRM-loop G k {} {} {FUNValidEmpty G k}),
epda-sigma=biElemSet (cfg-nonterms G) (cfg-sigma G),
epda-gamma={o::nat},
epda-delta=snd(FUNLRM-loop G k {} {} {FUNValidEmpty G k}),
epda-initial=FUNValidEmpty G k,
epda-box=o::nat,epda-final={}|)

### A.2.11 Remove Nonproductive Productions

**definition** FUNRNPP1 :: ('a,'b)CFG
⇒ 'a set
⇒ 'a set
**where**
FUNRNPP1 GI N = N∪{A. ∃p∈cfg-prods GI. prod-lhs p=A ∧ Set1BiElem (prod-rhs p) ⊆ N}

**function** (domintros) FUNRNPPL :: ('a,'b)CFG
⇒ 'a set
⇒ 'a set
**where**
FUNRNPPL GI N = (
if FUNRNPP1 GI N = N then N
else FUNRNPPL GI (FUNRNPP1 GI N))
**apply**(auto)

**done**

**definition** *FUNRNPP* :: ('a,'b)*CFG*
⇒ ('a,'b)*CFG option*
**where**
*FUNRNPP G* ≡ (
*if cfg-initial G* ∈ *FUNRNPPL G* {} *then Some*
(|*cfg-nonterms*=*FUNRNPPL G* {},
*cfg-sigma*=*cfg-sigma G*,
*cfg-initial*=*cfg-initial G*,
*cfg-prods*={*p*∈*cfg-prods G*. *Set1BiElem* ((*beA* (*prod-lhs p*))#(*prod-rhs p*)) ⊆ *FUNRNPPL G* {}}|)
*else None*)

### *A.2.12 First*

**definition** *FUNFirstDS* :: ('a, 'b) *CFG*
⇒ ('a, 'b) *biElem list set*
**where**
*FUNFirstDS G* = {*As*. ∃*e A*. *prod-lhs e* = *A* ∧ [*beA A*] = *As* ∧ *e* ∈ *cfg-prods G*}
∪ {*ws*. ∃*e w*. *prod-rhs e* = *w* ∧ *e* ∈ *cfg-prods G* ∧ *ws* ∈ *PostCl w*}

**definition** *FUNfirst1* :: ('a, 'b) *CFG*
⇒ (('a, 'b) *biElem list*
⇒ ('b *option*)*set*)
⇒ (('a, 'b) *biElem list*
⇒ ('b *option*) *set*)
**where**
*FUNfirst1 G f* = (λ*w*.
*if w* ∉ *FUNFirstDS G then* {}
*else f w* ∪ (
*case w of* [] ⇒ {*None*} |
*beA A* # *w'* ⇒
*f* [*beA A*]
− (*if w'* ≠ [] *then* {*None*} *else* {})
∪ (*if None* ∈ *f* [*beA A*] *then f w' else* {})
∪ ({*b*. ∃*x*. *b* ∈ *f x* ∧ (|*prod-lhs* = *A*, *prod-rhs* = *x*|) ∈ *cfg-prods G*}
− (*if w'* ≠ [] *then* {*None*} *else* {})) |
*beB a* # *w'* ⇒ {*Some a*}))

**function** (*domintros*) *FUNfirstL* :: ('a, 'b) *CFG*
⇒ (('a, 'b) *biElem list* ⇒ ('b *option*)*set*)
⇒ (('a, 'b) *biElem list* ⇒ ('b *option*)*set*)
**where**
*FUNfirstL G f* = (
*if FUNfirst1 G f* = *f then f*
*else FUNfirstL G* (*FUNfirst1 G f*))
**by** *pat-completeness auto*

**definition** *FUNfirstA* :: ('a, 'b) *CFG*
⇒ ('a, 'b) *biElem list*
⇒ 'b *option set*
**where**
*FUNfirstA G w* = *FUNfirstL G* (λ*x*. {}) *w*

```
function FUNfirst :: ('a, 'b) CFG
 ⇒ ('a, 'b) biElem list
 ⇒ 'b option set
 where
 FUNfirst G w = (
 case w of [] ⇒ {None} |
 beA A # w' ⇒
  (FUNfirstA G [beA A] − {None})
  ∪ (if None ∈ FUNfirstA G [beA A] then FUNfirst G w' else {}) |
 beB b # w' ⇒ {Some b})
by pat-completeness auto
termination by lexicographic-order

primrec BiElemFirst1 :: ('a, 'b) biElem list
 ⇒ 'a option
 where
 BiElemFirst1 [] = None
 | BiElemFirst1 (x#w) = (case x of beA A ⇒ Some A | beB x ⇒ BiElemFirst1 w)

definition FUNfirstReduced :: ('a,'b)CFG
 ⇒ ('a,'b) biElem list
 ⇒ 'b option set
 where
 FUNfirstReduced G w ≡ (
 if Set1BiElem w ⊆ cfg-nonterms G then FUNfirst G w
 else {})

definition FUNfirstAll :: ('a,'b)CFG
 ⇒ ('a,'b) biElem list
 ⇒ 'b option set
 where
 FUNfirstAll G w ≡
 if (Set1BiElem w={})then FUNfirst G w
 else (
  case BiElemFirst1 w of None ⇒ {} |
  Some A ⇒ (
   case (FUNRNPP (G(|cfg-initial:=A|))) of Some G' ⇒ FUNfirstReduced G' w |
   None ⇒ {}))

definition FUNfirst-leq1 :: ('a, 'b) CFG
 ⇒ nat
 ⇒ ('a, 'b) biElem list
 ⇒ 'b list set
 where
 FUNfirst-leq1 G k w ≡ (
 if k=0 then (if FUNfirstAll G w≠{} then {[]} else {})
 else (if k=Suc 0 then (λx. case x of None ⇒ [] | Some a ⇒ [a]) ' (FUNfirstAll G w)
 else {}))
```

### A.2.13 Step and Step-Sequences for EPDA

```
definition FUNEPDAGOTO :: ('q,'a,'b)EPDA
```

⇒ 'q
⇒ 'a
⇒ 'q set
**where**
FUNEPDAGOTO M q X ≡ {q'.
(|edge-src=q,
edge-read=Some X,
edge-pop=[epda-box M],
edge-push=[epda-box M],
edge-trg=q'|)
∈epda-delta M}

**primrec** FUNEPDAGOTOseq :: ('q,'a,'b)EPDA
⇒ 'q
⇒ 'a list
⇒ 'q list set
**where**
FUNEPDAGOTOseq M q [] = {[]}
| FUNEPDAGOTOseq M q (X#w) = {p#p-seq|p p-seq.
p∈FUNEPDAGOTO M q X
∧ p-seq ∈ FUNEPDAGOTOseq M p w}

**definition** DFAGOTO :: ('q,'a,'b)EPDA
⇒ 'q
⇒ 'a
⇒ 'q
**where**
DFAGOTO M q X ≡ (THE-default q (λx. x∈(FUNEPDAGOTO M q X)))

**definition** DFAGOTOseq :: ('q,'a,'b)EPDA
⇒ 'q
⇒ 'a list
⇒ 'q list
**where**
DFAGOTOseq M q w ≡ (THE-default [] (λx. x∈(FUNEPDAGOTOseq M q w)))

### A.2.14 LR1-Parser

**definition** FUNLRP-Rules :: ('a,'b)CFG
⇒ ('a,'b)CFG
⇒ (('a,'b)ITEM set,('a,'b)biElem,nat)EPDA
⇒ nat
⇒ ((('a,'b)ITEM set,'b)RULE × ('a, 'b) PRODUCTION option) set
**where**
FUNLRP-Rules G G' M k =
{(
(|rule-lpop=q#q-seq,
rule-rpop=y,
rule-lpush=[q,qA],
rule-rpush=y|),
Some (|prod-lhs=item-lhs I,
prod-rhs=(item-rhs1 I)@(item-rhs2 I)|))
| q q-seq (I::('a, 'b) ITEM) y qA.

```
  q∈epda-states M
  ∧ I∈q
  ∧ (|prod-lhs=item-lhs I,prod-rhs=item-rhs2 I|)∈cfg-prods G
  ∧ (item-rhs1 I=[])
  ∧ qA = (DFAGOTO M q (beA (item-lhs I)))
  ∧ q-seq = (DFAGOTOseq M q (item-rhs2 I))
  ∧ (|item-lhs=item-lhs I,item-rhs1=item-rhs2 I,item-rhs2=[],item-la=y|) ∈ last (q#q-seq)}
  ∪
  {(
  (|rule-lpop=[q],
  rule-rpop=a#y,
  rule-lpush=[q,qA],
  rule-rpush=y|),None)
  | q a y qA.
  (q∈epda-states M)
  ∧ (∃I∈q.
  (|prod-lhs=item-lhs I,prod-rhs=item-rhs1 I @ item-rhs2 I|)∈cfg-prods G
  ∧ [beB a] = take (Suc o) (item-rhs2 I)
  ∧ qA = (DFAGOTO M q (beB a))
  ∧ y∈(FUNfirst-leq1 G' (k−(1::nat)) ((drop (Suc o) (item-rhs2 I))@(Set2Conv (item-la I)))))}

definition FUNLRP :: ('a,'b)CFG
  ⇒ ('a,'b)CFG
  ⇒ (('a,'b)ITEM set,('a,'b)biElem,nat)EPDA
  ⇒ nat
  ⇒ 'b
  ⇒ (('a,'b)ITEM set,'b,(('a,'b)PRODUCTION option)option))PARSER
  where
  FUNLRP G G' M k Do =
   (|parser-nonterms=(epda-states M) − {epda-initial M, last (DFAGOTOseq M (epda-initial M) [beB
Do,beA (cfg-initial G),beB Do]), DFAGOTO M (epda-initial M) (beA (cfg-initial G))},
  parser-sigma=(cfg-sigma G'),
  parser-initial=DFAGOTO M (epda-initial M) (beB Do),
  parser-final={last (DFAGOTOseq M (epda-initial M) [beB Do,beA (cfg-initial G)])},
  parser-rules=(λ(x,y). x) ' (FUNLRP-Rules G G' M k),
  parser-output=(λx. if (∃!y. (x,y)∈(FUNLRP-Rules G G' M k))
    then Some(THE y. (x,y)∈(FUNLRP-Rules G G' M k))
    else None),
  parser-bottom=Do|)
```

### A.2.15 Drop Input Bottom Rules

```
definition FUNDIBR :: ('a,'b,'c option)PARSER
  ⇒ ('a,'b,'c option)PARSER
  where
  FUNDIBR P = (
  let R={r. r∈(parser-rules P) ∧ rule-rpop r ≠ [parser-bottom P]} in
  (P
  (|parser-rules:=R,
  parser-final:={n. ∃r∈parser-rules P. rule-rpop r = [parser-bottom P] ∧ n=last(rule-lpop r)},
  parser-output:=(λr. if (r∈R) then parser-output P r else None)|)))
```

### A.2.16 Remove Top Rules

**definition** *SmapA* :: *'a*
⇒ *'b*
⇒ *'a* × *'b*
**where**
*SmapA a x* ≡ (*a*,*x*)

**definition** *WmapA* :: *'a list*
⇒ *'b list*
⇒ (*'a* × *'b list*) *list*
**where**
*WmapA w x* ≡ (
*case w of* [] ⇒ [] |
*a'#w'* ⇒
*map* (λ*a*. *SmapA a* []) (*butlast*(*w*)) @
(*map* (λ*a*. *SmapA a x*) [*last w*]))

**definition** *FUNRTR-S1* :: (*'a*,*'b*,*'c option*)*PARSER*
⇒ (*'a* × *'b list*, *'b*) *RULE set*
**where**
*FUNRTR-S1 P* ≡ {
(|*rule-lpop*=*WmapA* (*rule-lpop r*) [],
*rule-rpop*=*rule-rpop r*,
*rule-lpush*=*WmapA* (*rule-lpush r*) [],
*rule-rpush*=[] |)
| *r*. *r*∈(*parser-rules P*) ∧ *rule-rpush r* = []}

**definition** *FUNRTR-S2* :: (*'a*,*'b*,*'c option*)*PARSER*
⇒ (*'a* × *'b list*, *'b*) *RULE set*
**where**
*FUNRTR-S2 P* ≡ {
(|*rule-lpop*=*WmapA* (*rule-lpop r*) (*rule-rpop r*),
*rule-rpop*=[],
*rule-lpush*=*WmapA* (*rule-lpush r*) [],
*rule-rpush*=[] |)
| *r*. *r*∈(*parser-rules P*) ∧ *rule-rpush r* = []}

**definition** *FUNRTR-R1* :: (*'a*,*'b*,*'c option*)*PARSER*
⇒ (*'a* × *'b list*, *'b*) *RULE set*
**where**
*FUNRTR-R1 P* ≡ {
(|*rule-lpop*=*WmapA* (*rule-lpop r*) [],
*rule-rpop*=*rule-rpop r*,
*rule-lpush*=*WmapA* (*rule-lpush r*) (*rule-rpop r*),
*rule-rpush*=[] |)
| *r*. *r*∈(*parser-rules P*) ∧ *rule-rpush r* = *rule-rpop r*}

**definition** *FUNRTR-R2* :: (*'a*,*'b*,*'c option*)*PARSER*

⇒ ('a × 'b list, 'b) RULE set
**where**
FUNRTR-R2 P ≡ {
(|rule-lpop=WmapA (rule-lpop r) (rule-rpop r),
rule-rpop=[],
rule-lpush=WmapA (rule-lpush r) (rule-rpop r),
rule-rpush=[]|)
| r. r∈(parser-rules P) ∧ rule-rpush r = rule-rpop r}

