

Deterministic Pushdown Automata as Specifications for Discrete Event Supervisory Control in Isabelle

vorgelegt von
Dipl. Inform.
Sven Schneider
ORCID: 0000-0001-9828-618X

von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
– Dr.-Ing. –

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender: Prof. Dr. Rolf Niedermeier
Gutachter: Prof. Dr. Uwe Nestmann
Gutachter: Prof. Dr. Jörg Raisch
Gutachter: Prof. Dr. Holger Giese
Gutachter: Dr. habil. Florian Kammüller

Tag der wissenschaftlichen Aussprache: 26. Februar 2019

Berlin 2019

Deterministic Pushdown Automata as Specifications
for Discrete Event Supervisory Control in Isabelle

Deterministic Pushdown Automata as Specifications for Discrete Event Supervisory Control in Isabelle

Sven Falco Schneider

Technische Universität Berlin
Max-Planck-Institut für Dynamik komplexer technischer Systeme Magdeburg
Hasso-Plattner-Institut für Digital Engineering gGmbH

CONTENTS

<i>Abstract</i>	ix
1. Introduction	1
2. Abstract and Concrete Discrete Event Systems	15
2.1. Discrete Event Systems	17
2.2. Extended Pushdown Automata	18
2.3. Parsers	22
2.4. Context-free Grammars	27
3. Abstract and Concrete Supervisory Control Problems	31
3.1. Abstract Supervisory Control Problem for Discrete Event Systems	33
3.2. Concrete Supervisory Control Problem for Deterministic Pushdown Automata	36
3.3. Correspondence between Abstract and Concrete Supervisory Control Problems	41
4. Abstract Controller Synthesis Algorithm for Discrete Event Systems	45
4.1. Framework of Abstract Building Blocks for Fixed-point Computation	47
4.2. Abstract Building Blocks for Enforcing Properties on Discrete Event Systems Least Restrictively	50
4.3. Abstract Controller Synthesis Algorithm for Discrete Event Systems	53
4.4. On the Termination of the Abstract Controller Synthesis Algorithm	56
5. Concrete Controller Synthesis Algorithm for Deterministic Pushdown Automata	61
5.1. Concrete Building Block for the Synchronous Composition of Deterministic Pushdown Automata with Deterministic Finite Automata	64
5.2. Concrete Building Block for Enforcing Nonblockingness for Deterministic Pushdown Automata	65
5.3. Concrete Building Block for Reducing Controllability to Nonblockingness for Deterministic Pushdown Automata	82
5.4. Concrete Synthesis Algorithm as an Instantiation of the Abstract Synthesis Algorithm	85
5.5. On the Termination of the Concrete Controller Synthesis Algorithm	87
	vii

6. Isabelle-based Formal Quality Assurance	89
6.1. Formal Methods for Quality Assurance	91
6.2. Isabelle-based Framework of Definitions and Properties	96
6.3. Isabelle-based Verification of the Translation of Deterministic Pushdown Automata into LR(1)-context-free Grammars	116
7. Application and Prototype-based Evaluation	141
7.1. Patterns for Specifications using Deterministic Pushdown Automata	143
7.2. Application Domains for Discrete Event Controller Synthesis	146
7.3. Applications and Use Cases of the Concrete Controller Synthesis Algorithm	147
7.4. Prototype Realization of the Concrete Controller Synthesis Algorithm	154
7.5. Prototype-based Evaluation	157
8. Related, Ongoing, and Future Work	163
8.1. Related Work	164
8.2. Ongoing Work	188
8.3. Future Work	209
9. Summary and Conclusion	219
A. Disclaimer on Collaborations and Joint Work	225
B. Isabelle-based Notation	227
C. Operational Properties for DPDA Controllers	233
D. The <i>cfgEsplit</i> Semantic for LR(1)-CFG	237
E. Bibliography	249

ABSTRACT

The problem of supporting the construction of software that is known to satisfy a set of given requirements is one of the grand challenges in software engineering. Herein, we focus on the field of control theory for systems with discrete states and event-based communication. In this field, controllers coordinate components given by a plant to ensure that the amalgamation of controller and plant executes a desired behavior. We focus on the supervisory control problem, which, given a plant and a specification, requires the synthesis of a controller in the form of a piece of software. The least restrictive satisfactory controllers to be synthesized are determined in this problem by the specification and additional well-formedness conditions. Similar problems also occurred in the field of computer science and it is of general importance in application domains such as in parallel, distributed, and embedded systems.

We focus on the fully automatic synthesis of controllers using algorithms. These algorithms construct controllers that are realizable in software and correct-by-construction for the two aforementioned inputs. The applicability of these algorithms is limited by insufficient expressiveness of the formalisms used for plants and specifications. However, expressiveness can not be increased arbitrarily while maintaining solvability of the problem in terms of a synthesis algorithm that solves all problem instances.

Our main contribution is a controller synthesis algorithm that synthesizes a DPDA controller when provided with a DFA plant model and a DPDA specification. This algorithm supersedes earlier algorithms synthesizing a DFA controller when provided with a DFA plant model and a DFA specification because DPDA are strictly more expressive than DFA. The increased expressiveness of DPDA compared to DFA allows for the specification and enforcement of more complex patterns of behavior. Initial approaches to mitigate limitations of our algorithm, such as its nontermination for some specifications stating requirements that are unreasonable from a control-theoretic perspective, are presented in the form of alternative constructions and optimizations.

We employed the interactive theorem prover Isabelle for the formal verification of this controller synthesis algorithm to obtain trustworthy proofs, which are free of faults and omissions. The resulting Isabelle framework for the formalization and verification of algorithms outstrips existing similar frameworks in covered formalisms, semantical properties, and provided results and is designed to be highly extendable in these aspects. It covers the formalisms of DPDA, CFGs, and Parsers as well as the notions relevant for our controller synthesis algorithm such as the unmarked and marked languages, nonblockingness, and controllability.

An evaluation of our algorithm implemented as the Java prototype CoSy shows promising efficiency. This evaluation was based on three examples from manufacturing employing DPDA specifications. Using these examples, we also demonstrate the application of three use cases of our algorithm for controller synthesis, controller verification, and input validation.

1

Introduction

The problem of supporting the *construction of software that is known to satisfy a set of given requirements* is one of the grand challenges in software engineering. We focus on software that interacts with its environment where the requirements then refer to the *closed loop*, which is the amalgamation of the constructed software and the environment. Established processes for the construction of such software vary in their degree of automation: the spectrum ranges from *fully automatic synthesis* to *fully manual implementation*. We focus in this thesis on fully automatic synthesis and present a formally verified synthesis procedure as one of our key contributions.

The fully automatic synthesis has been stated as a problem in the construction of hardware circuits in 1957 [81, 121]. A second formulation of a controller synthesis problem in terms of a game between a player representing the software to be constructed and a player representing the environment was first described as early as 1965 [244, 66, 237, 356]. In this game-based perspective, any finite winning strategy employed by the first player translates to a realization of the software to be constructed. Similar controller synthesis problems appeared in the form of the submodule construction problem in 1983 [245] and the *supervisory control problem* in 1984 [282].

Common to these formulations of control problems is the assumption of a message-based communication between the system and the environment. This assumption is readily satisfied by a variety of systems from the fields of computer science and control theory in which software is required to control or to coordinate components to ensure the execution of desired behavior such as communicating and embedded systems. These control problems are based on the same kinds of inputs: a model of the environment, of how software and environment coalesce into the closed loop, and of the requirements to be satisfied by the software and closed loop. We focus in this thesis on the supervisory control problem and continue now with a more detailed introduction of it.

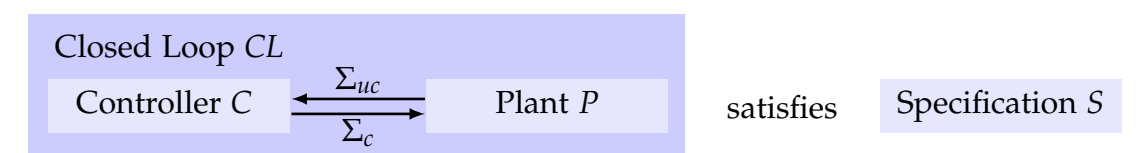
→ *The Supervisory Control Problem*

The supervisory control problem [282], for which we present an example on page 5, is a software synthesis problem related to the field of control theory. Hence, a field-specific terminology is used: the environment is called *plant* and the software that interacts with the plant is called *controller*. The purpose of the controller is to limit the behavior of the plant to the boundaries given by an additional specification that states requirements specific to the problem instance. A list of *general requirements*, which includes the satisfaction of the specification, then describes all desirable controllers of which one is to be synthesized. We now continue with a short discussion of the fundamental assumptions on how plant and controller are combined into a closed loop and then also present the requirements to be satisfied by controller and closed loop.

The plant and the controller are assumed to send messages, subsequently called events, to each other using reliable synchronous communication in the closed loop. Synchronous communication means here that the sending and the receiving of a single event occurs in a single (atomic) step that is carried out by sender and receiver at the same time. The reliability and synchronicity of the communication ensure the absence of effects such as event loss, alteration, duplication, insertion, or reordering. The events sent by the plant carry information from *sensors* and the events sent by the controller represent commands to be carried out by *actuators*. The sender and receiver of an event may change their local *state* upon synchronization. The state of a component is thereby given by the local knowledge of that component on the status of the closed loop that it has acquired in the past and that is the foundation for decisions on steps in its future.