**definition** FUNRTR :: ('a,'b,'c option)PARSER
⇒ ('a × ('b list),'b,'c option)PARSER
**where**
FUNRTR P = (
let R = (FUNRTR-S1 P) ∪ (FUNRTR-S2 P) ∪ (FUNRTR-R1 P) ∪ (FUNRTR-R2 P) in
let N = {n. ∃r∈R. n ∈ set (rule-lpop r) ∨ n ∈ set(rule-lpush r)}∪
{SmapA (parser-initial P) []} in
((|parser-nonterms=N,
parser-sigma=parser-sigma P,
parser-initial=SmapA (parser-initial P) [],
parser-final=N ∩ ((λa. SmapA a []) ' (parser-final P)),
parser-rules=R,
parser-output=(λr. None),
parser-bottom=parser-bottom P
|)))

### A.2.17 Remove Top Rule

**definition** list2option :: 'a list
⇒ 'a option
**where**
list2option w = (case w of [] ⇒ None | a#w' ⇒ Some a)

**definition** FUNP2A-R :: ('a,'b)RULE
⇒ ('a,'b,'a)EDGE
**where**
FUNP2A-R e ≡
(|edge-src=last (rule-lpop e),
edge-read=list2option (rule-rpop e),
edge-pop=rev(butlast (rule-lpop e)),
edge-push=rev(butlast (rule-lpush e)),
edge-trg=last (rule-lpush e)|)

**definition** FUNP2A :: ('a,'b,'c)PARSER
⇒ 'a
⇒ ('a,'b,'a)EPDA
**where**
FUNP2A G BOX ≡
(|
epda-states=parser-nonterms G,
epda-sigma=parser-sigma G − {parser-bottom G},
epda-gamma=parser-nonterms G ∪ {BOX},
epda-delta=FUNP2A-R ' (parser-rules G),
epda-initial=parser-initial G,

```
  epda-box=BOX,
  epda-final=parser-final G
  |)
```

### A.2.18 Replace Zero-Popping Edges

```
definition FUN-REPZact :: ('q,'a,'b)EDGE
  ⇒ 'b set
  ⇒ ('q,'a,'b)EDGE set
  where
  FUN-REPZact e X ≡ {
  (|edge-src=edge-src e,
  edge-read=edge-read e,
  edge-pop=[x],
  edge-push=(edge-push e)@[x],
  edge-trg=edge-trg e|)
  | x. x∈X}

definition FUN-REPZ :: ('q,'a,'b symbol)EPDA
  ⇒ ('q, 'a, 'b symbol) EPDA
  where
  FUN-REPZ G ≡
  (|epda-states=epda-states G,
  epda-sigma=epda-sigma G,
  epda-gamma=epda-gamma G,
  epda-delta=⋃(λe. if edge-pop e=[] then FUN-REPZact e (epda-gamma G) else {e})`(epda-delta G),
  epda-initial=epda-initial G,
  epda-box=epda-box G,
  epda-final=epda-final G|)
```

### A.2.19 Replace Multiple-Popping Edges

```
datatype ('q,'b) REPMState =
  old 'q
  | new 'q 'b list

definition FUN-REPMstate :: 'q
  ⇒ 'b list
  ⇒ nat
  ⇒ ('q,'b)REPMState
  where
  FUN-REPMstate q w i ≡ (
  if i=0 then old q
  else new q (take i w))

definition FUN-REPMact :: ('q,'a,'b)EDGE
  ⇒ (('q,'b)REPMState,'a,'b)EDGE set
  where
  FUN-REPMact e ≡ {
  (|edge-src=FUN-REPMstate (edge-src e) (edge-pop e) i,
  edge-read=(if i=length(edge-pop e)then edge-read e else None),
  edge-pop=(if i<length(edge-pop e) then [edge-pop e!i] else [edge-pop e!(i − 1)]),
  edge-push=if i=length(edge-pop e) then edge-push e
```

```
  else (if i=length(edge-pop e) − 1 then [edge-pop e!i]
  else []),
  edge-trg=if i<length(edge-pop e) then FUN-REPMstate (edge-src e) (edge-pop e) (Suc i)
   else FUN-REPMstate (edge-trg e) [] o|)
  | i. i≤length(edge-pop e)}

definition FUN-REPM :: ('q,'a,'b symbol)EPDA
  ⇒ (('q,'b symbol)REPMState, 'a, 'b symbol) EPDA
  where
  FUN-REPM G ≡
  (|epda-states={FUN-REPMstate q (edge-pop e) i
  | q i e. q∈epda-states G ∧ e∈epda-delta G ∧ i≤length(edge-pop e)},
  epda-sigma=epda-sigma G,
  epda-gamma=epda-gamma G,
  epda-delta=⋃(λe. FUN-REPMact e)'(epda-delta G),
  epda-initial=old (epda-initial G),
  epda-box=epda-box G,
  epda-final=old '(epda-final G)|)
```

### A.2.20 Convert EDPDA to DPDA

```
definition FUN-EDPDA2DPDA :: ('q,'a,'b symbol)EPDA
  ⇒ (('q, 'b symbol) REPMState, 'a, 'b symbol) EPDA
  where
  FUN-EDPDA2DPDA G ≡ FUN-REPZ (FUN-REPM (FUN-REPZ G))
```

### A.2.21 Symbolising

```
definition CFGtoSymbol :: ('a,'b)CFG
  ⇒ ('a symbol,'b symbol)CFG
  where
  CFGtoSymbol G ≡
  (|cfg-nonterms=symbol-atomic ' (cfg-nonterms G),
  cfg-sigma=symbol-atomic ' (cfg-sigma G),
  cfg-initial=symbol-atomic (cfg-initial G),
  cfg-prods={
  (|prod-lhs=symbol-atomic(prod-lhs p),
   prod-rhs=map (λx. case x of beA A ⇒ beA (symbol-atomic A) | beB b ⇒ beB (symbol-atomic b))
(prod-rhs p)|)
  |p. p∈cfg-prods G}|)

definition EPDAtoSymbol :: ('q,'a,'b)EPDA
  ⇒ ('qx symbol,'ax symbol,'bx symbol)EPDA
  where
  EPDAtoSymbol G ≡
  (let fq=SOME f. inj f; fg=SOME f. inj f; fs=SOME f. inj f in
  (|epda-states=fq ' epda-states G,
  epda-sigma=symbol-atomic ' (fs ' epda-sigma G),
  epda-gamma=fg ' (epda-gamma G),
  epda-delta={
  (|edge-src=fq (edge-src e),
  edge-read=
  case edge-read e of None ⇒ None |
```

```
  Some a ⇒ Some (symbol-atomic (fs a)),
  edge-pop=map fg (edge-pop e),
  edge-push=map fg (edge-push e),
  edge-trg=fq(edge-trg e)⦇)
  | e. e ∈ epda-delta G},
  epda-initial=fq (epda-initial G),
  epda-box=fg (epda-box G),
  epda-final=fq'(epda-final G)⦇))

definition PARSERtoSymbol :: ('q,'a,'b)PARSER
  ⇒ ('p symbol,'c symbol,nat option)PARSER
  where
  PARSERtoSymbol G ≡
  (let f=SOME f. inj f;fs=SOME f. inj f in
  (⦇parser-nonterms=f ' (parser-nonterms G),
  parser-sigma=symbol-atomic ' (fs'parser-sigma G),
  parser-initial=f(parser-initial G),
  parser-final=f'(parser-final G),
  parser-rules={
  (⦇rule-lpop=map f (rule-lpop e),
  rule-rpop=map (symbol-atomic∘fs) (rule-rpop e),
  rule-lpush=map f (rule-lpush e),
  rule-rpush=map (symbol-atomic∘fs) (rule-rpush e)⦇)
  | e. e∈parser-rules G},
  parser-output=(λx. None),
  parser-bottom=symbol-atomic (fs(parser-bottom G))⦇))
```

### A.2.22 Restrict to Edges

```
definition FUN-R2E :: ('q,'a,'b)EPDA
  ⇒ ('q,'a,'b)EDGE set ⇒ ('q,'a,'b)EPDA option
  where
  FUN-R2E G E = (let Q={q∈epda-states G. ∃e∈E. edge-src e=q ∨ edge-trg e=q} in
  (if (epda-initial G) ∈ Q then
  Some (⦇epda-states=Q,
  epda-sigma=epda-sigma G,
  epda-gamma=epda-gamma G,
  epda-delta=E,
  epda-initial=epda-initial G,
  epda-box=epda-box G,
  epda-final=Q ∩ (epda-final G)⦇)
  else None))
```

### A.2.23 Restrict to States

```
definition FUN-R2Q :: ('q,'a,'b)EPDA
  ⇒ 'q set
  ⇒ ('q,'a,'b)EPDA option
  where
  FUN-R2Q M Q= (if epda-initial M ∉ Q then None else Some
  (⦇epda-states=Q,
  epda-sigma=epda-sigma M,
  epda-gamma=epda-gamma M,
```

*epda-delta=*{*e*. *e*∈*epda-delta M* ∧ *edge-src e* ∈ *Q* ∧ *edge-trg e* ∈ *Q*},
*epda-initial=epda-initial M*,
*epda-box=epda-box M*,
*epda-final=epda-final M* ∩ *Q*|)
)

### *A.2.24 Accessibility*

**definition** *FUN-AC-RevLR1* :: (*'q*,*'a*,*'b*)*EPDA*
⇒ ((*'q*,*'b*) *LR1State*,*'a*)*PRODUCTION set*
⇒ (*'q*,*'a*,*'b*)*EDGE set*
**where**
*FUN-AC-RevLR1 G X* = {*e*∈*epda-delta G*. ∃*p*∈*X*.
*p* ∈ *FUN2LR1-EAnnRead1 e* (*epda-states G*)
∨ *p* ∈ *FUN2LR1-EAnnPop1 e*
∨ *p* ∈ *FUN2LR1-EAnnPush1 e* (*epda-states G*)
∨ *p* ∈ *FUN2LR1-EAnnRead2 e* (*epda-states G*)
∨ *p* ∈ *FUN2LR1-EAnnPush2 e* (*epda-states G*)}

**definition** *FUN-AC-RevRMP* :: (*'q*,*'a*,*'b*)*EPDA*
⇒ ((*'q*,*'a*,*'b*) *SDPDA2State*,*'a*,*'b*)*EDGE set*
⇒ (*'q*,*'a*,*'b*)*EDGE set*
**where**
*FUN-AC-RevRMP G X* = {*e*∈*epda-delta G*. ∃*p*∈*X*.
*p* ∈ *FUNRMP-steps e*
∨ *p* ∈ *FUNRMP-steps e*}

**definition** *FUN-AC-RevSPP* :: (*'q*,*'a*,*'b*)*EPDA*
⇒ ((*'q*,*'a*,*'b*) *SDPDA1State*,*'a*,*'b*)*EDGE set*
⇒ (*'q*,*'a*,*'b*)*EDGE set*
**where**
*FUN-AC-RevSPP G X* = {*e*∈*epda-delta G*. ∃*p*∈*X*.
*p* ∈ *FUNSPP-NoOp e* (*epda-gamma G*)
∨ *p* = *FUNSPP-else e*}

**definition** *FUN-AC-RevNoOp* :: (*'q*,*'a*,*'b*)*EPDA*
⇒ *'b*
⇒ ((*'q*,*'a*,*'b*) *SDPDA1State*,*'a*,*'b*)*EDGE set*
⇒ (*'q*,*'a*,*'b*)*EDGE set*
**where**
*FUN-AC-RevNoOp G PB X* = {*e*∈*epda-delta G*. ∃*p*∈*X*.
*p* ∈ *FUNRNoOp-NoOp e PB*
∨ *p* = *FUNRNoOp-else e*}

**definition** *FUN-AC-RevSR* :: (*'q*,*'a*,*'b*)*EPDA*
⇒ ((*'q*,*'a*,*'b*) *SDPDA1State*,*'a*,*'b*)*EDGE set*
⇒ (*'q*,*'a*,*'b*)*EDGE set*
**where**
*FUN-AC-RevSR G X* = {*e*∈*epda-delta G*. ∃*p*∈*X*.
*p* ∈ *FUNSR-read e*
∨ *p* ∈ *FUNSR-else e*}

**definition** *FUN-AC-Edges* :: (*'q*,*'a*,*'b symbol*)*EPDA*

```
 ⇒ ('q, 'a, 'b symbol) EDGE set
 where
 FUN-AC-Edges Go =
 (let Go2 = FUNSR Go;
 PB = FUNfreshSymbol (epda-gamma Go2);
 Go3 = FUNRNoOp Go2 PB;
 Go4 = FUNSPP Go3;
 G1 = FUNRMP Go4;
 G2 = FUN2LR1 G1;
 G3 = FUNRNPP G2
 in (
 case G3 of Some G3' ⇒
  FUN-AC-RevSR Go (
  FUN-AC-RevNoOp Go2 PB (
  FUN-AC-RevSPP Go3 (
  FUN-AC-RevRMP Go4 (
  FUN-AC-RevLR1 G1 (cfg-prods G3')))))) |
 None ⇒ {}))

definition FUN-AC :: ('q,'a,'b symbol)EPDA
 ⇒ ('q,'a,'b symbol)EPDA option
 where
 FUN-AC G = FUN-R2E G (FUN-AC-Edges G)
```

### A.2.25 Ensure Blockfreeness

```
definition FUN-BlockFree :: ('q symbol,'a symbol,'q symbol)EPDA
 ⇒ ('q symbol, 'a symbol, 'q symbol) EPDA option
 where
 FUN-BlockFree Go ≡ (
 let G1 = FUN-2SDPDA Go;
 G2 = FUNNDA G1;
 G3 = FUN2LR1 G2;
 G4RNPP = FUNRNPP G3
 in (
 case G4RNPP of None ⇒ None
 | Some G4RNPPSome ⇒ (let
 G = CFGtoSymbol G4RNPPSome;
 Do = FUNfreshSymbol (cfg-sigma G);
 S' = FUNfreshSymbol (cfg-nonterms G);
 G' = FUNDollarAugment G S' Do;
 M = FUNLRM G' (Suc o);
 Po = FUNLRP G G' M (Suc o) Do;
 P1 = FUNDIBR Po;
 P2 = FUNRTR P1;
 P2' = PARSERtoSymbol P2;
 G4 = FUNP2A P2' (FUNfreshSymbol (parser-nonterms P2'))
 in FUN-AC G4)))
```

### A.2.26 Split

```
datatype ('a,'b) splitstates =
 rear 'a 'b
```

```
  | sear 'a 'b
  | rhead 'a
  | shead 'a

definition FUN-lamLeave :: ('q,'a,'b)EDGE set
  ⇒ 'q set
  where
  FUN-lamLeave E = {q. (∃e∈E. edge-src e=q ∧ edge-read e=None)}

definition FUN-SPLIT-RH2RE :: ('q,'a,'b)EPDA
  ⇒ (('q, 'b) splitstates, 'a, 'b) EDGE set
  where
  FUN-SPLIT-RH2RE M = {
  (|edge-src=rhead q,
  edge-read=None,
  edge-pop=[γ],
  edge-push=[γ],
  edge-trg=rear q γ|)
  | q γ. q∈epda-states M ∧ γ∈epda-gamma M}

definition FUN-SPLIT-SH2SE :: ('q,'a,'b)EPDA
  ⇒ (('q, 'b) splitstates, 'a, 'b) EDGE set
  where
  FUN-SPLIT-SH2SE M = {
  (|edge-src=shead q,
  edge-read=None,
  edge-pop=[γ],
  edge-push=[γ],
  edge-trg=sear q γ|)
| q γ. q∈epda-states M ∧ γ∈epda-gamma M}

definition FUN-SPLIT-RE2RH :: ('q,'a,'b)EPDA
  ⇒ (('q, 'b) splitstates, 'a, 'b) EDGE set
  where
  FUN-SPLIT-RE2RH M = {
  (|edge-src=rear q γ,
  edge-read=x,
  edge-pop=[γ],
  edge-push=w,
  edge-trg=rhead q'|)
  | q γ x w q'.
  (|edge-src=q,
  edge-read=x,
  edge-pop=[γ],
  edge-push=w,
  edge-trg=q'|)
  ∈epda-delta M ∧ (q∉epda-final M ∨ x≠None)}

definition FUN-SPLIT-RE2SH :: ('q,'a,'b)EPDA
  ⇒ (('q, 'b) splitstates, 'a, 'b) EDGE set
  where
  FUN-SPLIT-RE2SH M = {
  (|edge-src=rear q γ,
```

```
  edge-read=None,
  edge-pop=[γ],
  edge-push=w,
  edge-trg=shead q'|)
  | q γ w q'.
  (|edge-src=q,
  edge-read=None,
  edge-pop=[γ],
  edge-push=w,
  edge-trg=q'|)
∈epda-delta M ∧ q∈epda-final M}

definition FUN-SPLIT-SE2SH :: ('q,'a,'b)EPDA
  ⇒ (('q, 'b) splitstates, 'a, 'b) EDGE set
  where
  FUN-SPLIT-SE2SH M = {
  (|edge-src=sear q γ,
  edge-read=None,
  edge-pop=[γ],
  edge-push=w,
  edge-trg=shead q'|)
  | q γ w q'.
  (|edge-src=q,
  edge-read=None,
  edge-pop=[γ],
  edge-push=w,
  edge-trg=q'|)
  ∈epda-delta M}

definition FUN-SPLIT-SE2RH :: ('q,'a,'b)EPDA
  ⇒ (('q, 'b) splitstates, 'a, 'b) EDGE set
  where
  FUN-SPLIT-SE2RH M = {
  (|edge-src=sear q γ,
  edge-read=x,
  edge-pop=[γ],
  edge-push=w,
  edge-trg=rhead q'|)
  | q γ x w q'.
  (|edge-src=q,
  edge-read=x,
  edge-pop=[γ],
  edge-push=w,
  edge-trg=q'|)
  ∈epda-delta M ∧ x≠None}

definition FUN-SPLIT-edges :: ('q,'a,'b)EPDA
  ⇒ (('q,'b)splitstates,'a,'b)EDGE set
  where
  FUN-SPLIT-edges M =
  FUN-SPLIT-RH2RE M
  ∪ FUN-SPLIT-SH2SE M
  ∪ FUN-SPLIT-RE2RH M
```

```
 ∪ FUN-SPLIT-RE2SH M
 ∪ FUN-SPLIT-SE2SH M
 ∪ FUN-SPLIT-SE2RH M

definition FUN-SPLIT :: ('q,'a,'b)EPDA
 ⇒ (('q,'b)splitstates,'a,'b)EPDA
 where
 FUN-SPLIT M =
 (|epda-states=
 (λ(q,X). rear q X) ` ((epda-states M)×(epda-gamma M))
 ∪(λ(q,X). sear q X) ` ((epda-states M)×(epda-gamma M))
 ∪(rhead ` (epda-states M))
 ∪(shead ` (epda-states M)),
 epda-sigma=epda-sigma M,
 epda-gamma=epda-gamma M,
 epda-delta=FUN-SPLIT-edges M,
 epda-initial=rhead (epda-initial M),
 epda-box=epda-box M,
 epda-final=(λ(q,X). rear q X) ` ((epda-final M)×(epda-gamma M))
 ∪ (λ(q,X). sear q X) ` ((epda-states M)×(epda-gamma M))
 −FUN-lamLeave (FUN-SPLIT-edges M)
 |)
```

### A.2.27 Remove Noncontrollable States

```
datatype ('a,'b) pair = pair 'a 'b

definition NCS :: ((('qa,'qb)pair,'b)splitstates,'a,'b)EPDA
 ⇒ ('qa,'a,'c)EPDA
 ⇒ 'a set
 ⇒ (('qa,'qb)pair,'b)splitstates set
 where
 NCS M P SigmaUC = {ear. ∃p q γ.
 (ear=rear (pair p q) γ ∨ ear=sear (pair p q) γ)
 ∧ (ear ∉ FUN-lamLeave (epda-delta M))
 ∧ (∃u∈SigmaUC. ∃ep∈epda-delta P.
 edge-src ep=p
 ∧ edge-read ep=Some u
 ∧ (¬(∃em∈epda-delta M. edge-src em=ear ∧ edge-read em=Some u ∧ edge-pop em=[γ])))
 }

definition RNCS :: ((('qa,'qb)pair,'b)splitstates,'a,'b)EPDA
 ⇒ ('qa,'a,'c)EPDA
 ⇒ 'a set
 ⇒ (((((('qa,'qb)pair,'b)splitstates,'a,'b)EPDA option)×bool)
 where
 RNCS M P SigmaUC = (let Q = NCS M P SigmaUC in (FUN-R2Q M Q,Q={}))
```

### A.2.28 Product Automaton

```
definition FUNtimes :: ('qa,'a,'b)EPDA
 ⇒ ('qb,'a,nat)EPDA
 ⇒ (('qa,'qb)pair,'a,'b)EPDA
```

**where**
*FUNtimes C P =*
*(|epda-states={pair p q| p q. p∈ epda-states C ∧ q ∈ epda-states P},*
*epda-sigma=(epda-sigma C)∩(epda-sigma P),*
*epda-gamma=epda-gamma C,*
*epda-delta={*
*(|edge-src=pair (edge-src e) p,*
*edge-read=None,*
*edge-pop=edge-pop e,*
*edge-push=edge-pop e,*
*edge-trg=pair (edge-trg e) p|)*
*| e p. p∈epda-states P ∧ e∈epda-delta C ∧ edge-read e = None}∪*
*{*
*(|edge-src=pair (edge-src e) (edge-src e'),*
*edge-read=edge-read e,*
*edge-pop=edge-pop e,*
*edge-push=edge-pop e,*
*edge-trg=pair (edge-trg e) (edge-trg e')|)*
*| e e'. e∈epda-delta C ∧ e'∈epda-delta P ∧ edge-read e = edge-read e'},*
*epda-initial=pair (epda-initial C) (epda-initial P),*
*epda-box=epda-box C,*
*epda-final={pair p q| p q. p∈ epda-final C ∧ q ∈ epda-final P}*
*|)*

### *A.2.29 Ensure Controllability*

**definition** *FUN-Controllable :: ('q symbol,'a symbol,'q symbol)EPDA*
*⇒ ('q symbol,'a symbol,nat)EPDA*
*⇒ 'a symbol set*
*⇒ (('q symbol,'a symbol,'q symbol)EPDA option)×bool*
**where**
*FUN-Controllable Go P SigmaUC ≡ (*
*let*
*Mtimes = FUNtimes Go P;*
*Msplit = FUN-SPLIT Mtimes;*
*Mac = FUN-AC Msplit*
*in*
*(case Mac of None ⇒ (None,False) | Some Mac' ⇒*
*case RNCS Mac' P SigmaUC of (None,x) ⇒ (None,x) |*
*(Some Mres,x) ⇒ (Some (EPDAtoSymbol Mres),x))*
*)*

### *A.2.30 Synthesize*

**definition** *FUN-iterator :: ('q symbol,'a symbol,'q symbol)EPDA*
*⇒ ('q symbol,'a symbol,nat)EPDA*
*⇒ 'a symbol set*
*⇒ (('q symbol,'a symbol,'q symbol)EPDA option)×bool*
**where**
*FUN-iterator X P SigmaUC = (*
*case FUN-Controllable X P SigmaUC of (None,x) ⇒ (None,False) |*
*(Some MCont',x) ⇒ (*
*if x then (Some MCont',False)*

```
   else (FUN-BlockFree MCont',True)))

function FUN-synthesize-loop :: ('q symbol,'a symbol,'q symbol)EPDA
  ⇒ ('q symbol,'a symbol,nat)EPDA
  ⇒ 'a symbol set
  ⇒ ('q symbol,'a symbol,'q symbol)EPDA option
  where
  FUN-synthesize-loop X P SigmaUC = (
  case FUN-iterator X P SigmaUC of (None,x) ⇒ None |
  (Some X',x) ⇒ (
    if x then FUN-synthesize-loop X' P SigmaUC
    else Some X'))
apply(force)+
done

definition FUN-synthesize :: ('q symbol,'a symbol,'q symbol)EPDA
  ⇒ ('q symbol,'a symbol,nat)EPDA
  ⇒ 'a symbol set
  ⇒ ('q symbol,'a symbol,'q symbol)EPDA option
  where
  FUN-synthesize S P SigmaUC = (
  let Init = FUN-BlockFree (EPDAtoSymbol (FUNtimes S P)) in
  (case Init of None ⇒ None |
  Some Init' ⇒ FUN-synthesize-loop (Init') P SigmaUC))

end
```

## A.3 Good Iterators

We present the lemmas, theorems, and corollary in Isabelle/HOL notation for Section 2.5. The quite lengthy proofs are omitted here and available from the authors.
**theory** *GFP-SOUND*
**imports**
*Fundamentals*
*LaTeXsugar*
**begin**

### *A.3.1 Language Basics*

**lemma** *preservePrec*:
**assumes** *asm*: $\bigwedge B.\ B \in C \Longrightarrow B = prec\ B$
**shows** $\bigcup C = prec\ (\bigcup C)$
⟨*proof*⟩

*required by:*
*lemma: lem1 (page 89)*

**lemma** *lem1*:
$\bigcup \{A::'a\ list \Rightarrow bool.\ A \subseteq LUM \wedge A = prec\ A\} = LUM$
$\Longrightarrow prec\ LUM = LUM$
⟨*proof*⟩

*depends on:*
*lemma: preservePrec (page 89)*

*required by:*
*theorem: GoodIterator-Fdes (page 94)*

**definition** *sconc* :: $'a\ list\ set \Rightarrow 'a\ set \Rightarrow 'a\ list\ set$ **where**
$sconc\ L\ A = \{w@[a] |\ w\ a.\ w \in L \wedge a \in A\}$

**lemma** *lem2*:
$w' \in prec\ \{w\ @\ [a]\}$
$\Longrightarrow w' \neq w\ @\ [a]$
$\Longrightarrow w' \notin prec\ \{w\}$
$\Longrightarrow Q$
⟨*proof*⟩

*required by:*
*theorem: GoodIterator-Fcont (page 95)*

**lemma** *prec-closed-sets-closed-under-intersection*:
$A = prec\ A$
$\Longrightarrow B = prec\ B$
$\Longrightarrow prec\ (A \cap B) = A \cap B$
⟨*proof*⟩

*required by:*
*theorem: GoodIterator-Fspec (page 96)*

## A.3.2 Complete Lattice Basics

**lemma** *le-trans*:
$p \leq (q::'a::complete\text{-}lattice) \Longrightarrow q \leq r \Longrightarrow p \leq r$
⟨*proof*⟩

*required by:*
*theorem: GoodIterator-composeCond (page 91)*
*theorem: Supremum-to-gfp-initialDec (page 106)*
*theorem: GoodIterator-compose (page 90)*
*lemma: computation-via-computeN-initial-hlp (page 104)*
*lemma: gfp-Decomposition (page 105)*
*lemma: gfp-mixed-fixed-point (page 103)*
*lemma: computation-via-computeN-initialDec-hlp (page 105)*
*lemma: gfp-invariant (page 92)*
*theorem: Supremum-in-DESSIMMSol (page 112)*