The response of the plant (if any) to communication attempts initiated by the controller is predefined because the plant is one of the inputs to be provided. That is, the provided plant model specifies for each state of the plant and each event sent by the controller whether the plant is willing to synchronously receive this event or whether the controller is not able to send the event. Thereby, the controller is limited by the plant model in its capabilities to control the plant. However, since the controller is to be synthesized, the existence of a response to every event sent by the plant is an important requirement, called *controllability*. The satisfaction of controllability by the controller ensures that (a) the state of the controller also includes the relevant information on the state of the plant at the same time, (b) the controller can derive well-founded decisions, and (c) the models for controllers and closed loops are equivalent in formal considerations.

Figure 1.1: «The Supervisory Control Problem»



The general requirements of the control problem (which are discussed subsequently) rely on a uniform characterization of the behavior of the plant, the controller, the closed loop, and the specification. This characterization is given for

each of these components by the set of *unmarked behaviors* and a particular subset thereof called the set of *marked behaviors*. The unmarked behaviors of plant and controller are the sequences of events from a finite set Σ that *may* be used in successive steps by this component; the specification also describes such sequences without use of an actual communication partner. The marked behaviors identify a goal region or those unmarked behaviors in which a task has been completed. While every prefix of an unmarked behavior is an unmarked behavior, we permit cases where this does not apply for the marked behaviors. The (un)marked behaviors of the closed loop are given by the intersection of the (un)marked behaviors of plant and controller due to their synchronous composition.

The event set Σ is partitioned into the sets of events Σ_c and Σ_{uc} that are sent by the controller and plant, respectively. The indices c and uc abbreviate *controllable* and *uncontrollable*, respectively, and are chosen based on the perspective of the prospective controller. Firstly, the decision on whether or not to send an event from Σ_c is fully controllable by the controller. Secondly, the events that are sent by the plant are not under the influence of the controller and must not be rejected.

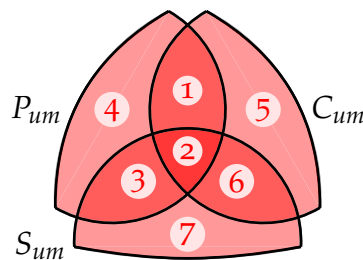
The requirement of *controllability* states that the controller never prevents the plant from sending an event by refusing its reception. An example of the notion of controllability is given in Example 1.1|p.4.

The requirement of *specification satisfaction* states that the controller must successfully restrict the (un)marked behaviors of the plant (that is, the uncontrolled behaviors of the plant) to those of the specification (that is, the safe region and the goal region). This means that the (un)marked behaviors of the closed loop must be contained in those of the specification.

The requirement of *nonblockingness* states that every unmarked behavior of the closed loop can be continued to a marked behavior of the closed loop. The notion of nonblockingness is exemplified in Example 1.1|p.4. We conclude that the marked behavior, which is required to be invariantly reachable, is also limited by the marked behaviors of the specification when the specification is satisfied.

Figure 1.2: «Relationship between Plant, Specification, and Controller»

We relate the unmarked behaviors of the plant (P_{um}), the specification (S_{um}), and the controller (C_{um}) using the Venn diagram below. The sets 1 and 2 are the unmarked behaviors of the closed loop where the elements in 1 violate the specification. The sets 3 and 4 are unmarked behaviors of the plant that are prevented by the controller where only the elements in 4 had to be prevented according to the specification. The sets 5, 6, and 7 are permitted by the controller/specification even though the plant is not capable of executing them.



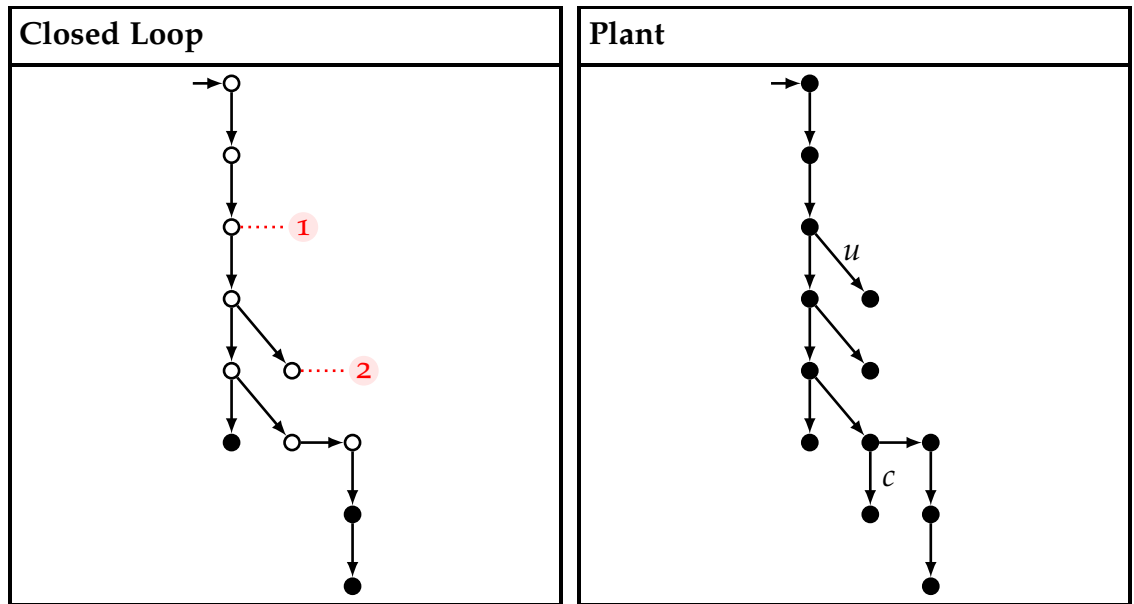
We conclude our presentation by stating the supervisory control problem in terms of its inputs, outputs, and the requirements stated on them. The *inputs* are a plant (model) to be controlled, a specification (model) to be enforced, and the partitioning of Σ into the sets Σ_{uc} and Σ_c . The *output* is a controller (model) to be synthesized that satisfies the three discussed requirements *least restrictively*. The three requirements ensure the absence of undesirable behavior while the property of least restrictiveness ensures a maximal amount of desirable behavior (i.e., the set 3 in Figure 1.2|p.3 should be as small as possible).

The supervisory control problem as stated in [282] also used a second specification that describes marked and unmarked behaviors that must be feasible in the closed loop. However, the satisfaction of this requirement can also be checked a posteriori for a synthesized controller and its induced closed loop.

In addition to the introduction of the two properties of controllability and nonblockingness from before, we demonstrate violations of these two properties in the following example.

Example 1.1: «Properties of Controllability and Nonblockingness»

We represent the sets of (un)marked behaviors of components as trees where each path is an unmarked behavior and where each path to a filled node is a marked behavior. The left tree for the closed loop is to be understood as a subtree of the right tree for the plant: we label only the edges that are relevant to our reasoning with the event used in this step and assume $u \in \Sigma_{uc}$ and $c \in \Sigma_c$. The closed loop violates the property of controllability at 1 because it prevents the occurrence of the uncontrollable event u at the designated unmarked behavior. The prevention of the step using the event c is no violation of controllability because c is controllable. The closed loop violates the property of nonblockingness at 2 because the designated unmarked behavior cannot be continued to a marked behavior.



—• *Automata-based Concrete Supervisory Control Problem*

Concrete instantiations of the presented supervisory control problem are obtained by (a) selecting formalisms for modelling plants, specifications, controllers, and closed loops, (b) defining for instances of these formalisms the sets of (un)marked behaviors, and (c) defining an operation that, based on the selected formalisms, constructs for a plant and a controller the induced closed loop. Every instance of the resulting concrete supervisory control problem is then given, as before, by a plant model, a specification model, and a set of events Σ that is partitioned into Σ_c and Σ_{uc} as described.

In this thesis, we employ classes of automata as formalisms to model the four above mentioned components with discrete states and an event-based communication as in [282]. The used classes of automata have the generation of (un)marked behaviors as a built-in concept and constructions for implementing the synchronous composition of plant and controller are available as well. Moreover, these constructions also ensure that the sets of (un)marked behaviors of the closed loop are equal to the intersection of the (un)marked behaviors of the plant and the specification as assumed in the supervisory control problem.

A first concrete supervisory control problem occurred already in [282]. In this concrete problem, to which we refer to as the *pure DFA problem*, all involved components are deterministic finite automata (DFA). Moreover, this concrete problem served as the foundation for the vast majority of further research results in the domain of supervisory controller synthesis thereafter.

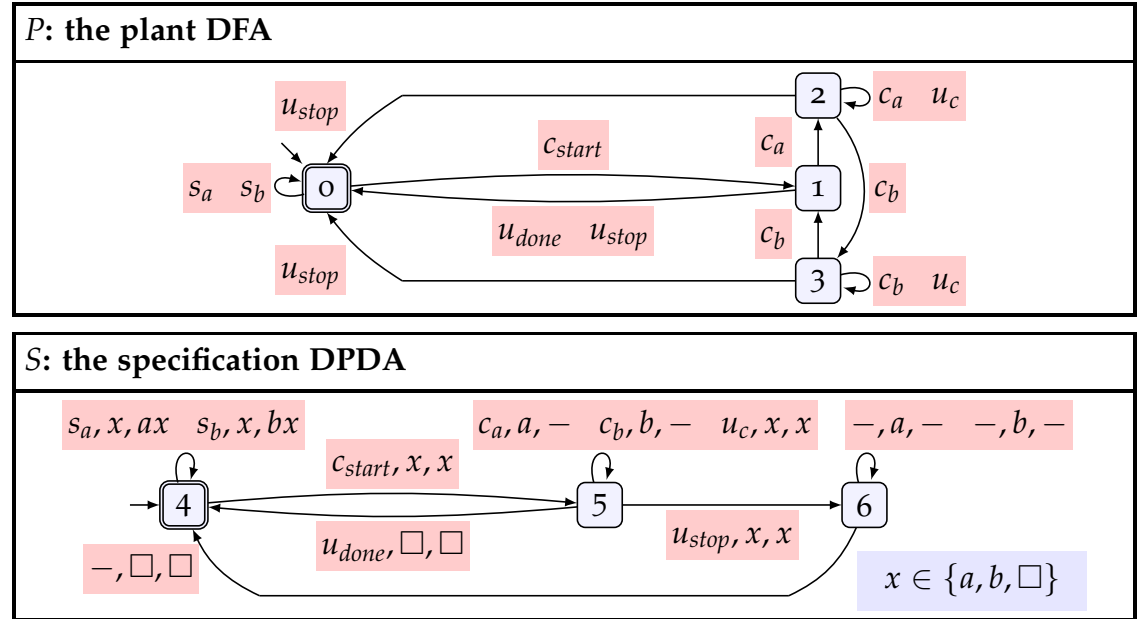
In this thesis, we consider a second concrete supervisory control problem in which plants are modelled by DFA, where deterministic pushdown automata (DPDA) allow for more complex specifications, and result in DPDA controllers and closed loops. These DPDA can be understood as DFA that are endowed with a stack of unbounded depth, which permits the storage of information for later reuse at runtime and thereby increases the expressiveness of the specification. Technically, a stack is a sequence of symbols that can only be modified by adding symbols to its left or by removing symbols from its left and that only permits read access to the left-most symbol. The second concrete problem subsumes the pure DFA problem since each DFA is also a DPDA.

We now consider an instance of the second concrete supervisory control problem using a DPDA specification to motivate its meaningfulness.

Example 1.2: «Validation of Schedules Using Controller Synthesis»

Description of Scenario: A schedule is provided to the plant by an external operator in the form of a list of events that are sent to the plant. The controller to be synthesized records the events of the schedule using its stack and permits only schedules that can be executed successfully to the end. The external operator then initiates the execution of the schedule by the plant and the controller ensures that the plant actually implements the entered schedule.

The notation used in the following visualizations for the DFA plant and the DPDA specification is discussed below.



Notation and Semantics: DFA and DPDA are visualized as labelled directed graphs where nodes are called states. The states 0 and 4 are the initial states of the two automata as indicated by the arrow without source state pointing to them. These two states are also marking states as indicated by their double border line and correspond to the goal region for both automata to be reached. Two edges with common source and target are represented by only one edge where the two labels are given next to each other (e.g. the edge from state 1 to state 0 or from state 6 to state 6). The labels of the DFA edges have one component: the event executed. The labels of the DPDA edges have three components separated by commas: the event executed, the symbols that are popped from the top of the stack, and the symbols that are pushed onto the stack. The symbol $-$ is used to denote the absence of symbols and events in a component and the placeholder x is used to specify multiple edges at once. We introduce DPDA in section 2.2|p.18 and subsection 6.2.6|p.112.

The steps of a DFA depend on the states contained in its *configurations*. An application of an edge replaces the state in a configuration, which must be the source state of the edge, with the target state of the edge. Iterated edge applications lead to sequences of configurations called derivations. See also Example 6.2|p.113 for an example of a derivation of another DPDA using two equivalent semantics.

Configurations of DPDA additionally contain the current stack. Moreover, the symbols given in the second component of a label of the edge, say w_1 , must also be on top of the stack, i.e., the stack equals w_1v for some v , for applicability of an edge. A step then additionally updates the stack of the configuration to w_2v . The stack initially contains only the special symbol \square , which denotes the bottom of the stack and which stays there throughout any derivation. DPDA may perform silent steps (e.g. using the edge from state 6 to state 4) executing no event.

Description: In the initial state 0 of the plant, a schedule can be entered by an operator by executing the events s_a and s_b . These executions are to be recorded by the controller as specified in state 4 where a and b are pushed onto the stack for each execution of the events s_a and s_b , respectively (the symbol x allows for a more compact visualization using edge schemas).

The execution of the event c_{start} changes the state to 1, ends the phase of entering the schedule, and starts the phase in which the schedule is executed in some process from the concrete application domain. The events executed in this process are limited via the edges between the plant states 1–3 to sequences of at least one c_a , one c_b , and at least one c_c where further u_c events may be executed in between. The specification then also determines in state 5 how the controller must restrict the plant in these states to implement the schedule by popping symbols a and b from the stack for each execution of one c_a and c_b , respectively. When the schedule is completed (that is, when it is empty) and the state 1 has been reached again, the machine can return to the marking state 0 by executing the u_{stop} event. Finally, the operator may execute the event u_{stop} to interrupt the execution of the schedule, to reset the schedule in state 6 of the specification, and to also return to the initial state 4.

In this example, we assume that the set of events Σ is partitioned into the sets $\Sigma_c = \{s_a, s_b, c_a, c_b, c_{start}\}$ and $\Sigma_{uc} = \{u_c, u_{stop}, u_{done}\}$. This means that the controller to be synthesized may prevent certain schedules from being entered (e.g. those schedules that request to start with an event c_b). This check is carried out when the operator attempts to execute events s_a and s_b synchronously with the controller, that is, the schedule is validated before the event c_{start} is executed. Moreover, the controller to be synthesized can not prevent executions of the uncontrollable event u_c during schedule implementation.

Discussion: The usage of automata as an *operational* formalism leads to specifications that describe the usage of the stack by the prospective controller already in great detail. We take advantage of this by using the specification as an initial controller candidate, which is then restricted in further steps. Hence, the DPDA specification can refine the DFA plant because the specification does not distinguish between restrictions that are meant to refine the plant model and those that should be enforced by the prospective controller. However, this specification-based refinement of the plant is limited because the occurrence of events from Σ_{uc} is obtained from the plant alone for controllability.

Moreover, the usage of the stack of DPDA is used here to store the schedule to be implemented and to preemptively check its validity. Such an upfront check is not possible when using DFA specifications where a nonexecutable schedule (entered event-wise at runtime or stored entirely without formal foundation beforehand) would result in a deadlock at runtime.

Finally, resetting the schedule in state 6 requires an unbounded number of internal steps in the controller and, hence, the controller to be synthesized will not have a worst case execution time (WCET). However, a runtime environment that executes the controller can clear the stack in constant time in this example.

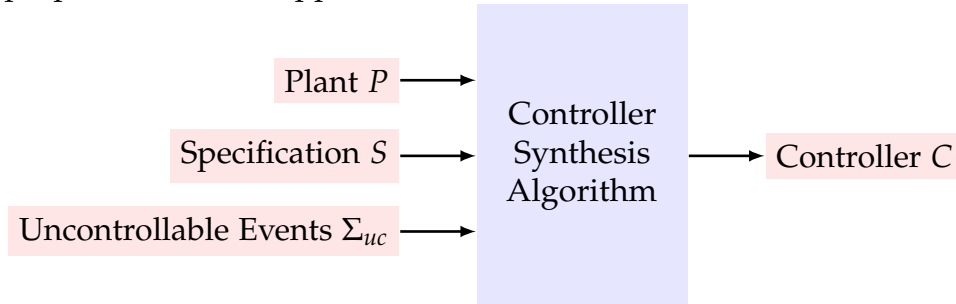
→ *Relevance and Applicability of our Contributions*

The controller synthesis algorithm for solving the pure DFA problem from [282] has been applied successfully in an abundance of domains such as manufacturing, robotics, chemical process control, protocol design for communication, feature interaction management in telephony, queueing systems, traffic control, database systems, hybrid systems, and fault diagnosis [70, 71]. The development of a controller synthesis algorithm in this thesis for more expressive specifications given by DPDA enables the specification and enforcement of even more complex properties and maintains applicability in the described domains.

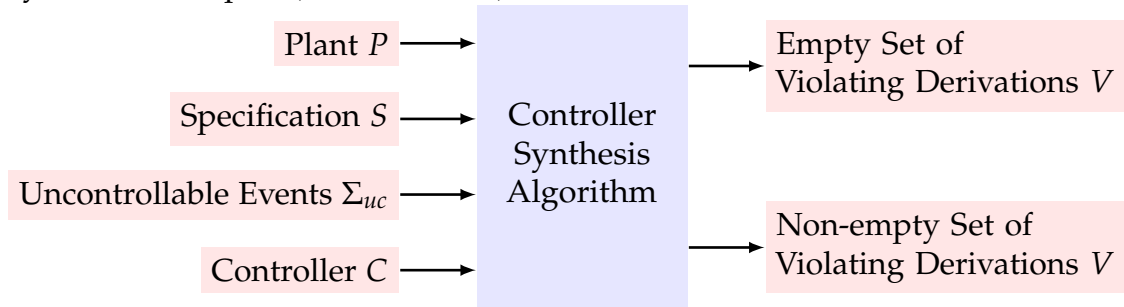
We determine three complementing use cases for the application of our concrete controller synthesis algorithm. These use cases take advantage of the fact that our algorithm, in the same way as the algorithm provided in [282] for the pure DFA problem, can not only be used to synthesize a controller but can also be used to verify that a provided controller candidate is satisfactory or to determine violations of controllability for a provided controller candidate. These algorithms refine an initially determined controller candidate by (a) identifying violations of the requirements and by (b) removing such violations in a follow up step while maintaining all behaviors without violations.