**definition** *GoodIterator* :: $('a::complete\text{-}lattice \Rightarrow 'a) \Rightarrow ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool) \Rightarrow bool$ **where**
$GoodIterator\ F\ Qinp\ Qterm\ Qout \equiv ($
$(\forall X{\in}Collect\ Qinp.\ F\ X \leq X)$
$\wedge\ (\forall X{\in}Collect\ Qinp.\ Qterm\ X \longleftrightarrow (F\ X = X))$
$\wedge\ (\forall X{\in}Collect\ Qinp.\ Qout\ (F\ X))$
$\wedge\ (\forall X{\in}Collect\ Qinp.\ \forall Y{\in}Collect\ Qinp.\ X \leq Y \longrightarrow F\ X \leq F\ Y)$
$\wedge\ (\forall X{\in}Collect\ Qinp.\ \forall Y{\in}Collect\ Qinp.\ F\ X < X \longrightarrow Y < X \longrightarrow F\ Y = Y \longrightarrow Y \leq F\ X))$

**definition** *GoodIteratorDec* :: $('a::complete\text{-}lattice \Rightarrow 'a) \Rightarrow ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool) \Rightarrow bool$ **where**
$GoodIteratorDec\ F\ Qinp\ Qterm\ Qout \equiv ($
$(\forall X.\ F\ X \leq X)$
$\wedge\ (\forall X{\in}Collect\ Qinp.\ Qterm\ X \longleftrightarrow (F\ X = X))$
$\wedge\ (\forall X{\in}Collect\ Qinp.\ Qout\ (F\ X))$
$\wedge\ (\forall X.\ \forall Y.\ X \leq Y \longrightarrow F\ X \leq F\ Y)$
$\wedge\ (\forall X{\in}Collect\ Qinp.\ \forall Y{\in}Collect\ Qinp.\ F\ X < X \longrightarrow Y < X \longrightarrow F\ Y = Y \longrightarrow Y \leq F\ X))$

**lemma** *GoodIteratorDec-to-GoodIterator*:
$GoodIteratorDec\ F\ Qinp\ Qterm\ Qout$
$\Longrightarrow GoodIterator\ F\ Qinp\ Qterm\ Qout$
⟨*proof*⟩

*required by:*
*theorem: Supremum-to-gfp-initialDec (page 106)*
*lemma: gfp-Decomposition (page 105)*

**lemma** *GoodIterator-weakening*:
$GoodIterator\ F\ Qinp\ Qterm\ Qout$
$\Longrightarrow GoodIterator\ F\ Qinp\ Qterm\ UNIV$
⟨*proof*⟩

*required by:*
*theorem: GoodIteratorX-F-spec-cont2-bf-Fdes (page 99)*
*theorem: GoodIteratorX-F-spec-cont-bf-Fdes (page 98)*
*theorem: GoodIteratorX-F-spec-cont3-bf-Fdes (page 100)*

**theorem** *GoodIterator-compose*:
$GoodIterator\ F1\ Qinp\ Qterm1\ Qinter1$

$\Longrightarrow$ *GoodIterator F2 Qinter2 Qterm2 Qout2*
$\Longrightarrow$ $(\bigwedge Q.$ *Qinter1 Q* $\longrightarrow$ *Qinter2 Q* $)$
$\Longrightarrow$ *GoodIterator* $(F2 \circ F1)$ *Qinp* $(\lambda X.$ *Qterm1 X* $\wedge$ *Qterm2 X*$)$ *Qout2*
⟨*proof*⟩

*depends on:*
*lemma: le-trans (page 90)*

*required by:*
*lemma: GoodIterator-F-spec-cont (page 97)*
*lemma: GoodIterator-F-spec-cont-bf-Fdes (page 97)*
*theorem: Supremum-to-gfp-initialDec (page 106)*
*lemma: gfp-Decomposition (page 105)*
*lemma: GoodIterator-F-spec2-cont-bf (page 99)*
*lemma: GoodIterator-F-spec-cont3 (page 99)*
*lemma: GoodIterator-F-spec-cont3-bf (page 100)*
*lemma: GoodIterator-F-bf-spec (page 110)*
*lemma: GoodIterator-F-spec-cont2-bf-Fdes (page 99)*
*lemma: GoodIterator-F-spec2-cont (page 98)*
*lemma: GoodIterator-F-spec-cont-bf (page 97)*
*lemma: GoodIterator-F-spec-cont3-bf-Fdes (page 100)*
*lemma: GoodIterator-F-bf-cont-bf-des (page 110)*

**definition** *ifcomp* :: $('a \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'a)$ $(- \cdot - 120)$ **where**
*ifcomp F2 F1* $\equiv (\lambda C.$ *if F1 C=C then C else* $(F2 \circ F1)(C))$

**theorem** *GoodIterator-composeCond*:
*GoodIterator F1 Qinp Qterm1 Qinter1*
$\Longrightarrow$ *GoodIterator F2 Qinter2 Qterm2 Qout2*
$\Longrightarrow$ $(\bigwedge Q.$ *Qinter1 Q* $\longrightarrow$ *Qinter2 Q* $)$
$\Longrightarrow$ $(\bigwedge Q.$ *Qinp Q* $\longrightarrow$ *Qterm2 Q*$)$
$\Longrightarrow$ *GoodIterator* $(F2 \cdot F1)$ *Qinp* $(\lambda C.$ *if F1 C=C then*
*Qterm1 C* $\wedge$ *Qinp C else Qterm1 C* $\wedge$ *Qterm2 C*$)$ $((Qinp \cap Qinter1 \cap Qterm1) \cup Qout2)$
⟨*proof*⟩

*depends on:*
*lemma: le-trans (page 90)*

**definition** *GoodIteratorX* :: $('a$::*complete-lattice* $\Rightarrow 'a) \Rightarrow ('a \Rightarrow bool) \Rightarrow bool$ **where**
*GoodIteratorX F Qterm* $\equiv$ (
$(\forall X.\ F\ X \leq X)$
$\wedge\ (\forall X.\ Qterm\ X \longleftrightarrow (F\ X = X))$
$\wedge\ (\forall X\ Y.\ X \leq Y \longrightarrow F\ X \leq F\ Y)$
$\wedge\ (\forall X\ Y.\ F\ X < X \longrightarrow Y < X \longrightarrow F\ Y = Y \longrightarrow Y \leq F\ X))$

**lemma** *GoodIteratorX-vs-GoodIterator*:
*GoodIteratorX F Qterm*= *GoodIterator F UNIV Qterm UNIV*
⟨*proof*⟩

*required by:*
*theorem: GoodIteratorX-F-spec-cont2-bf-Fdes (page 99)*
*theorem: GoodIteratorX-F-spec-cont-bf-Fdes (page 98)*
*theorem: GoodIteratorX-F-spec-cont3-bf-Fdes (page 100)*

**lemma** *Supremum-to-gfp*:
*GoodIteratorX F Qterm*
$\Longrightarrow Sup\{X.\ Qterm\ X\} = gfp\ F$
⟨*proof*⟩

*required by:*
*theorem: Supremum-to-gfp-initialDec (page 106)*
*theorem: computation-via-compute2 (page 109)*

**lemma** *gfp-invariant*:
*GoodIteratorX F Q*
$\Longrightarrow$ *gfp* ($\lambda X.\ F$ (*inf X* (*F Y*))) = *gfp* ($\lambda X.\ F$ (*inf X Y*))
⟨*proof*⟩

*depends on:*
*lemma: le-trans (page 90)*

*required by:*
*corollary: decomposition-sound-and-least-restrictive (page 98)*

### *A.3.3 Complete Lattice of DES*

**datatype** *'a DES* =
*DES 'a list set 'a list set*

**definition** *des-langUM* :: *'a DES* ⇒ *'a list set* **where**
*des-langUM D* = (*case D of DES A B* ⇒ *A*)

**definition** *des-langM* :: *'a DES* ⇒ *'a list set* **where**
*des-langM D* = (*case D of DES A B* ⇒ *B*)

**definition** *IsDES* :: *'a DES* ⇒ *bool* **where**
*IsDES D* ≡ (*des-langM D* ⊆ *des-langUM D*
∧ *prec* (*des-langUM D*) = *des-langUM D*)

**definition** *lesseqDES* :: *'a DES* ⇒ *'a DES* ⇒ *bool* **where**
*lesseqDES D1 D2* ≡ (
*des-langUM D1* ⊆ *des-langUM D2*
∧ *des-langM D1* ⊆ *des-langM D2*)

**definition** *lessDES* :: *'a DES* ⇒ *'a DES* ⇒ *bool* **where**
*lessDES D1 D2* ≡ *lesseqDES D1 D2* ∧ *D1*≠*D2*

**definition** *infDES* :: *'a DES* ⇒ *'a DES* ⇒ *'a DES* **where**
*infDES D1 D2* ≡ *DES*
(*des-langUM D1* ∩ *des-langUM D2*)
(*des-langM D1* ∩ *des-langM D2*)

**definition** *supDES* :: *'a DES* ⇒ *'a DES* ⇒ *'a DES* **where**
*supDES D1 D2* ≡ *DES* (*des-langUM D1* ∪ *des-langUM D2*) (*des-langM D1* ∪ *des-langM D2*)

**definition** *botDES* :: *'a DES* **where**
*botDES* ≡ *DES* {} {}

**definition** *topDES* :: *'a DES* **where**
*topDES* ≡ *DES UNIV UNIV*

**definition** *SupDES* :: *'a DES set* ⇒ *'a DES* **where**
*SupDES A* ≡ *DES* (⋃ (*des-langUM* ' *A*)) (⋃ (*des-langM* ' *A*))

**definition** *InfDES* :: *'a DES set* ⇒ *'a DES* **where**
*InfDES A* ≡ *DES* (⋂ (*des-langUM* ‘*A*)) (⋂ (*des-langM* ‘*A*))

**instantiation** *DES* :: (*type*)*complete-lattice*
**begin**

**print-context**

**definition**
*bot-DES-ext-def*: *bot* = *botDES*

**definition**
*inf-DES-ext-def*: *inf D1 D2* = *infDES D1 D2*

**definition**
*sup-DES-ext-def*: *sup D1 D2* = *supDES D1 D2*

**definition**
*top-DES-ext-def*: *top* = *topDES*

**definition**
*less-eq-DES-ext-def*: *less-eq D1 D2* = *lesseqDES D1 D2*

**definition**
*less-DES-ext-def*: *less A B* = *lessDES A B*

**definition**
*Sup-DES-ext-def*: *Sup A* = *SupDES A*

**definition**
*Inf-DES-ext-def*: *Inf A* = *InfDES A*

**instance**
⟨*proof*⟩
**end**

**lemma** *infDES-preserves-IsDES*:
*IsDES A*
⟹ *IsDES B*
⟹ *IsDES* (*infDES A B*)
⟨*proof*⟩

*required by:*
*lemma: computeN-Fbf-Fcont3-Fbf-Fdes-eq-Fbf-ifcomp-Fcont3 (page 112)*
*theorem: Supremum-in-DESSIMMSol (page 112)*

### *A.3.4 Synthesis Basics*

**definition** *ContW* :: *'a list* ⇒ *'a list set* ⇒ *'a list set* ⇒ *'a list set* ⇒ *bool* **where**
*ContW w U Pum Cum* ≡ (∀ *u*∈*U*. *w*@*u*∈*Pum* ⟶ *w*@*u*∈*Cum*)

**definition** *Cont* :: *'a list set* ⇒ *'a list set* ⇒ *'a list set* ⇒ *'a list set* ⇒ *bool* **where**

*Cont A U Pum Cum* ≡ (∀ *w'*∈*A*. *ContW w' U Pum Cum*)

### A.3.5 Operation Fdes

**definition** *Fdes* :: *'a DES* ⇒ *'a DES* **where**
*Fdes C* = (
*let SUPprec* = *Sup*{*A*. *A* ⊆ *des-langUM C* ∧ *A*=*prec A*} *in*
*DES SUPprec* (*SUPprec* ∩ (*des-langM C*)))

**lemma** *Fdes-makes-DES*:
*IsDES* (*Fdes D*)
⟨*proof*⟩

*required by:*
*theorem: GoodIterator-Fdes (page 94)*

**lemma** *Fdes-is-decreasing*:
*Fdes D* ≤ *D*
⟨*proof*⟩

*required by:*
*theorem: GoodIterator-Fdes (page 94)*

**theorem** *GoodIterator-Fdes*:
*GoodIterator Fdes UNIV IsDES IsDES*
⟨*proof*⟩

*depends on:*
*lemma: Fdes-is-decreasing (page 94)*
*lemma: lem1 (page 89)*
*lemma: Fdes-makes-DES (page 94)*

*required by:*
*lemma: GoodIterator-F-spec-cont-bf-Fdes (page 97)*
*lemma: computeN-Fbf-Fcont3-Fbf-Fdes-eq-Fbf-ifcomp-Fcont3 (page 112)*
*lemma: GoodIterator-F-spec-cont2-bf-Fdes (page 99)*
*lemma: GoodIterator-F-spec-cont3-bf-Fdes (page 100)*
*lemma: GoodIterator-F-bf-cont-bf-des (page 110)*
*theorem: compute-DESSIMMSol (page 113)*

### A.3.6 Operation Fbf

**definition** *Fbf* :: *'a DES* ⇒ *'a DES* **where**
*Fbf C* = *DES* (*prec* (*des-langM C*)) (*des-langM C*)

**definition** *PropBF* :: *'a DES* ⇒ *bool* **where**
*PropBF C* = (*des-langUM C* ⊆ *prec* (*des-langM C*))

**theorem** *GoodIterator-Fbf*:
*GoodIterator Fbf IsDES* (*IsDES* ∩ *PropBF*) (*IsDES* ∩ *PropBF*)
⟨*proof*⟩

*required by:*
*lemma: computeN-Fbf-Fcont3-Fbf-Fdes-eq-Fbf-ifcomp-Fcont3 (page 112)*
*lemma: GoodIterator-F-spec2-cont-bf (page 99)*
*lemma: GoodIterator-F-spec-cont3-bf (page 100)*

*lemma: GoodIterator-F-bf-spec (page 110)*
*lemma: GoodIterator-F-spec-cont-bf (page 97)*
*lemma: GoodIterator-F-bf-cont-bf-des (page 110)*
*theorem: Supremum-in-DESSIMMSol (page 112)*
*theorem: compute-DESSIMMSol (page 113)*

### A.3.7 Operation Fcont

**definition** *Fcont* :: *'a DES ⇒ 'a set ⇒ 'a list set ⇒ 'a list set ⇒ 'a DES* **where**
*Fcont C SigmaUC Pum Cum = (let A = {w. w∈des-langUM C ∧ Cont (prec {w}) (STAR SigmaUC) Pum Cum} in*
*DES A (A ∩ (des-langM C)))*

**definition** *DESCont* :: *'a DES ⇒ 'a DES ⇒ 'a set ⇒ bool* **where**
*DESCont C P SigmaUC = ((sconc(des-langUM C)(SigmaUC))∩(des-langUM P)⊆des-langUM C)*

**lemma** *lem3*:
*IsDES (DES PUM PM)*
*⟹ IsDES (DES LUM LM)*
*⟹ sconc (LUM) SigmaUC ∩ PUM ⊆ LUM*
*⟹ x ∈ LUM*
*⟹ w' ∈ prec {x}*
*⟹ set u ⊆ SigmaUC*
*⟹ w' @ u ∈ PUM*
*⟹ w' @ u ∈ LUM*
⟨*proof*⟩

*required by:*
*theorem: GoodIterator-Fcont2 (page 95)*
*theorem: GoodIterator-Fcont3 (page 96)*
*theorem: GoodIterator-Fcont (page 95)*

**theorem** *GoodIterator-Fcont*:
*IsDES P*
*⟹ GoodIterator (λC. Fcont C SigmaUC (des-langUM P) (des-langUM C)) IsDES (λC. IsDES C ∧ DESCont C P SigmaUC) (λC. IsDES C ∧ DESCont C P SigmaUC)*
⟨*proof*⟩

*depends on:*
*lemma: lem3 (page 95)*
*lemma: lem2 (page 89)*

*required by:*
*lemma: GoodIterator-F-spec-cont (page 97)*

### A.3.8 Operation Fcont2

**definition** *Fcont2* :: *'a DES ⇒ 'a set ⇒ 'a list set ⇒ 'a list set ⇒ 'a DES* **where**
*Fcont2 C SigmaUC Pum Cum = (let A = {w. w∈des-langUM C ∧ Cont (prec {w}) (convAL SigmaUC) Pum Cum} in*
*DES A (A ∩ (des-langM C)))*

**theorem** *GoodIterator-Fcont2*:
*IsDES P*

$\Longrightarrow$ *GoodIterator*
(λ*C*. *Fcont2 C SigmaUC* (*des-langUM P*) (*des-langUM C*))
*IsDES*
(λ*C*. *IsDES C* ∧ *DESCont C P SigmaUC*)
*IsDES*
⟨*proof*⟩

*depends on:*
*lemma: lem3 (page 95)*

*required by:*
*lemma: GoodIterator-F-spec2-cont (page 98)*

### *A.3.9 Operation Fcont3*

**definition** *Fcont3* :: *'a DES* ⇒ *'a set* ⇒ *'a list set* ⇒ *'a list set* ⇒ *'a DES* **where**
*Fcont3 C SigmaUC Pum Cum* = (*let*
*B* = {*w*. (*w*∈*des-langM C*)
∧ (*Cont* (*prec* {*w*}) (*convAL SigmaUC*) *Pum Cum*)};
*A* = {*w*. (*w*∈*des-langUM C*)
∧ (*Cont* ((*prec* {*w*})−{*w*}) (*convAL SigmaUC*) *Pum Cum*)
∧ ((¬(*ContW w* (*convAL SigmaUC*) *Pum Cum*))⟶((*w*∉ *prec B*)))} *in*
*DES A B*)

**theorem** *GoodIterator-Fcont3*:
*IsDES P*
$\Longrightarrow$ *GoodIterator* (λ*C*. *Fcont3 C SigmaUC* (*des-langUM P*) (*des-langUM C*)) (*IsDES*∩*PropBF*) (λ*C*. *IsDES C* ∧ *DESCont C P SigmaUC*) *IsDES*
⟨*proof*⟩

*depends on:*
*lemma: lem3 (page 95)*

*required by:*
*lemma: GoodIterator-F-spec-cont3 (page 99)*
*lemma: GoodIterator-F-bf-cont-bf-des (page 110)*

### *A.3.10 Operation Fspec*

**definition** *Fspec* :: *'a DES* ⇒ *'a DES* ⇒ *'a DES* **where**
*Fspec C S* = *DES* (*des-langUM S* ∩ (*des-langUM C*)) (*des-langM S* ∩ (*des-langM C*))

**definition** *PropSpec* :: *'a DES* ⇒ *'a DES* ⇒ *bool* **where**
*PropSpec C S* =
(*des-langUM C* ⊆ *des-langUM S*
∧ *des-langM C* ⊆ *des-langM S*)

**theorem** *GoodIterator-Fspec*:
*IsDES P*
$\Longrightarrow$ *IsDES S*
$\Longrightarrow$ *GoodIterator* (λ*C*. *Fspec C S*) *IsDES* (λ*C*. *IsDES C* ∧ *PropSpec C S*) (λ*C*. *IsDES C* ∧ *PropSpec C S*)
⟨*proof*⟩

*depends on:*
*lemma: prec-closed-sets-closed-under-intersection (page 89)*

*required by:*
*lemma: GoodIterator-F-spec-cont (page 97)*
*lemma: GoodIterator-F-spec-cont3 (page 99)*
*lemma: GoodIterator-F-bf-spec (page 110)*
*lemma: GoodIterator-F-spec2-cont (page 98)*
*theorem: Supremum-in-DESSIMMSol (page 112)*

### A.3.11 Composition of Fspec, Fcont, Fbf, and Fdes

**lemma** *GoodIterator-F-spec-cont*:
$IsDES\ P$
$\Longrightarrow IsDES\ S$
$\Longrightarrow GoodIterator$
$((\lambda C.\ Fspec\ C\ S) \circ (\lambda C.\ Fcont\ C\ SigmaUC\ (des\text{-}langUM\ P)\ (des\text{-}langUM\ C)))$
$IsDES$
$(\lambda X.\ (\lambda C.\ IsDES\ C \wedge DESCont\ C\ P\ SigmaUC)\ X \wedge (\lambda C.\ IsDES\ C \wedge PropSpec\ C\ S)\ X)$
$(\lambda C.\ IsDES\ C \wedge PropSpec\ C\ S)$
⟨*proof*⟩

*depends on:*
*theorem: GoodIterator-compose (page 90)*
*theorem: GoodIterator-Fcont (page 95)*
*theorem: GoodIterator-Fspec (page 96)*

*required by:*
*lemma: GoodIterator-F-spec-cont-bf (page 97)*

**lemma** *GoodIterator-F-spec-cont-bf*:
$IsDES\ P$
$\Longrightarrow IsDES\ S$
$\Longrightarrow GoodIterator$
$(((\lambda C.\ Fspec\ C\ S) \circ (\lambda C.\ Fcont\ C\ SigmaUC\ (des\text{-}langUM\ P)\ (des\text{-}langUM\ C))) \circ Fbf)$
$IsDES$
$(\lambda X.\ ((IsDES \cap PropBF)\ X) \wedge ((\lambda C.\ IsDES\ C \wedge DESCont\ C\ P\ SigmaUC)\ X \wedge (\lambda C.\ IsDES\ C \wedge PropSpec\ C\ S)\ X))$
$(\lambda C.\ IsDES\ C \wedge PropSpec\ C\ S)$
⟨*proof*⟩

*depends on:*
*theorem: GoodIterator-compose (page 90)*
*lemma: GoodIterator-F-spec-cont (page 97)*
*theorem: GoodIterator-Fbf (page 94)*

*required by:*
*lemma: GoodIterator-F-spec-cont-bf-Fdes (page 97)*

**lemma** *GoodIterator-F-spec-cont-bf-Fdes*:
$IsDES\ P$
$\Longrightarrow IsDES\ S$
$\Longrightarrow GoodIterator$
$(((\lambda C.\ Fspec\ C\ S) \circ (\lambda C.\ Fcont\ C\ SigmaUC\ (des\text{-}langUM\ P)\ (des\text{-}langUM\ C))) \circ Fbf \circ Fdes)$
$UNIV$
$(\lambda X.\ (IsDES\ X) \wedge (((IsDES \cap PropBF)\ X) \wedge ((\lambda C.\ IsDES\ C \wedge DESCont\ C\ P\ SigmaUC)\ X \wedge (\lambda C.\ IsDES\ C \wedge PropSpec\ C\ S)\ X)))$
$(\lambda C.\ IsDES\ C \wedge PropSpec\ C\ S)$

⟨*proof*⟩

*depends on:*
*theorem: GoodIterator-compose (page 90)*
*lemma: GoodIterator-F-spec-cont-bf (page 97)*
*theorem: GoodIterator-Fdes (page 94)*

*required by:*
*theorem: GoodIteratorX-F-spec-cont-bf-Fdes (page 98)*

**theorem** *GoodIteratorX-F-spec-cont-bf-Fdes*:
*IsDES P*
$\Longrightarrow$ *IsDES S*
$\Longrightarrow$ *GoodIteratorX*
$(((\lambda C.\ Fspec\ C\ S) \circ (\lambda C.\ Fcont\ C\ SigmaUC\ (des\text{-}langUM\ P)\ (des\text{-}langUM\ C))) \circ Fbf \circ Fdes)$
$(\lambda X.\ (IsDES\ X) \wedge (((IsDES \cap PropBF)\ X) \wedge ((\lambda C.\ IsDES\ C \wedge DESCont\ C\ P\ SigmaUC)\ X \wedge (\lambda C.\ IsDES\ C \wedge PropSpec\ C\ S)\ X)))$
⟨*proof*⟩

*depends on:*
*lemma: GoodIteratorX-vs-GoodIterator (page 91)*
*lemma: GoodIterator-weakening (page 90)*
*lemma: GoodIterator-F-spec-cont-bf-Fdes (page 97)*

*required by:*
*theorem: computation-of-least-restrictive-maximal-solution-via-compute-for-cont (page 107)*
*corollary: decomposition-sound-and-least-restrictive (page 98)*

**corollary** *decomposition-sound-and-least-restrictive*:
*IsDES P*
$\Longrightarrow$ *IsDES S*
$\Longrightarrow F = (((\lambda C.\ Fspec\ C\ S) \circ (\lambda C.\ Fcont\ C\ SigmaUC\ (des\text{-}langUM\ P)\ (des\text{-}langUM\ C))) \circ Fbf \circ Fdes)$
$\Longrightarrow gfp\ (\lambda X.\ F\ (inf\ X\ (F\ Y))) = gfp\ (\lambda X.\ F\ (inf\ X\ Y))$
⟨*proof*⟩

*depends on:*
*lemma: gfp-invariant (page 92)*
*theorem: GoodIteratorX-F-spec-cont-bf-Fdes (page 98)*

### *A.3.12 Composition of Fspec, Fcont2, Fbf, and des*

**lemma** *GoodIterator-F-spec2-cont*:
*IsDES P*
$\Longrightarrow$ *IsDES S*
$\Longrightarrow$ *GoodIterator*
$((\lambda C.\ Fspec\ C\ S) \circ (\lambda C.\ Fcont2\ C\ SigmaUC\ (des\text{-}langUM\ P)\ (des\text{-}langUM\ C)))$
*IsDES*
$(\lambda X.\ (\lambda C.\ IsDES\ C \wedge DESCont\ C\ P\ SigmaUC)\ X \wedge (\lambda C.\ IsDES\ C \wedge PropSpec\ C\ S)\ X)$
$(\lambda C.\ IsDES\ C \wedge PropSpec\ C\ S)$
⟨*proof*⟩

*depends on:*
*theorem: GoodIterator-compose (page 90)*
*theorem: GoodIterator-Fcont2 (page 95)*
*theorem: GoodIterator-Fspec (page 96)*

*required by:*
*lemma: GoodIterator-F-spec2-cont-bf (page 99)*

**lemma** *GoodIterator-F-spec2-cont-bf*:
*IsDES P*
$\Longrightarrow$ *IsDES S*
$\Longrightarrow$ *GoodIterator*
$(((\lambda C.\ Fspec\ C\ S) \circ (\lambda C.\ Fcont2\ C\ SigmaUC\ (des\text{-}langUM\ P)\ (des\text{-}langUM\ C))) \circ Fbf)$
*IsDES*
$(\lambda X.\ ((IsDES \cap PropBF)\ X) \wedge ((\lambda C.\ IsDES\ C \wedge DESCont\ C\ P\ SigmaUC)\ X \wedge (\lambda C.\ IsDES\ C \wedge PropSpec\ C\ S)\ X))$
$(\lambda C.\ IsDES\ C \wedge PropSpec\ C\ S)$
⟨*proof*⟩

*depends on:*
*theorem: GoodIterator-compose (page 90)*
*lemma: GoodIterator-F-spec2-cont (page 98)*
*theorem: GoodIterator-Fbf (page 94)*