Figure 1.3: «Use Cases of Our Concrete Controller Synthesis Algorithm »

Input/Output Situation for Controller Synthesis: The standard controller synthesis (first use case) requires no human involvement, but defective inputs can lead to a waste of resources and the resulting controller may not have all required properties in some application contexts.



Input/Output Situation for Controller Analysis: The fully automatic analysis of a controller by means of a formal foundation can be useful if a manually constructed controller is to be verified (second use case) or when the specification as the initial controller candidate is to be adapted manually to remove violations by a domain expert (third use case).



We underline the applicability of DPDA specifications in the context of controller synthesis by introducing usage patterns for DPDA specifications (beyond what was used in the scheduling Example 1.2|p.5 above) and demonstrated their usefulness by applying them in examples in chapter 7|p.141. Moreover, DPDA can be much more succinct compared to equivalent DFA and, hence, their usage can ease the process of formalizing and managing also DFA specifications in terms of the required expertise and costs. Additional challenges originating in the use of DPDA and initial steps for their resolution, such as the potential absence of a WCET, are discussed in chapter 8|p.163.

Potential applications of our algorithm for solving the considered instantiation of the supervisory control problem with DPDA specifications in other contexts of computer science for software synthesis (e.g. for solving instances of the submodule construction problem) and extensions of this algorithm further strengthening applicability are discussed in chapter 8|p.163.

Before detailing on our concrete controller synthesis algorithm for the concrete supervisory control problem, we now discuss several additional aspects of systems not covered in this thesis and the alternative approach of manual implementation.

— Scope of this Thesis

The vast number of combinations of potential (kinds of) traits of the closed loop and the properties of the closed loop to be guaranteed results in a large multidimensional space of instances of control problems defined by means of suitable formalisms capable of capturing these characteristics. Consequently, research focuses on the construction of synthesis algorithms that solve meaningful (classes of) control problems as in [282] and on establishing negative results stating that certain (classes of) control problems defy solution as in [273].

An increased expressiveness allows for more precise modelling of the possible event sequences, but it complicates the problem of fully automatic synthesis because effective and efficient operations must be available (a) for the explicit construction of a closed loop and (b) for the analysis and enforcement of the properties that are required for controllers to be synthesized. For example, Turing machines are beneficial for modelling due to their high expressive power but only structural properties can be analyzed for them. Extending the pure DFA problem by using the more expressive DPDA for expressing specifications is thereby one further attempt to push the boundaries of fully automatic synthesis using one particular choice of formalisms.

Further aspects that have been considered in controller synthesis to increase expressiveness of models/specifications or to enhance decidability such as duration of events, delay between events, probabilistic step decisions, imperfect communication, asynchronous communication, costs, rewards, horizontal composition of controllers, and vertical composition of controllers are not in the scope of this thesis. However, we point out relevant connections and approaches for improving some aspects of our contributions in chapter 8|p.163.

→ *Comparison of Fully Automatic Synthesis and Fully Manual Implementation*

We now discuss general benefits and impediments of fully automatic and fully manual controller construction as well as possibilities of their combination.

As pointed out before, automatic synthesis cannot replace manual construction in all cases because some control problems do not allow for a general solution in terms of a synthesis algorithm when the expressive power of the employed formalisms is too large. However, models of such formalisms that occur in an actual application may be simple enough for manual implementation. That is, there is no known process for obtaining synthesis algorithms from the domain expertise of developers of such manual solutions. We now assume a setting in which both approaches are applicable.

The development of synthesis algorithms imposes additional upfront costs compared to manual construction. However, once an effective, efficient synthesis algorithm has been constructed, the synthesis costs are mostly related to hardware costs and can be reduced by means of techniques such as multithreading while costs in manual construction are related to the required time and human personnel. Moreover, synthesis algorithms are limited to the formalisms for which they have been developed whereas the manual construction is more flexible as it does not require a general solution to *all* problem instances but rather the solution of the single problem instance at hand.

The verification of synthesis algorithms is of particular importance because (a) the manual analysis of each synthesis step would incur further costs and (b) errors in the synthesis algorithms could have arbitrarily adverse effects. After all, the correct-by-construction principle of controller synthesis is the key motivation for its application because the trustworthiness of constructed controllers is vital. Manual implementations of controllers are prone to error and require post-construction analysis using e.g. computer-aided techniques such as simulation, testing, or even more costly techniques of verification such as model checking or manual verification depending on the tractability of the analysis and the desired level of trustworthiness.

A precise understanding of the plant is required in either of the approaches. The manual approach does not depend on the usage of a certain formalism whereas synthesis algorithms require formal inputs. However, the formalization of assumptions on the plant and guarantees given in the form of the specification leads to a precise documentation. While this documentation requires an additional effort in the manual implementation approach, it automatically stays in sync with the synthesized controller. Defective plant/specification models are a common problem of both approaches as they result in controllers that induce closed loops with very limited or even unsafe behavior. While some kinds of modelling defects would be detected by experienced developers, validation techniques for these inputs are indispensable in both approaches.

A first combination of both approaches is given when controllers have been developed using both approaches. These two controllers can then be compared using back to back testing to reveal various implementation errors or modelling

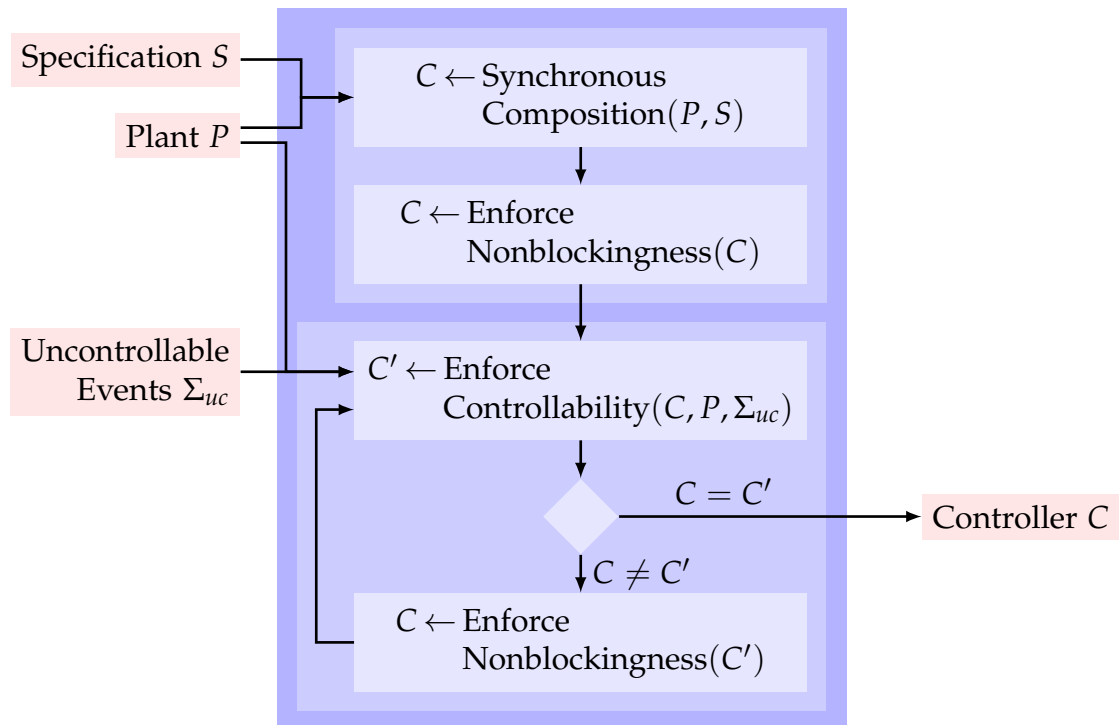
defects. Moreover, some controller synthesis algorithms such as the one in this thesis can be used to analyze a provided (possibly manually constructed) solution by verifying the absence of violations of the requirements or by returning descriptions of instances of such violations. Thereby, such synthesis algorithms allow for the combination of manual implementation and fully automatic synthesis with their respective advantages into a powerful technique as explained in the previous paragraph. We conclude this basic comparison of the two approaches by stating that the mentioned advantages and disadvantages may be of varying relevance in different application scenarios.

—• *Approach and Roadmap*

We define our automata-based concrete controller synthesis algorithm as an instantiation of an abstract controller synthesis algorithm, which is constructed using three building blocks for synchronous composition, for enforcing controllability, and for enforcing nonblockingness. The abstract algorithm operates on (abstract) discrete event systems (DES), which are obtained from automata by selecting their two generated languages.