*required by:*
*lemma: GoodIterator-F-spec-cont2-bf-Fdes (page 99)*

**lemma** *GoodIterator-F-spec-cont2-bf-Fdes*:
*IsDES P*
$\Longrightarrow$ *IsDES S*
$\Longrightarrow$ *GoodIterator*
$(((\lambda C.\ Fspec\ C\ S) \circ (\lambda C.\ Fcont2\ C\ SigmaUC\ (des\text{-}langUM\ P)\ (des\text{-}langUM\ C))) \circ Fbf \circ Fdes)$
*UNIV*
$(\lambda X.\ (IsDES\ X) \wedge (((IsDES \cap PropBF)\ X) \wedge ((\lambda C.\ IsDES\ C \wedge DESCont\ C\ P\ SigmaUC)\ X \wedge (\lambda C.\ IsDES\ C \wedge PropSpec\ C\ S)\ X)))$
$(\lambda C.\ IsDES\ C \wedge PropSpec\ C\ S)$
⟨*proof*⟩

*depends on:*
*theorem: GoodIterator-compose (page 90)*
*lemma: GoodIterator-F-spec2-cont-bf (page 99)*
*theorem: GoodIterator-Fdes (page 94)*

*required by:*
*theorem: GoodIteratorX-F-spec-cont2-bf-Fdes (page 99)*

**theorem** *GoodIteratorX-F-spec-cont2-bf-Fdes*:
*IsDES P*
$\Longrightarrow$ *IsDES S*
$\Longrightarrow$ *GoodIteratorX*
$(((\lambda C.\ Fspec\ C\ S) \circ (\lambda C.\ Fcont2\ C\ SigmaUC\ (des\text{-}langUM\ P)\ (des\text{-}langUM\ C))) \circ Fbf \circ Fdes)$
$(\lambda X.\ (IsDES\ X) \wedge (((IsDES \cap PropBF)\ X) \wedge ((\lambda C.\ IsDES\ C \wedge DESCont\ C\ P\ SigmaUC)\ X \wedge (\lambda C.\ IsDES\ C \wedge PropSpec\ C\ S)\ X)))$
⟨*proof*⟩

*depends on:*
*lemma: GoodIteratorX-vs-GoodIterator (page 91)*
*lemma: GoodIterator-weakening (page 90)*
*lemma: GoodIterator-F-spec-cont2-bf-Fdes (page 99)*

*required by:*
*theorem: computation-of-least-restrictive-maximal-solution-via-compute-for-cont2 (page 107)*

### *A.3.13 Composition of Fspec, Fcont2, Fbf, and Fdes*

**lemma** *GoodIterator-F-spec-cont3*:

*IsDES P*
$\Longrightarrow$ *IsDES S*
$\Longrightarrow$ *GoodIterator*
((λC. *Fspec C S*)∘(λC. *Fcont3 C SigmaUC* (*des-langUM P*) (*des-langUM C*)))
(*IsDES∩PropBF*)
(λX. (λC. *IsDES C* ∧ *DESCont C P SigmaUC*) *X* ∧ (λC. *IsDES C* ∧ *PropSpec C S*) *X*)
(λC. *IsDES C* ∧ *PropSpec C S*)
⟨*proof*⟩

*depends on:*
*theorem: GoodIterator-compose (page 90)*
*theorem: GoodIterator-Fcont3 (page 96)*
*theorem: GoodIterator-Fspec (page 96)*

*required by:*
*lemma: GoodIterator-F-spec-cont3-bf (page 100)*

**lemma** *GoodIterator-F-spec-cont3-bf*:
*IsDES P*
$\Longrightarrow$ *IsDES S*
$\Longrightarrow$ *GoodIterator*
(((λC. *Fspec C S*)∘(λC. *Fcont3 C SigmaUC* (*des-langUM P*) (*des-langUM C*)))∘*Fbf*)
*IsDES*
(λX. ((*IsDES∩PropBF*) *X*) ∧ ((λC. *IsDES C* ∧ *DESCont C P SigmaUC*) *X* ∧ (λC. *IsDES C* ∧ *PropSpec C S*) *X*))
(λC. *IsDES C* ∧ *PropSpec C S*)
⟨*proof*⟩

*depends on:*
*theorem: GoodIterator-compose (page 90)*
*lemma: GoodIterator-F-spec-cont3 (page 99)*
*theorem: GoodIterator-Fbf (page 94)*

*required by:*
*lemma: GoodIterator-F-spec-cont3-bf-Fdes (page 100)*

**lemma** *GoodIterator-F-spec-cont3-bf-Fdes*:
*IsDES P*
$\Longrightarrow$ *IsDES S*
$\Longrightarrow$ *GoodIterator*
(((λC. *Fspec C S*)∘(λC. *Fcont3 C SigmaUC* (*des-langUM P*) (*des-langUM C*)))∘*Fbf*∘*Fdes*)
*UNIV*
(λX. (*IsDES X*) ∧ (((*IsDES∩PropBF*) *X*) ∧ ((λC. *IsDES C* ∧ *DESCont C P SigmaUC*) *X* ∧ (λC. *IsDES C* ∧ *PropSpec C S*) *X*)))
(λC. *IsDES C* ∧ *PropSpec C S*)
⟨*proof*⟩

*depends on:*
*theorem: GoodIterator-compose (page 90)*
*lemma: GoodIterator-F-spec-cont3-bf (page 100)*
*theorem: GoodIterator-Fdes (page 94)*

*required by:*
*theorem: GoodIteratorX-F-spec-cont3-bf-Fdes (page 100)*

**theorem** *GoodIteratorX-F-spec-cont3-bf-Fdes*:
*IsDES P*
$\Longrightarrow$ *IsDES S*

$\Longrightarrow$ *GoodIteratorX*
(((λ*C*. *Fspec C S*)∘(λ*C*. *Fcont3 C SigmaUC* (*des-langUM P*) (*des-langUM C*)))∘*Fbf*∘*Fdes*)
(λ*X*. (*IsDES X*) ∧ (((*IsDES*∩*PropBF*) *X*) ∧ ((λ*C*. *IsDES C* ∧ *DESCont C P SigmaUC*) *X* ∧ (λ*C*. *IsDES C* ∧ *PropSpec C S*) *X*)))
⟨*proof*⟩

*depends on:*
*lemma: GoodIteratorX-vs-GoodIterator (page 91)*
*lemma: GoodIterator-weakening (page 90)*
*lemma: GoodIterator-F-spec-cont3-bf-Fdes (page 100)*

*required by:*
*theorem: computation-of-least-restrictive-maximal-solution-via-compute-for-cont3 (page 107)*

### A.3.14 Function Computation of Fixedpoint

**function** (*domintros*) *compute* :: ('*a DES* ⇒ '*a DES*) ⇒ '*a DES* ⇒ '*a DES* **where**
*compute F D* = (*if*(*D*=*F D*)*then D else compute F* (*F D*))
⟨*proof*⟩

**primrec** *computeN* :: *nat* ⇒ ('*a DES* ⇒ '*a DES*) ⇒ '*a DES* ⇒ '*a DES* **where**
*computeN o F D* = *D*
| *computeN* (*Suc n*) *F D* = *computeN n F* (*F D*)

**lemma** *computeN-iterate-vs-pre-apply*:
*computeN n F* (*F A*) = *computeN* (*Suc n*) *F A*
⟨*proof*⟩

*required by:*
*lemma: computation-via-computeN-initial-hlp (page 104)*
*lemma: computeN-reaches-Qinter (page 105)*
*lemma: computation-via-computeN-initialDec-hlp (page 105)*
*lemma: computation-via-computeN-hlp (page 102)*

**lemma** *computeN-iterate-vs-post-apply-hlp*:
∀*A*. *F* (*computeN n F A*) = *computeN* (*Suc n*) *F A*
⟨*proof*⟩

*required by:*
*lemma: computation-via-computeN-initial-hlp (page 104)*
*lemma: computeN-iterate-vs-post-apply (page 101)*
*lemma: computeN-preserve-prop (page 112)*
*lemma: computation-via-computeN-initialDec-hlp (page 105)*

**lemma** *computeN-iterate-vs-post-apply*:
*F* (*computeN n F A*) = *computeN* (*Suc n*) *F A*
⟨*proof*⟩

*depends on:*
*lemma: computeN-iterate-vs-post-apply-hlp (page 101)*

*required by:*
*theorem: EqualCompute-initial (page 108)*
*lemma: computeterm-by-computeN-hlp (page 108)*
*lemma: computation-via-computeN-initial-hlp (page 104)*
*lemma: computeN-smaller-than-input (page 104)*
*lemma: computeN-reaches-Qinter (page 105)*
*lemma: computation-via-computeN-initialDec-hlp (page 105)*

*lemma: computeN-invariant-after-fixed-point (page 108)*
*lemma: computation-via-computeN-hlp (page 102)*

**lemma** *compute-by-computeN*:
$compute\text{-}dom\ (F,A)$
$\Longrightarrow \exists n.\ compute\ F\ A = computeN\ n\ F\ A \wedge F\ (computeN\ n\ F\ A) = computeN\ n\ F\ A$
⟨*proof*⟩

*required by:*
*theorem: computation-via-compute-initial (page 104)*
*theorem: equal-computation-via-compute (page 108)*
*theorem: computation-via-compute (page 103)*
*theorem: computation-via-compute2 (page 109)*
*theorem: Supremum-to-gfp-initialTranslate (page 109)*
*theorem: computation-via-compute-initialDec (page 106)*

**lemma** *computation-via-computeN-hlp1*:
$GoodIteratorX\ F\ Q$
$\Longrightarrow F\ A \neq A$
$\Longrightarrow F\ (F\ A) \neq F\ A$
$\Longrightarrow gfp\ F < A$
$\Longrightarrow gfp\ F < F\ A$
⟨*proof*⟩

*required by:*
*lemma: computation-via-computeN-hlp (page 102)*

**lemma** *computation-via-computeN-hlp2*:
$GoodIteratorX\ F\ Q$
$\Longrightarrow F\ A \neq A$
$\Longrightarrow F\ (F\ A) = F\ A$
$\Longrightarrow gfp\ F < A$
$\Longrightarrow F\ A = gfp\ F$
⟨*proof*⟩

*required by:*
*lemma: computation-via-computeN-hlp (page 102)*

**lemma** *computation-via-computeN-hlp*:
$GoodIteratorX\ F\ Q$
$\Longrightarrow$
$(F\ (computeN\ n\ F\ top) = computeN\ n\ F\ top \longrightarrow computeN\ n\ F\ top = gfp\ F)$
$\wedge\ (F\ (computeN\ n\ F\ top) \neq computeN\ n\ F\ top \longrightarrow computeN\ n\ F\ top > gfp\ F)$
⟨*proof*⟩

*depends on:*
*lemma: computeN-iterate-vs-pre-apply (page 101)*
*lemma: computeN-iterate-vs-post-apply (page 101)*
*lemma: computation-via-computeN-hlp1 (page 102)*
*lemma: computation-via-computeN-hlp2 (page 102)*

*required by:*
*lemma: computation-via-computeN (page 102)*

**lemma** *computation-via-computeN*:
$GoodIteratorX\ F\ Q$
$\Longrightarrow F\ (computeN\ n\ F\ top) = computeN\ n\ F\ top$

$\Longrightarrow$ *computeN n F top* = *gfp F*
⟨*proof*⟩

*depends on:*
*lemma: computation-via-computeN-hlp (page 102)*

*required by:*
*theorem: computation-via-compute (page 103)*

**theorem** *computation-via-compute*:
*GoodIteratorX F Q*
$\Longrightarrow$ *compute-dom* (*F*,*top*)
$\Longrightarrow$ *compute F top* = (*gfp F*)
⟨*proof*⟩

*depends on:*
*lemma: compute-by-computeN (page 102)*
*lemma: computation-via-computeN (page 102)*

*required by:*
*theorem: computation-via-compute2 (page 109)*
*theorem: computation-of-least-restrictive-maximal-solution-via-compute-for-cont3 (page 107)*
*theorem: computation-of-least-restrictive-maximal-solution-via-compute-for-cont2 (page 107)*
*theorem: computation-of-least-restrictive-maximal-solution-via-compute-for-cont (page 107)*

### *A.3.15 Validity of Initialised Fixedpoint*

**lemma** *gfp-not-fixed-point*:
$F\ X \neq X$
$\Longrightarrow$ *mono F*
$\Longrightarrow$ *gfp F* $\neq X$
⟨*proof*⟩

*required by:*
*lemma: computation-via-computeN-initial-hlp (page 104)*
*lemma: computation-via-computeN-initialDec-hlp (page 105)*

**lemma** *gfp-fixed-point*:
$F\ X = X$
$\Longrightarrow X \leq$ *gfp F*
⟨*proof*⟩

*required by:*
*lemma: computation-via-computeN-initial-hlp (page 104)*
*lemma: computation-via-computeN-initialDec-hlp (page 105)*

**lemma** *gfp-mixed-fixed-point*:
$(\bigwedge X.\ F\ X \leq X)$
$\Longrightarrow (\bigwedge X\ Y.\ X \leq Y \Longrightarrow F\ X \leq F\ Y)$
$\Longrightarrow F$ (*gfp* (%*X*. *F* (*inf X S*))) = *gfp* (%*X*. *F* (*inf X S*))
⟨*proof*⟩

*depends on:*
*lemma: le-trans (page 90)*

*required by:*
*lemma: computation-via-computeN-initial-hlp (page 104)*
*lemma: computation-via-computeN-initialDec-hlp (page 105)*
*theorem: Supremum-to-gfp-initialDec-instantiate (page 107)*

**lemma** *computeN-smaller-than-input*:
$(\bigwedge C.\ F\ C \leq C)$
$\Longrightarrow$ *computeN n F S* $\leq S$
⟨*proof*⟩

*depends on:*
*lemma: computeN-iterate-vs-post-apply (page 101)*

*required by:*
*lemma: computation-via-computeN-initial-hlp (page 104)*
*lemma: computation-via-computeN-initialDec-hlp (page 105)*

**lemma** *computation-via-computeN-initial-hlp*:
*GoodIterator F UNIV Qterm Qout*
$\Longrightarrow$
$(F\ (\mathit{computeN}\ n\ F\ S) = \mathit{computeN}\ n\ F\ S \longrightarrow \mathit{computeN}\ n\ F\ S = \mathit{gfp}\ (\%X.\ F\ (\mathit{inf}\ X\ S)))$
$\wedge\ (F\ (\mathit{computeN}\ n\ F\ S) \neq \mathit{computeN}\ n\ F\ S \longrightarrow \mathit{computeN}\ n\ F\ S > \mathit{gfp}\ (\%X.\ F\ (\mathit{inf}\ X\ S)))$
⟨*proof*⟩

*depends on:*
*lemma: le-trans (page 90)*
*lemma: gfp-not-fixed-point (page 103)*
*lemma: computeN-iterate-vs-post-apply (page 101)*
*lemma: gfp-fixed-point (page 103)*
*lemma: computeN-smaller-than-input (page 104)*
*lemma: computeN-iterate-vs-pre-apply (page 101)*
*lemma: gfp-mixed-fixed-point (page 103)*
*lemma: computeN-iterate-vs-post-apply-hlp (page 101)*

*required by:*
*lemma: computation-via-computeN-initial (page 104)*

**lemma** *computation-via-computeN-initial*:
*GoodIterator F UNIV Qterm Qout*
$\Longrightarrow$ *F (computeN n F S) = computeN n F S*
$\Longrightarrow$ *computeN n F S = gfp (%X. F (inf X S))*
⟨*proof*⟩

*depends on:*
*lemma: computation-via-computeN-initial-hlp (page 104)*

*required by:*
*theorem: computation-via-compute-initial (page 104)*

**theorem** *computation-via-compute-initial*:
*GoodIterator F UNIV Qterm Qout*
$\Longrightarrow$ *compute-dom (F,S)*
$\Longrightarrow$ *compute F S = (gfp (%X. F (inf X S)))*
⟨*proof*⟩

*depends on:*
*lemma: compute-by-computeN (page 102)*
*lemma: computation-via-computeN-initial (page 104)*

*required by:*
*theorem: Supremum-to-gfp-initial (page 104)*

**theorem** *Supremum-to-gfp-initial*:
*GoodIterator F UNIV Qterm Qout*

$\Longrightarrow$ *compute-dom* (*F*,*S*)
$\Longrightarrow$ *compute F S* = *Sup*{*X*. *Qterm X* $\wedge$ *X* $\leq$ *S*}
⟨*proof*⟩

*depends on:*
*theorem: computation-via-compute-initial (page 104)*

*required by:*
*theorem: Supremum-to-gfp-initialTranslate (page 109)*

**lemma** *gfp-Decomposition*:
*GoodIterator G UNIV Qinter Qinter*
$\Longrightarrow$ *GoodIteratorDec F Qinter QFterm QFout*
$\Longrightarrow$ ($\bigwedge$*X*. *QFout* (*F X*))
$\Longrightarrow$ ($\bigwedge$*C*. *QFout C* $\Longrightarrow$ *Qinter C*)
$\Longrightarrow$ *gfp* (*F* $\circ$ *G*) = *gfp* (%*X*. *F* (*inf X* (*G top*)))
⟨*proof*⟩

*depends on:*
*theorem: GoodIterator-compose (page 90)*
*lemma: GoodIteratorDec-to-GoodIterator (page 90)*
*lemma: le-trans (page 90)*

*required by:*
*theorem: Supremum-to-gfp-initialDec (page 106)*

**lemma** *GoodIterator-to-GoodIteratorX*:
*GoodIterator F UNIV Qterm Qout*
$\Longrightarrow$ *GoodIteratorX F Qterm*
⟨*proof*⟩

*required by:*
*theorem: Supremum-to-gfp-initialDec (page 106)*

**lemma** *computeN-reaches-Qinter*:
*Qinter S*
$\Longrightarrow$ *GoodIteratorDec F Qinter QFterm QFout*
$\Longrightarrow$ ($\bigwedge$*C*. *QFout C* $\Longrightarrow$ *Qinter C*)
$\Longrightarrow$ *Qinter* (*computeN n F S*)
⟨*proof*⟩

*depends on:*
*lemma: computeN-iterate-vs-pre-apply (page 101)*
*lemma: computeN-iterate-vs-post-apply (page 101)*

*required by:*
*lemma: computation-via-computeN-initialDec-hlp (page 105)*

**lemma** *computation-via-computeN-initialDec-hlp*:
*Qinter S*
$\Longrightarrow$ *GoodIteratorDec F Qinter QFterm QFout*
$\Longrightarrow$ ($\bigwedge$*C*. *QFout C* $\Longrightarrow$ *Qinter C*)
$\Longrightarrow$ *Qinter* (*gfp* ($\lambda$*X*. *F* (*inf X S*)))
$\Longrightarrow$
(*F* (*computeN n F S*) = *computeN n F S* $\longrightarrow$ *computeN n F S* = *gfp* (%*X*. *F* (*inf X S*)))
$\wedge$ (*F* (*computeN n F S*) $\neq$ *computeN n F S* $\longrightarrow$ *computeN n F S* > *gfp* (%*X*. *F* (*inf X S*)))
⟨*proof*⟩

*depends on:*
*lemma: le-trans (page 90)*
*lemma: gfp-not-fixed-point (page 103)*
*lemma: computeN-iterate-vs-post-apply (page 101)*
*lemma: gfp-fixed-point (page 103)*
*lemma: computeN-smaller-than-input (page 104)*
*lemma: computeN-iterate-vs-pre-apply (page 101)*
*lemma: computeN-reaches-Qinter (page 105)*
*lemma: gfp-mixed-fixed-point (page 103)*
*lemma: computeN-iterate-vs-post-apply-hlp (page 101)*

*required by:*
*lemma: computation-via-computeN-initialDec (page 106)*

**lemma** *computation-via-computeN-initialDec*:
$Qinter\ S$
$\Longrightarrow GoodIteratorDec\ F\ Qinter\ QFterm\ QFout$
$\Longrightarrow (\bigwedge C.\ QFout\ C \Longrightarrow Qinter\ C)$
$\Longrightarrow Qinter\ (gfp\ (\lambda X.\ F\ (inf\ X\ S)))$
$\Longrightarrow F\ (computeN\ n\ F\ S) = computeN\ n\ F\ S$
$\Longrightarrow computeN\ n\ F\ S = gfp\ (\%X.\ F\ (inf\ X\ S))$
⟨*proof*⟩

*depends on:*
*lemma: computation-via-computeN-initialDec-hlp (page 105)*

*required by:*
*theorem: computation-via-compute-initialDec (page 106)*

**theorem** *computation-via-compute-initialDec*:
$Qinter\ S$
$\Longrightarrow GoodIteratorDec\ F\ Qinter\ QFterm\ QFout$
$\Longrightarrow (\bigwedge C.\ QFout\ C \Longrightarrow Qinter\ C)$
$\Longrightarrow Qinter\ (gfp\ (\lambda X.\ F\ (inf\ X\ S)))$
$\Longrightarrow compute\text{-}dom\ (F,S)$
$\Longrightarrow compute\ F\ S = (gfp\ (\%X.\ F\ (inf\ X\ S)))$
⟨*proof*⟩

*depends on:*
*lemma: compute-by-computeN (page 102)*
*lemma: computation-via-computeN-initialDec (page 106)*

*required by:*
*theorem: Supremum-to-gfp-initialDec (page 106)*

**theorem** *Supremum-to-gfp-initialDec*:
$GoodIterator\ G\ UNIV\ Qinter\ Qinter$
$\Longrightarrow GoodIteratorDec\ F\ Qinter\ QFterm\ QFout$
$\Longrightarrow (\bigwedge C.\ QFterm\ C \Longrightarrow Qinter\ C)$
$\Longrightarrow (\bigwedge X.\ QFout\ (F\ X))$
$\Longrightarrow (\bigwedge C.\ QFout\ C \Longrightarrow Qinter\ C)$
$\Longrightarrow Qinter\ (gfp\ (\lambda X.\ F\ (inf\ X\ (G\ top))))$
$\Longrightarrow compute\text{-}dom\ (F,G\ top)$
$\Longrightarrow compute\ F\ (G\ top) = Sup\{X.\ QFterm\ X \wedge X \leq G\ top\}$
⟨*proof*⟩

*depends on:*
*theorem: computation-via-compute-initialDec (page 106)*

*lemma: gfp-Decomposition (page 105)*
*theorem: GoodIterator-compose (page 90)*
*lemma: GoodIteratorDec-to-GoodIterator (page 90)*
*lemma: Supremum-to-gfp (page 91)*
*lemma: GoodIterator-to-GoodIteratorX (page 105)*
*lemma: le-trans (page 90)*

*required by:*
*theorem: Supremum-to-gfp-initialDec-instantiate (page 107)*

**theorem** *Supremum-to-gfp-initialDec-instantiate*:
$GoodIterator\ G\ UNIV\ (\%C.\ IsDES\ C \wedge PropBF\ C)\ (\%C.\ IsDES\ C \wedge PropBF\ C)$
$\Longrightarrow GoodIteratorDec\ F\ (\%C.\ IsDES\ C \wedge PropBF\ C)\ (\%C.\ IsDES\ C \wedge PropBF\ C \wedge DESCont\ C\ P\ SigmaUC)\ (\%C.\ IsDES\ C \wedge PropBF\ C)$
$\Longrightarrow compute\text{-}dom\ (F,G\ top)$
$\Longrightarrow (\bigwedge X.\ IsDES(F\ X))$
$\Longrightarrow (\bigwedge X.\ PropBF(F\ X))$
$\Longrightarrow compute\ F\ (G\ top) = Sup\{X.\ (\%C.\ IsDES\ C \wedge PropBF\ C \wedge DESCont\ C\ P\ SigmaUC)\ X \wedge X \leq G\ top\}$
⟨*proof*⟩

*depends on:*
*theorem: Supremum-to-gfp-initialDec (page 106)*
*lemma: gfp-mixed-fixed-point (page 103)*

### A.3.16 Function Computations for Instances (Classic Approach)

**theorem** *computation-of-least-restrictive-maximal-solution-via-compute-for-cont*:
$IsDES\ P$
$\Longrightarrow IsDES\ S$
$\Longrightarrow F = (((\lambda C.\ Fspec\ C\ S) \circ (\lambda C.\ Fcont\ C\ SigmaUC\ (des\text{-}langUM\ P)\ (des\text{-}langUM\ C))) \circ Fbf \circ Fdes)$
$\Longrightarrow compute\text{-}dom\ (F,top)$
$\Longrightarrow compute\ F\ top = (gfp\ F)$
⟨*proof*⟩

*depends on:*
*theorem: computation-via-compute (page 103)*
*theorem: GoodIteratorX-F-spec-cont-bf-Fdes (page 98)*

**theorem** *computation-of-least-restrictive-maximal-solution-via-compute-for-cont2*:
$IsDES\ P$
$\Longrightarrow IsDES\ S$
$\Longrightarrow F = (((\lambda C.\ Fspec\ C\ S) \circ (\lambda C.\ Fcont2\ C\ SigmaUC\ (des\text{-}langUM\ P)\ (des\text{-}langUM\ C))) \circ Fbf \circ Fdes)$
$\Longrightarrow compute\text{-}dom\ (F,top)$
$\Longrightarrow compute\ F\ top = (gfp\ F)$
⟨*proof*⟩

*depends on:*
*theorem: computation-via-compute (page 103)*
*theorem: GoodIteratorX-F-spec-cont2-bf-Fdes (page 99)*

**theorem** *computation-of-least-restrictive-maximal-solution-via-compute-for-cont3*:
$IsDES\ P$
$\Longrightarrow IsDES\ S$
$\Longrightarrow F = (((\lambda C.\ Fspec\ C\ S) \circ (\lambda C.\ Fcont3\ C\ SigmaUC\ (des\text{-}langUM\ P)\ (des\text{-}langUM\ C))) \circ Fbf \circ Fdes)$

$\Longrightarrow$ *compute-dom* (*F*,*top*)
$\Longrightarrow$ *compute F top* = (*gfp F*)
⟨*proof*⟩

*depends on:*
*theorem: computation-via-compute (page 103)*
*theorem: GoodIteratorX-F-spec-cont3-bf-Fdes (page 100)*

### A.3.17 Verification of Fixedpoint Algorithm (with Precomputation and Conditional Composition)

**lemma** *computeN-invariant-after-fixed-point*:
*computeN n F A* = *F* (*computeN n F A*)
$\Longrightarrow$ *n*≤*na*
$\Longrightarrow$ *computeN n F A* = *computeN na F A*
⟨*proof*⟩

*depends on:*
*lemma: computeN-iterate-vs-post-apply (page 101)*

*required by:*
*theorem: equal-computation-via-compute (page 108)*

**theorem** *EqualCompute-initial*:
(⋀*C*. *Q C* $\Longrightarrow$ *F C* = *G C*)
$\Longrightarrow$ (⋀*n*. *Q* (*computeN n F S*))
$\Longrightarrow$ *Qinp S*
$\Longrightarrow$ (⋀*C*. *Qout C* $\Longrightarrow$ *Qinp C*)
$\Longrightarrow$ *computeN n F S* = *computeN n G S* ∧ *Q* (*computeN n F S*)
⟨*proof*⟩

*depends on:*
*lemma: computeN-iterate-vs-post-apply (page 101)*

*required by:*
*lemma: computeN-Fbf-Fcont3-Fbf-Fdes-eq-Fbf-ifcomp-Fcont3 (page 112)*
*lemma: computeterm-by-computeN (page 109)*
*theorem: equal-computation-via-compute (page 108)*