Figure 1.4: «Fixed-point Controller Synthesis Algorithm»

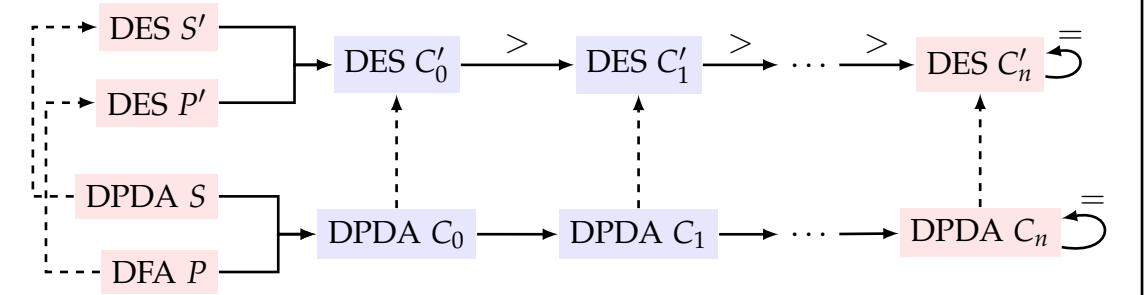
Abstract/Concrete Controller Synthesis Algorithm: The arrows represent dataflow and the rhombus represents a test on whether the fixed point has been computed, that is, whether the former controller candidate C is equal to the controller candidate C' obtained from enforcing controllability on C . For our formal verification, we define pre/post conditions for each of the three major building blocks used in this algorithm at the abstract level of DES as well as on the concrete level of automata.



In more detail, the abstract algorithm uses the DES specification S as a first controller candidate, computes the closed loop for this controller candidate and the given DES plant P , and then enforces the property of nonblockingness on this closed loop to obtain a controller candidate that satisfies the properties of nonblockingness and of specification satisfaction. The abstract algorithm then proceeds by computing a fixed point, starting with this controller candidate, and enforces controllability and nonblockingness alternately until the step of enforcing controllability does not further restrict the controller candidate.

We provide automata-based operations instantiating these three building blocks to obtain our concrete controller synthesis algorithm as an instantiation of the abstract algorithm. Consequently, both algorithms operate equivalently on the concrete DPDA controllers and on the abstract DES controllers.

Corresponding Fixed-point Computation of Both Algorithms: The concrete and the abstract synthesis algorithms operate equivalently when being started on the corresponding inputs: a DPDA specification S and a DFA plant P and on the DES abstraction S' and P' , respectively. That is, the obtained controller candidates computed by the abstract algorithm will be the abstractions of the controller candidates computed by the concrete controller synthesis algorithm (visualized by the dashed arrow). This correspondence results from the construction of the concrete algorithm as an instantiation of the abstract algorithm and the compatibility of this instantiation with the pre/post conditions provided for both algorithms.



In this thesis, we rely on the interactive theorem prover Isabelle to ensure that our definitions and constructions are type-correct and that the proofs of our theorems are fault-free and omission-free. For this purpose, we developed a framework in Isabelle for the handling of the involved formalisms and the constructions applied on them in our synthesis algorithms. Note, we provide a brief overview of the syntax and semantics of Isabelle in Appendix B|p.227.

Some of the contributions of this thesis (in particular results from section 3.1|p.33, chapter 4|p.45, section 5.3|p.82, and an example in section 7.3|p.147) have been published in [310, 301, 302] and refereed in [307, 309]. Further publications [308, 306, 305] are not part of this thesis, but they lay the foundation for future applications of our concrete controller synthesis algorithm. The role of the coauthors for the development of the contributions presented in this thesis is covered in Appendix A|p.225.

Contents of this Thesis

- 2|p.15 *Abstract and Concrete Discrete Event Systems*
We introduce the abstract and concrete formalisms of discrete event systems, automata, context-free grammars, and Parsers used in our controller synthesis algorithms.
- 3|p.31 *Abstract and Concrete Supervisory Control Problems*
We formally introduce the abstract and the concrete supervisory control problem, which are defined by means of discrete event systems and automata, respectively.
- 4|p.45 *Abstract Controller Synthesis Algorithm for Discrete Event Systems*
We formally introduce the above sketched abstract supervisory controller synthesis algorithm including our theory of fixed-point iterators ensuring the computation of the least restrictive DES controller.
- 5|p.61 *Concrete Controller Synthesis Algorithm
for Deterministic Pushdown Automata*
We formally introduce the instantiation of the abstract supervisory control problem for DFA plants and DPDA specifications by providing detailed instantiations of the three major building blocks.
- 6|p.89 *Isabelle-based Formal Quality Assurance*
We discuss our general usage of Isabelle and also the Isabelle framework [303] developed specifically for the verification of our formal results. Moreover, we provide a proof idea for our novel proof of the fact that DPDA can be converted into LR(1)-CFG.
- 7|p.141 *Application and Prototype-based Evaluation*
We demonstrate the implementability of our developed concrete controller synthesis algorithm by our prototype implementation CoSy [304]. Moreover, we discuss general applicability of the algorithm by means of usage patterns for DPDA specifications and also consider the efficiency of our implementation.
- 8|p.163 *Related, Ongoing, and Future Work*
We outline several meaningful results from ongoing research, provide a discussion of related work as well as recommendations for future work.
- 9|p.219 *Summary and Conclusion*
We conclude the thesis with an overall evaluation of our contributions and a discussion of their significance.

2

Abstract and Concrete Discrete Event Systems

Our concrete controller synthesis algorithm presented in chapter 5|p.61 requires formalisms for describing plants, specifications, closed loops, and controllers. The inputs and outputs of this algorithm are modelled using the concrete formalisms of DFA and DPDA defined in this chapter. We also introduce extended pushdown automata (EPDA), Parsers, and context-free grammars (CFGs) for models occurring during computations of this algorithm. As a foundational theory, we formalize these concrete formalisms in Isabelle as instantiations of abstract parametrized theories by providing for each parameter of these theories and each concrete formalism an interpretation. We introduce the formalisms in this chapter at an intuitive level and provide relevant details of our Isabelle-based framework in section 6.2|p.96.

Common to these concrete formalisms is that they define unmarked and marked languages as abstractions of actual and desired behavior. The abstract formalism of DES captures these two kinds of languages in a novel unified representation, which has distinct benefits over approaches from the literature using only one of the two languages. That is, we benefit from their simplicity and the inclusion of both languages when defining the abstract supervisory control problem in section 3.1|p.33 as well as when defining the abstract synthesis algorithm in chapter 4|p.45 at this high level of abstraction. Moreover, we show in section 3.3|p.41 that DES are a suitable abstraction for models of the used concrete formalisms in the sense that the abstract and the concrete supervisory control problem from section 3.2|p.36 correspond to each other.

CONTENTS OF THIS CHAPTER

2.1|p.17 *Discrete Event Systems*

We introduce the abstract formalism of discrete event systems (DES) along with basic composition and comparison operations on them. Moreover, we demonstrate that DES determine a complete lattice by instantiating the corresponding parametrized theory.

2.2|p.18 *Extended Pushdown Automata*

We introduce the concrete formalism of extended pushdown automata (EPDA) with various of its subclasses such as DFA, DPDA, and SDPDA.

2.3|p.22 *Parsers*

We introduce the concrete formalism of Parsers, of which instances occur in computations of our concrete controller synthesis algorithm.

2.4|p.27 *Context-free Grammars*

We introduce the concrete formalism of context-free grammars (CFGs), which is also used to represent intermediate models obtained in computations of our concrete controller synthesis algorithm.

2.1. DISCRETE EVENT SYSTEMS

We use various concrete formalisms to represent plants, controllers, and specifications in our concrete controller synthesis algorithm in chapter 5|p.61. These formalisms determine for their models *marked and unmarked* languages over an event alphabet. On the one hand, these unmarked languages are prefix-closed and describe all possible evolutions of the models. On the other hand, the marked languages are contained in the unmarked languages and describe all desired evolutions in the sense of safe regions. Our umbrella notion of a discrete event system (DES) [310] captures these two languages and serves as an abstract representation for the models of the concrete formalisms employed in this thesis.

The following definition introduces the type $\Sigma \text{ des}$ of DES, the constructor DES , and the two selectors L_{um}^{des} and L_m^{des} . The constructor DES takes the unmarked language and the marked language (formally given by sets of lists) over elements from a type Σ to construct a DES. The two selectors L_{um}^{des} and L_m^{des} enable the extraction of the unmarked and the marked languages, respectively, from a DES.

Definition 2.1: «Type of DES»

datatype $\Sigma \text{ des} = DES \quad L_{um}^{des} :: \Sigma \text{ list set} \quad L_m^{des} :: \Sigma \text{ list set}$

We define the predicate *valid-des* to identify the DESs that satisfy the constraints of prefix-closure and inclusion mentioned above.

Definition 2.2: «valid-des»

definition $\text{valid-des} :: \Sigma \text{ des} \Rightarrow \text{bool}$

where $\text{valid-des } D \equiv L_m^{des} D \subseteq L_{um}^{des} D \wedge \text{prefix-closure } (L_{um}^{des} D) = L_{um}^{des} D$

While the separation of the event alphabet into controllable and uncontrollable events is required for controller synthesis, we do not include it in our formalization at this level.

DESs determine a complete lattice according to the instantiation of the abstract operations of a complete lattice as given in the following theorem where we make use of operations from the complete powerset lattice over words of events.

Theorem 2.1: «Discrete Event Systems Form a Complete Lattice»

$$\begin{aligned}
 bot &\equiv DES \{ \} \{ \} \\
 top &\equiv DES \text{ UNIV UNIV} \\
 less\text{-}eq &\equiv \lambda A B. L_{um}^{des} A \subseteq L_{um}^{des} B \wedge L_m^{des} A \subseteq L_m^{des} B \\
 less &\equiv \lambda A B. less\text{-}eq A B \wedge A \neq B \\
 inf &\equiv \lambda A B. DES (L_{um}^{des} A \cap L_{um}^{des} B) (L_m^{des} A \cap L_m^{des} B) \\
 sup &\equiv \lambda A B. DES (L_{um}^{des} A \cup L_{um}^{des} B) (L_m^{des} A \cup L_m^{des} B) \\
 Inf &\equiv \lambda X. DES (\bigcap (L_{um}^{des} \setminus X)) (\bigcap (L_m^{des} \setminus X)) \\
 Sup &\equiv \lambda X. DES (\bigcup (L_{um}^{des} \setminus X)) (\bigcup (L_m^{des} \setminus X))
 \end{aligned}$$

By using this complete lattice later on, we deviate from [282] where a complete lattice over only one of both languages is used to state a supervisory control problem. The adequacy of DESs with their integrated handling of (un)marked languages as abstract representations of concrete formalisms for the purpose of controller synthesis is demonstrated in the subsequent chapters where the complete lattice of DES is used extensively in the definition of the abstract supervisory control problem in chapter 3|p.31 and the definition of our abstract controller synthesis algorithm in chapter 4|p.45.

2.2. EXTENDED PUSHDOWN AUTOMATA

We introduce, amongst others, the automata formalisms of deterministic finite automata (DFA), deterministic pushdown automata (DPDA) [129], simple deterministic pushdown automata (SDPDA) [189], and deterministic extended pushdown automata (EDPDA) as restrictions of our novel formalism of extended pushdown automata (EPDA) from which they inherit various definitions, results, and semantics.

EPDA contain for every reasonable semantics a stack variable in their configurations, which stores words over an additional alphabet of stack elements. It may be used to record information on previously executed steps and to limit step applicability accordingly. For example, counting events and remembering their orderings enables the description of languages containing (a) bracketed/balanced words such as $((()((()))))$ with unbounded nesting depth, (b) words of the form $u^{2 \times n}v^n$ where the event u appears twice as often as the event v , and (c) words of the form wxw^{-1} that are odd-length palindroms with the center element x not contained in w .

We now define EPDA by introducing their type and by providing restrictions that determine the subset of (valid) EPDA. Note, the edges of EPDA used in this definition are defined subsequently.

Definition 2.3: «Type of EPDA and valid-epda»

If Q is a finite set of states, Σ is a finite set of events, Γ is a finite set of stack elements, δ is a finite set of edges, q_0 is an initial state in Q , \square is an end-of-stack element, and F is a finite set of marking states contained in Q , then EPDA G that satisfy *valid-epda* G are of the following form.

$$\langle epda\text{-}states=Q, epda\text{-}events=\Sigma, epda\text{-}\gamma=\Gamma, epda\text{-}\delta=\delta, \\ epda\text{-}initial=q_0, epda\text{-}eos=\square, epda\text{-}marking=F \rangle$$

We do not partition the event alphabet Σ into controllable events Σ_c and uncontrollable events Σ_{uc} by introducing for them two separate record fields to obtain a general formalization that can also be used for other application domains without being cluttered by such a separation. However, we use the terminology of *marking states* from control theory; these states are also called accepting or final states in computer science.

The edges of EPDA (also called step labels), given in the definition above by the set δ , have four different basic shapes. Firstly, each edge contains a source state q_1 and a target state q_2 . Secondly, each edge *may* contain an event a from the set Σ . Thirdly, each edge contains two words of stack elements to be popped and pushed, respectively. These two words *may* contain the end-of-stack element \square only at their end and, moreover, they *must* agree whether it occurs there.

Definition 2.4: «Type of Edges of EPDA and valid-epda-step-label»

If q_1 and q_2 are states from Q , a is an event from Σ , s_1 and s_2 are words over Γ not containing \square , $x = \text{None}$ or $x = \text{Some } a$, and $(w_1, w_2) = (s_1, s_2)$ or $(w_1, w_2) = (s_1 @ [\square], s_2 @ [\square])$, then the edges e of an EPDA G that satisfy *valid-epda-step-label* $G \ e$ are records of the following form.

$$\langle \text{edge-src}=q_1, \text{edge-event}=x, \text{edge-pop}=w_1, \text{edge-push}=w_2, \text{edge-trg}=q_2 \rangle$$

By considering the four cases explicitly we have the following forms.

$$\begin{aligned} &\langle q_1, \text{None}, s_1, s_2, q_2 \rangle \\ &\langle q_1, \text{None}, s_1 @ [\square], s_2 @ [\square], q_2 \rangle \\ &\langle q_1, \text{Some } a, s_1, s_2, q_2 \rangle \\ &\langle q_1, \text{Some } a, s_1 @ [\square], s_2 @ [\square], q_2 \rangle \end{aligned}$$

We now introduce our custom semantics *epdaH* of EPDA by introducing the configurations, the step relation, and the derived notions of determinism as well as the generated marked and unmarked languages.

A configuration is given by a state, a stack of stack elements (which always has a trailing \square and which is to be read from left to right), and a history variable (which is used to log the events executed so far). The initial configuration is constructed by taking the initial state, the stack containing only the \square element, and an empty history.

Definition 2.5: «(Initial) Configurations of an EPDA in epdaH»

If q is a state from Q , s is a word over Γ not containing \square , and h is a word over Σ , then configurations of an EPDA are of the following form.

$$\langle \text{epdaH-conf-state}=q, \text{epdaH-conf-history}=h, \text{epdaH-conf-stack}=s @ [\square] \rangle$$

The unique initial configuration is $\langle q_0, [], [\square] \rangle$.

The step relation defines how a configuration *Pre* can be modified by an edge *Edge* into a configuration *Post*. According to the four kinds of edge patterns from above, we also differentiate between four kinds of steps in the following definition. In the steps 1–4 the state is changed from q_1 to q_2 . In the steps 1 and 2 no event is executed whereas in steps 3 and 4 the event a is executed. In the steps 1 and 3 a strict prefix of the stack is popped (hence, the end-of-stack element \square is not popped) whereas in steps 2 and 4 the entire stack is popped.

Definition 2.6: «Step Relation *epdaH*-step-relation»

If q_1 and q_2 are states from Q , a is an event from Σ , h is a word over Σ , s, s_1 , and s_2 are words over Γ not containing \square , then there are the following kinds of steps.

Step 1	Edge: $\langle q_1, \text{None}, s_1, s_2, q_2 \rangle$ Pre: $\langle q_1, h, s_1 @ s @ [\square] \rangle$ Post: $\langle q_2, h, s_2 @ s @ [\square] \rangle$ Kind: execute no event, do not observe \square at end of stack
Step 2	Edge: $\langle q_1, \text{None}, s_1 @ [\square], s_2 @ [\square], q_2 \rangle$ Pre: $\langle q_1, h, s_1 @ [\square] \rangle$ Post: $\langle q_2, h, s_2 @ [\square] \rangle$ Kind: execute no event, observe \square at end of stack
Step 3	Edge: $\langle q_1, \text{Some } a, s_1, s_2, q_2 \rangle$ Pre: $\langle q_1, h, s_1 @ s @ [\square] \rangle$ Post: $\langle q_2, h @ [a], s_2 @ s @ [\square] \rangle$ Kind: execute event a , do not observe \square at end of stack
Step 4	Edge: $\langle q_1, \text{Some } a, s_1 @ [\square], s_2 @ [\square], q_2 \rangle$ Pre: $\langle q_1, h, s_1 @ [\square] \rangle$ Post: $\langle q_2, h @ [a], s_2 @ [\square] \rangle$ Kind: execute event a , observe \square at end of stack

The iterated application of this step relation results in derivations that are (possibly infinite) sequences of configurations and applied edges. Moreover, initial derivations are those derivations that start in initial configurations and reachable configurations are the configurations in initial derivations.

An EPDA is then deterministic if (during any of its initial derivations) every two distinct steps that are applicable to a common configuration would add different events to the history variable.

Definition 2.7: «Determinism for *epdaH*»

An EPDA is deterministic in *epdaH* if, whenever an initial derivation leads to a configuration c and *epdaH*-step-relation $G c e_1 c_1$ and *epdaH*-step-relation $G c e_2 c_2$ are two applicable steps satisfying *epdaH*-conf-history $c_1 \sqsubseteq \text{epdaH-conf-history } c_2$, then $e_1 = e_2$ and $c_1 = c_2$.

Nondeterministic EPDA are more expressive than deterministic EPDA (EDPDA) because, for example, they can nondeterministically guess the center of palindroms of the form ww^{-1} . However, we expect that nondeterministic specifications lead to nondeterministic controllers that would force the runtime environment to maintain not only one current configuration, which would incur additional undesirable costs for the runtime environment and used hardware. Hence, we do not consider nondeterministic EPDA for modelling specifications or controllers in this thesis.

EDPDA are more expressive than DFA because DFA may not change the stack variable and therefore do not benefit from its existence.

The (un)marked language of an EPDA contains the (un)marked words that are obtained from initial derivations d as follows. The unmarked words are the histories contained in configurations of d . The marked words are the histories contained in configurations c of d where (a) c contains a state that is a marking state of the EPDA and (b) all configurations that follow c in d have the same history. The condition (b) is required to ensure a proper correspondence with the standard semantics $epdaS$ (explained in more detail in section 6.2|p.96).

Definition 2.8: «unmarked-language and marked-language for $epdaH$ »

The unmarked language of an EPDA is given by all words v over Σ such that some configuration c is reachable by some initial derivation d at index n such that $epdaH\text{-}conf\text{-}history\ c = v$.

An unmarked word is also in the marked language of the EPDA when, additionally, $epdaH\text{-}conf\text{-}state\ c$ is a marking state from F and when the history variable is not extended after index n in the derivation d .

Based on the unmarked language and the marked language we define the DES that corresponds to a given EPDA.