**theorem** *equal-computation-via-compute*:
(⋀*C*. *QFdom C* $\Longrightarrow$ *F C* = *G C*)
$\Longrightarrow$ (⋀*n*. *QFdom* (*computeN n F S*))
$\Longrightarrow$ *compute-dom* (*F*,*S*)
$\Longrightarrow$ *compute-dom* (*G*,*S*)
$\Longrightarrow$ *compute F S* = *compute G S*
⟨*proof*⟩

*depends on:*
*lemma: compute-by-computeN (page 102)*
*theorem: EqualCompute-initial (page 108)*
*lemma: computeN-invariant-after-fixed-point (page 108)*

*required by:*
*theorem: computation-via-compute2 (page 109)*
*theorem: Supremum-to-gfp-initialTranslate (page 109)*

**lemma** *computeterm-by-computeN-hlp*:

$F\ (computeN\ n\ F\ S) = computeN\ n\ F\ S$
$\Longrightarrow computeN\ n\ F\ S = computeN\ n\ G\ S$
$\Longrightarrow Qinp\ (computeN\ n\ F\ S)$
$\Longrightarrow (\bigwedge C.\ Qinp\ C \Longrightarrow F\ C = G\ C)$
$\Longrightarrow m \leq n$
$\Longrightarrow compute\text{-}dom\ (G,computeN\ (n-m)\ G\ S)$
⟨*proof*⟩

*depends on:*
*lemma: computeN-iterate-vs-post-apply (page 101)*

*required by:*
*lemma: computeterm-by-computeN (page 109)*

**lemma** *computeterm-by-computeN*:
$(\bigwedge C.\ Q\ C \Longrightarrow F\ C = G\ C)$
$\Longrightarrow (\bigwedge n.\ Q\ (computeN\ n\ F\ S))$
$\Longrightarrow compute\ F\ S = computeN\ n\ F\ S$
$\Longrightarrow F\ (computeN\ n\ F\ S) = computeN\ n\ F\ S$
$\Longrightarrow compute\text{-}dom\ (G,S)$
⟨*proof*⟩

*depends on:*
*theorem: EqualCompute-initial (page 108)*
*lemma: computeterm-by-computeN-hlp (page 108)*

*required by:*
*theorem: computation-via-compute2 (page 109)*
*theorem: Supremum-to-gfp-initialTranslate (page 109)*

**theorem** *computation-via-compute2*:
$GoodIteratorX\ G\ Q$
$\Longrightarrow (\bigwedge C.\ QFdom\ C \Longrightarrow F\ C = G\ C)$
$\Longrightarrow (\bigwedge n.\ QFdom\ (computeN\ n\ F\ top))$
$\Longrightarrow compute\text{-}dom\ (F,top)$
$\Longrightarrow compute\ F\ top = Sup\ \{X.\ Q\ X\}$
⟨*proof*⟩

*depends on:*
*lemma: Supremum-to-gfp (page 91)*
*theorem: equal-computation-via-compute (page 108)*
*theorem: computation-via-compute (page 103)*
*lemma: compute-by-computeN (page 102)*
*lemma: computeterm-by-computeN (page 109)*

**theorem** *Supremum-to-gfp-initialTranslate*:
$GoodIterator\ F\ UNIV\ Qterm\ Qout$
$\Longrightarrow compute\text{-}dom\ (G,S)$
$\Longrightarrow (\bigwedge n.\ computeN\ n\ F\ S = computeN\ n\ G\ S)$
$\Longrightarrow (\bigwedge C.\ QFdom\ C \Longrightarrow F\ C = G\ C)$
$\Longrightarrow (\bigwedge n.\ QFdom\ (computeN\ n\ F\ S))$
$\Longrightarrow compute\ G\ S = Sup\{X.\ Qterm\ X \wedge X \leq S\}$
⟨*proof*⟩

*depends on:*
*lemma: compute-by-computeN (page 102)*
*lemma: computeterm-by-computeN (page 109)*

*theorem: Supremum-to-gfp-initial (page 104)*
*theorem: equal-computation-via-compute (page 108)*

*required by:*
*theorem: compute-DESSIMMSol (page 113)*

**lemma** *GoodIterator-F-bf-spec*:
*IsDES P*
$\Longrightarrow$ *IsDES S*
$\Longrightarrow$ *GoodIterator*
(*Fbf*∘(%*C*. *Fspec C S*))
*IsDES*
(λ*X*. (λ*C*. *IsDES C* ∧ *PropSpec C S*)*X* ∧ (*IsDES*∩*PropBF*)*X*)
(*IsDES*∩*PropBF*)
⟨*proof*⟩

*depends on:*
*theorem: GoodIterator-compose (page 90)*
*theorem: GoodIterator-Fspec (page 96)*
*theorem: GoodIterator-Fbf (page 94)*

*required by:*
*lemma: computeN-Fbf-Fcont3-Fbf-Fdes-eq-Fbf-ifcomp-Fcont3 (page 112)*

**lemma** *GoodIterator-F-bf-cont-bf-des*:
*IsDES P*
$\Longrightarrow$ *IsDES S*
$\Longrightarrow$ *GoodIterator*
(*Fbf*∘(λ*C*. *Fcont3 C SigmaUC* (*des-langUM P*) (*des-langUM C*))∘*Fbf*∘*Fdes*)
*UNIV*
(λ*X*. (%*C*. *IsDES C*)*X* ∧ (*IsDES*∩*PropBF*)*X* ∧ (λ*C*. *IsDES C* ∧ *DESCont C P SigmaUC*) *X* ∧ (*IsDES*∩*PropBF*)*X*)
(*IsDES*∩*PropBF*)
⟨*proof*⟩

*depends on:*
*theorem: GoodIterator-compose (page 90)*
*theorem: GoodIterator-Fdes (page 94)*
*theorem: GoodIterator-Fbf (page 94)*
*theorem: GoodIterator-Fcont3 (page 96)*

*required by:*
*lemma: computeN-Fbf-Fcont3-Fbf-Fdes-eq-Fbf-ifcomp-Fcont3 (page 112)*
*theorem: compute-DESSIMMSol (page 113)*

**lemma** *spec-simplify*:
(λ*X*. (%*C*. *IsDES C*)*X* ∧ (*IsDES*∩*PropBF*)*X* ∧ (λ*C*. *IsDES C* ∧ *DESCont C P SigmaUC*) *X* ∧ (*IsDES*∩*PropBF*)*X*) = (%*C*. *IsDES C* ∧ *DESCont C P SigmaUC* ∧ *PropBF C*)
⟨*proof*⟩

*required by:*
*theorem: compute-DESSIMMSol (page 113)*

**definition** *DESMMSol* :: *'a DES* ⇒ *'a DES* ⇒ *'a set* ⇒ *'a DES set* **where**
*DESMMSol P S SigmaUC* = {*C*. *inf C P* = *Sup*{*C*. *IsDES C* ∧ *DESCont C P SigmaUC* ∧ *PropBF C* ∧ *PropSpec C S*}}

**definition** *DESSIMMSol* :: *'a DES* ⇒ *'a DES* ⇒ *'a set* ⇒ *'a DES set* **where**

*DESSIMMSol P S SigmaUC = DESMMSol P (inf P S) SigmaUC*

**lemma** *Fbf-mono*:
*x≤y*
*⟹ Fbf x≤Fbf y*
⟨*proof*⟩

*required by:*
*theorem: Supremum-in-DESSIMMSol (page 112)*

**lemma** *Fbf-term*:
*PropBF x*
*⟹ IsDES x*
*⟹ x=Fbf x*
⟨*proof*⟩

*required by:*
*theorem: Supremum-in-DESSIMMSol (page 112)*

**lemma** *Sup-DES-contained*:
*(⋀X. X∈A ⟹ IsDES X)*
*⟹ IsDES (Sup A)*
⟨*proof*⟩

*required by:*
*lemma: DESMMSol-contained (page 111)*

**lemma** *Sup-Cont-contained*:
*(⋀X. X∈A ⟹ DESCont X P SigmaUC)*
*⟹ DESCont (Sup A) P SigmaUC*
⟨*proof*⟩

*required by:*
*lemma: DESMMSol-contained (page 111)*

**lemma** *Sup-BF-contained*:
*(⋀X. X∈A ⟹ PropBF X)*
*⟹ PropBF (Sup A)*
⟨*proof*⟩

*required by:*
*lemma: DESMMSol-contained (page 111)*

**lemma** *Sup-Spec-contained*:
*(⋀X. X∈A ⟹ PropSpec X S)*
*⟹ PropSpec (Sup A) S*
⟨*proof*⟩

*required by:*
*lemma: DESMMSol-contained (page 111)*

**lemma** *DESMMSol-contained*:
*IsDES P*
*⟹ IsDES S*
*⟹ Sup ({ (X::'a DES).*
*(IsDES X ∧ DESCont X P SigmaUC ∧ PropBF X ∧ X ≤ inf P S)})*
*∈ { (C::'a DES).*

$(IsDES\ C \wedge DESCont\ C\ P\ SigmaUC \wedge PropBF\ C \wedge PropSpec\ C\ (inf\ P\ S))\}$
⟨*proof*⟩

*depends on:*
*lemma: Sup-DES-contained (page 111)*
*lemma: Sup-Cont-contained (page 111)*
*lemma: Sup-BF-contained (page 111)*
*lemma: Sup-Spec-contained (page 111)*

*required by:*
*theorem: Supremum-in-DESSIMMSol (page 112)*

**theorem** *Supremum-in-DESSIMMSol*:
$IsDES\ P$
$\Longrightarrow IsDES\ S$
$\Longrightarrow init{=}(Fbf{\circ}(\%C.\ Fspec\ C\ (inf\ P\ S)))top$
$\Longrightarrow Sup\{X.\ (\%C.\ IsDES\ C \wedge DESCont\ C\ P\ SigmaUC \wedge PropBF\ C)\ X \wedge X \leq init\} \in DESSIMMSol\ P\ S\ SigmaUC$
⟨*proof*⟩

*depends on:*
*theorem: GoodIterator-Fbf (page 94)*
*theorem: GoodIterator-Fspec (page 96)*
*lemma: infDES-preserves-IsDES (page 93)*
*lemma: le-trans (page 90)*
*lemma: Fbf-mono (page 111)*
*lemma: Fbf-term (page 111)*
*lemma: DESMMSol-contained (page 111)*

*required by:*
*theorem: compute-DESSIMMSol (page 113)*

**lemma** *computeN-preserve-prop*:
$Q\ S$
$\Longrightarrow (\bigwedge X.\ Q\ X \Longrightarrow Q\ (F\ X))$
$\Longrightarrow Q\ (computeN\ n\ F\ S)$
⟨*proof*⟩

*depends on:*
*lemma: computeN-iterate-vs-post-apply-hlp (page 101)*

*required by:*
*lemma: computeN-Fbf-Fcont3-Fbf-Fdes-eq-Fbf-ifcomp-Fcont3 (page 112)*

**lemma** *computeN-Fbf-Fcont3-Fbf-Fdes-eq-Fbf-ifcomp-Fcont3*:
$IsDES\ P$
$\Longrightarrow IsDES\ S$
$\Longrightarrow Q{=}(IsDES{\cap}PropBF)$
$\Longrightarrow computeN\ n$
$(Fbf \circ (\lambda C{::}'a\ DES.\ Fcont3\ C\ SigmaUC\ (des\text{-}langUM\ P)\ (des\text{-}langUM\ C)) \circ$
$Fbf \circ$
$Fdes)$
$(Fbf\ (Fspec\ top\ (inf\ P\ S))) =$
$computeN\ n$
$(Fbf \cdot \lambda C{::}'a\ DES.\ Fcont3\ C\ SigmaUC\ (des\text{-}langUM\ P)\ (des\text{-}langUM\ C))$
$(Fbf\ (Fspec\ top\ \ (inf\ P\ S))) \wedge Q\ (computeN\ n$
$(Fbf \circ (\lambda C{::}'a\ DES.\ Fcont3\ C\ SigmaUC\ (des\text{-}langUM\ P)\ (des\text{-}langUM\ C)) \circ$
$Fbf \circ$

*Fdes*)
(*Fbf* (*Fspec top* (*inf P S*))))
⟨*proof*⟩

*depends on:*
*lemma: GoodIterator-F-bf-spec (page 110)*
*lemma: infDES-preserves-IsDES (page 93)*
*theorem: EqualCompute-initial (page 108)*
*theorem: GoodIterator-Fdes (page 94)*
*theorem: GoodIterator-Fbf (page 94)*
*lemma: computeN-preserve-prop (page 112)*
*lemma: GoodIterator-F-bf-cont-bf-des (page 110)*

*required by:*
*theorem: compute-DESSIMMSol (page 113)*

**theorem** *compute-DESSIMMSol*:
*IsDES P*
⟹ *IsDES S*
⟹ *init*=(*Fbf*∘(%*C*. *Fspec C* (*inf P S*)))*top*
⟹ *G*=*Fbf*•(λ*C*. *Fcont*3 *C SigmaUC* (*des-langUM P*) (*des-langUM C*))
⟹ *compute-dom* (*G*,*init*)
⟹ *compute G init* ∈ *DESSIMMSol P S SigmaUC*
⟨*proof*⟩

*depends on:*
*theorem: Supremum-in-DESSIMMSol (page 112)*
*theorem: Supremum-to-gfp-initialTranslate (page 109)*
*lemma: spec-simplify (page 110)*
*lemma: GoodIterator-F-bf-cont-bf-des (page 110)*
*lemma: computeN-Fbf-Fcont3-Fbf-Fdes-eq-Fbf-ifcomp-Fcont3 (page 112)*
*theorem: GoodIterator-Fdes (page 94)*
*theorem: GoodIterator-Fbf (page 94)*

**end**