Definition 2.9: «Conversion of EPDA Using $epda\text{-}to\text{-}des$ to DES»

If G is an EPDA, then $epda\text{-}to\text{-}des\ G$ is a DES defined as

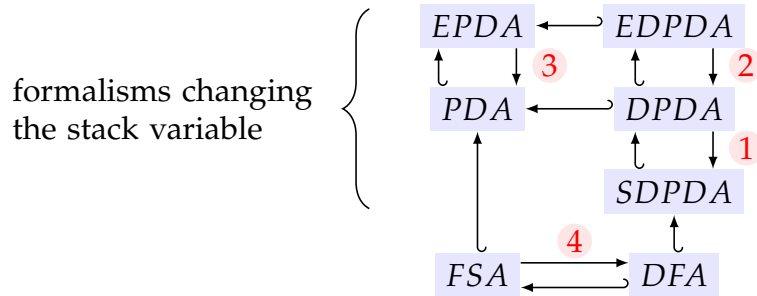
$$DES\ (epdaH.unmarked\text{-}language\ G)\ (epdaH.marked\text{-}language\ G).$$

We obtain the other automata formalisms as follows. PDA are EPDA where the *edge-pop* component has length 1, FSA are PDA where the stack variable is never changed, DPDA are deterministic PDA (we use the predicate *valid-dpda*), and DFA are deterministic FSA. Moreover, SDPDA are DPDA (we use the predicate *valid-sdpda*) with only three kinds of edges: edges executing a single event while not modifying the stack, edges pushing a single element to the stack, and edges popping the single top-stack element.

The following figure visualizes the relationship between these automata formalisms. The edges labelled with 1 and 2 are implemented by operations from our concrete controller synthesis algorithm introduced in chapter 5|p.61, the edge 3 is subsumed by our reasoning for edge 2, and the edge 4 is included only for completeness and is not part of our formalization.

Figure 2.1: «Relationship Between Automata Formalisms»

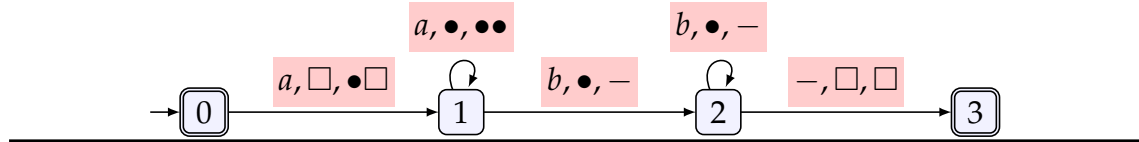
Notation: $A \hookrightarrow B$ denotes that A is a subclass of B and $A \rightarrow B$ denotes that every $a \in A$ can be translated into an equivalent $b \in B$.



Finally, we provide an example of a DPDA for which no equivalent DFA exists.

Example 2.1: «DPDA Example»

The visualized DPDA G generates the marked language $\{ a^n b^n \cdot n \in \mathbf{N} \}$ and the unmarked language $\{ a^n b^m \cdot n, m \in \mathbf{N}, m \leq n \}$. For this DPDA, the generated unmarked language is the prefix closure of the generated marked language, formally, $epdaH.unmarked-language\ G = prefix-closure\ (epdaH.marked-language\ G)$. Hence, every unmarked word can be extended to a marked word.



2.3. PARSERS

Parsing theory is a fundamental and well established research domain in the field of computer science. Many Parser constructions such as LR(k)-Parsers have been developed for the parsing of programming languages. In this thesis we rely on the general notion of Parsers as presented in [325], which relies on [189].

We consider two kinds of semantics in this thesis: branching and linear semantics (see Par. *Branching and Linear Semantics*|p.105 for more details). Branching semantics such as the semantics $epdaH$ above for EPDA have a unique initial configuration from which every unmarked word can be executed. Linear semantics have one initial configuration for each potential unmarked word that is contained in the configuration in the form of a scheduler variable to be followed.

For modelling, Parsers have three features that are not available in EPDA.

Partial Execution: Parsers have a refined event mechanism compared to EPDA that decouples the observation of events (by a context) and their execution (by the Parser). That is, Parsers may execute only a prefix of the events observed in a step whereas EPDA execute an event in the step in which it is observed. The fixed scheduler variable, handled implicitly or explicitly in Parser semantics, consists of the observed events that have not been executed so far. These events have to be executed before any subsequently observed events. Also, in parsing theory, the term *look ahead* stands for the list of events (or its length) the Parser can observe in a single step. A longer look ahead allows for more concise Parsers because the look ahead restricts rule applicability as well: the look ahead must be compatible with the fixed scheduler as mentioned above and, in linear semantics, also with the scheduler. From this perspective each EPDA has a look ahead of at most 1 because the event component of an edge of an EPDA has a length of at most 1.

Parsers making use of this feature are intermediately obtained by our concrete controller synthesis algorithm and are translated into Parsers not using this feature by changing the stack variables such that they contain the fixed scheduler variable explicitly (see F2 in section 5.2.3|p.78).

Multiple Event Generation: Parsers may execute/observe any (finite) number of events in a single step whereas EPDA only allow the execution of at most one event at a time.

However, apart from some results from parsing theory, we assume in many of our proofs that the Parser at hand executes at most one event per step.

Processing Terminator: Parsers may fix a special end-of-input event *parser-eoi* (also called processing terminator) in their steps. Fixing this special event prevents the observation of further events in subsequent steps. That is, in a branching semantics, the Parser may *decide* to stop the observation of further events by fixing the *parser-eoi* event and, in a linear semantics, the Parser may *detect* the *parser-eoi* event at the end of the scheduler variable. In both cases the Parser may still execute previously observed and not yet executed events and may also operate on the stack.

Parsers observing the processing terminator are intermediately obtained by our concrete controller synthesis algorithm. We remove this usage without invalidating desired semantical properties (see F1 in subsection 5.2.2|p.73).

In our concrete controller synthesis algorithm in chapter 5|p.61, we obtain instances of LR(1)-Parsers, which are a subclass of deterministic Parsers. Then, we verify that these LR(1)-Parsers can be translated into DPDA while preserving the satisfaction of semantical properties that are of interest for controller synthesis.

We define Parsers, which make use of the notion of rules defined below.

Definition 2.10: «Type of Parser and valid-parser»

If N is a finite set of nonterminals, Σ is a finite set of events, q_0 is an initial nonterminal in N , F is a finite set of marking nonterminals from N , δ is a finite set of rules, and $\$$ is the end-of-input event from Σ , then Parsers G that satisfy *valid-parser* G are of the following form.

$$\langle \begin{array}{ll} \text{parser-nonterms} = N, & \text{parser-events} = \Sigma, \\ \text{parser-initial} = q_0, & \text{parser-marking} = F, \\ \text{parser-rules} = \delta, & \text{parser-eoi} = \$ \end{array} \rangle$$

The rules of Parsers, given above by the set δ , have two different shapes. The first shape is used to change the current stack by exchanging a certain prefix of it, to execute the events of a word v_1 , and to fix the events of v_2 that have not been fixed in an earlier step. The second shape is similar, but it fixes the processing terminator if it has not been fixed in an earlier step and, hence, after applying such a rule only the events from v_2 can be executed subsequently.

Definition 2.11: «Type of Rules of Parsers and valid-parser-step-label»

If q_1 and q_2 are nonterminals from N , s_1 and s_2 are words over N , v_1 and v_2 are words over Σ not containing $\$$, and $(w_1, w_2) = (s_1 @ s_2, s_2)$ or $(w_1, w_2) = (s_1 @ s_2 @ [\$], s_2 @ [\$])$, then the rules r of a Parser G that satisfy *valid-parser-step-label* $G r$ are of the following form.

$$\langle \text{rule-stack-pop} = s_1 @ [q_1], \text{rule-scheduler-pop} = w_1, \\ \text{rule-stack-push} = s_2 @ [q_2], \text{rule-scheduler-push} = w_2 \rangle$$

By considering the two cases explicitly we have the following forms.

$$\langle s_1 @ [q_1], v_1 @ v_2, s_2 @ [q_2], v_2 \rangle \\ \langle s_1 @ [q_1], v_1 @ v_2 @ [\$], s_2 @ [q_2], v_2 @ [\$] \rangle$$

The elements q_1 and q_2 may be interpreted as the source and target state of the rule, which are not separately stored as in the edges of EPDA. We also use the notation $x_1 | y_1 \rightarrow x_2 | y_2$ for rules $\langle x_1, y_1, x_2, y_2 \rangle$ from [325].

We now introduce our custom semantics *parserHF* of Parsers by introducing the configurations, the step relation, and the derived notions of determinism as well as the generated unmarked and marked languages. The more simple and equivalent semantics *parserS* is explained in subsection 6.2.7|p.114.

A configuration is given by a stack of nonterminals, which is never empty and which is to be read from *right to left*, a history for storing the already observed events, which is extended to the right, and a fixed scheduler of the events observed but not yet executed. The initial configurations are constructed using the initial stack element, an empty history, and an empty fixed scheduler.

Definition 2.12: «(Initial) Configurations of a Parser in parserHF»

If s is a word over N , q is from N , v_1 and v_2 are words over Σ not containing $\$$, $h = v_1 @ v_2$, and $f = v_2$ or $f = v_2 @ [\$]$ then configurations of a Parser are of the following form.

$$\langle \text{parserHF-conf-fixed} = f, \text{parserHF-conf-history} = h, \text{parserHF-conf-stack} = s @ [q] \rangle$$

By considering the two cases explicitly we have the following forms.

$$\langle v_2, v_1 @ v_2, s @ [q] \rangle \\ \langle v_2 @ [\$], v_1 @ v_2, s @ [q] \rangle$$

The unique initial configuration is $\langle [], [], [q_0] \rangle$.

The step relation determines how a configuration *Pre* can be modified by a rule *Rule* into a configuration *Post*. We distinguish between seven kinds of steps due to the interplay between fixed scheduler and history variable. In all seven steps the stack is changed by replacing the top-most elements $s_1 @ [q_1]$ by $s_2 @ [q_2]$. Also, in each of the seven steps the rule must respect the fixed scheduler in the sense that the component *rule-scheduler-pop* of the rule is either a prefix of the fixed scheduler of the *Pre*-configuration or has this fixed scheduler as a prefix.

Definition 2.13: «Step Relation parserHF-step-relation»

If q_1 and q_2 are from N , s , s_1 , and s_2 are words over N , h , v_1 , v_2 , and v_3 are words over Σ not containing $\$$, and v_1 , v_2 , and v_3 are not empty, then there are the following kinds of steps.

- | | |
|---------|---|
| Step 1a | Rule: $\langle s_1 @ [q_1], v_1 @ v_2 @ v_3, s_2 @ [q_2], v_3 \rangle$
Pre: $\langle v_1, h, s @ s_1 @ [q_1] \rangle$
Post: $\langle v_3, h @ v_2 @ v_3, s @ s_2 @ [q_2] \rangle$
Kind: $(\neg A), (\neg B), (\neg C), (D), (E)$ |
| Step 1b | Rule: $\langle s_1 @ [q_1], v_1 @ v_2 @ v_3, s_2 @ [q_2], v_2 @ v_3 \rangle$
Pre: $\langle v_1 @ v_2, h, s @ s_1 @ [q_1] \rangle$
Post: $\langle v_2 @ v_3, h @ v_3, s @ s_2 @ [q_2] \rangle$
Kind: $(\neg A), (\neg B), (\neg C), (D), (\neg E)$ |
| Step 1c | Rule: $\langle s_1 @ [q_1], v_1 @ v_2, s_2 @ [q_2], v_2 \rangle$
Pre: $\langle v_1 @ v_2 @ v_3, h, s @ s_1 @ [q_1] \rangle$
Post: $\langle v_2 @ v_3, h, s @ s_2 @ [q_2] \rangle$
Kind: $(\neg A), (\neg B), (\neg C), (\neg D), (\neg E)$ |
| Step 2a | Rule: $\langle s_1 @ [q_1], v_1 @ v_2 @ v_3 @ [\$], s_2 @ [q_2], v_3 @ [\$] \rangle$
Pre: $\langle v_1, h, s @ s_1 @ [q_1] \rangle$
Post: $\langle v_3 @ [\$], h @ v_2 @ v_3, s @ s_2 @ [q_2] \rangle$
Kind: $(\neg A), (B), (C), (D), (E)$ |
| Step 2b | Rule: $\langle s_1 @ [q_1], v_1 @ v_2 @ v_3 @ [\$], s_2 @ [q_2], v_2 @ v_3 @ [\$] \rangle$
Pre: $\langle v_1 @ v_2, h, s @ s_1 @ [q_1] \rangle$
Post: $\langle v_2 @ v_3 @ [\$], h @ v_3, s @ s_2 @ [q_2] \rangle$
Kind: $(\neg A), (B), (C), (D), (\neg E)$ |
| Step 3a | Rule: $\langle s_1 @ [q_1], v_1 @ v_2, s_2 @ [q_2], v_2 \rangle$
Pre: $\langle v_1 @ v_2 @ v_3 @ [\$], h, s @ s_1 @ [q_1] \rangle$
Post: $\langle v_2 @ v_3 @ [\$], h, s @ s_2 @ [q_2] \rangle$
Kind: $(A), (B), (\neg C), (\neg D), (\neg E)$ |
| Step 3b | Rule: $\langle s_1 @ [q_1], v_1 @ v_2 @ [\$], s_2 @ [q_2], v_2 @ [\$] \rangle$
Pre: $\langle v_1 @ v_2 @ [\$], h, s @ s_1 @ [q_1] \rangle$
Post: $\langle v_2 @ [\$], h, s @ s_2 @ [q_2] \rangle$
Kind: $(A), (B), (C), (\neg D), (\neg E)$ |

Legend: We use the following classification of the steps also explained below.

- (A) $\$$ is fixed in the Pre-configuration (i.e., *parserHF-conf-fixed* ends with $\$$)
- (B) $\$$ is fixed in the Post-configuration (i.e., *parserHF-conf-fixed* ends with $\$$)
- (C) $\$$ is observed by the rule (i.e., *rule-scheduler-pop* ends with $\$$)
- (D) new events are executed (i.e., new events are added to the history)
- (E) the fixed scheduler of the Pre-configuration is entirely removed

Firstly, the labels (A)–(C) are related to the status and the observation of the processing terminator: the processing terminator $\$$ is not fixed in the Pre- and Post-configurations of steps 1a–1c and in the Pre-configurations of steps 2a and 2b. Vice versa, the processing terminator $\$$ is fixed in the Post-configurations of steps 2a and 2b and in the Pre- and Post-configurations of steps 3a and 3b. Secondly, the label (D) is related to the execution of further events: in steps 1a, 1b, 2a, and 2b further events are executed and possibly fixed. Finally, the label (E) is related to the removal of the fixed scheduler: in steps 1a and 2a the entire fixed scheduler is removed whereas in the other steps 1b, 1c, 2b, 3a, and 3b a nonempty suffix of the fixed scheduler is retained.

Intuitively a given Parser is deterministic if (during any initial derivation) every two distinct applicable steps would add different events to the history variable.

Definition 2.14: «Determinism for parserHF»

A Parser is deterministic in the semantics *parserHF* if, whenever a given initial derivation reaches a certain configuration c and *parserHF*-step-relation $G \ c \ e_1 \ c_1$ and *parserHF*-step-relation $G \ c \ e_2 \ c_2$ are two applicable steps that satisfy that *parserHF*-conf-history $c_1 \sqsubseteq \text{parserHF-conf-history } c_2$, then $e_1 = e_2$ and $c_1 = c_2$.

The (un)marked language of a Parser contains the (un)marked words that are obtained from initial derivations d as follows. The unmarked words are the histories contained in configurations of d . The marked words are the histories contained in configurations c of d where (a) the stack of c ends in a marking stack element of the Parser and (b) all configurations that follow c in d have the same history. As for EPDA, the condition (b) is required to ensure a proper correspondence with the standard semantics *parserS* (explained in more detail in section 6.2[p.96]).

Definition 2.15: «unmarked-language and marked-language for parserHF»

The unmarked language of a Parser is given by all words v over Σ such that some configuration c is reachable by some initial derivation d at index n such that *parserHF*-conf-history $c = v$.

For a word to be in the marked language of the Parser we additionally require that *parserHF*-conf-stack c is of the form $s @ [q]$ where q is a marking stack element from F and that the history variable is not extended after index n .

Based on the unmarked language and the marked language we define the DES that corresponds to a given Parser.

Definition 2.16: «Conversion of Parsers Using parser-to-des to DES»

If G is a Parser, then *parser-to-des* G is a DES defined as

$$DES(\text{parserHF.unmarked-language } G) (\text{parserHF.marked-language } G).$$

In the following example, we present a Parser that does not rely on any of the three Parser-specific features introduced above.

Example 2.2: «Parser Example»

The Parser with the initial stack element 0, the marking stack elements $\{0, 3\}$, and the five rules below has the same unmarked and marked language as the EPDA from Example 2.1|p.22.

$r_1 : 0 \mid a \longrightarrow \square \bullet 1 \mid$	$r_4 : \bullet 2 \mid b \longrightarrow 2 \mid$
$r_2 : \bullet 1 \mid a \longrightarrow \bullet \bullet 1 \mid$	$r_5 : \square 2 \mid \longrightarrow 3 \mid$
$r_3 : \bullet 1 \mid b \longrightarrow 2 \mid$	

2.4. CONTEXT-FREE GRAMMARS

In this section, we introduce the well-known formalism of context-free grammars (CFGs) from [80] and their subclass of LR(k)-CFGs from [189].

Intuitively, CFGs are rule-based replacement systems operating on words over two kinds of elements: events (also called terminals in other contexts) and nonterminals. In each step (of any semantics of CFGs) a single nonterminal that is contained in the configuration is replaced by a (possibly empty) replacement list according to one production of the CFG.

The parsing of programming languages, described by LR(k)-CFG, using LR(k)-Parsers was introduced in [189] and more technical details and results for LR(k)-CFG have been summarized in a consistent notation in [325]. In particular, the LR(k)-CFGs can be parsed deterministically with look ahead k . Also, these Parsers are *correct-prefix-parsers* [325, Volume II, Theorem 9.1, p. 291] stating that errors are detected when they occur, which (a) supports users in finding the syntax errors in their program to be parsed and which (b) is fundamentally required in our control setting as it allows us to execute *only* the desired events.

In the fields of computer science and discrete control theory, the formalism of CFG is not commonly used for the description of running systems. One reason may be that the definition of LR(1)-CFG is more complex compared to the definitions of determinism for formalisms such as DPDA resulting in a more difficult modelling processes of deterministic systems. The formalism of LR(1)-CFG is used in our concrete controller synthesis algorithm from chapter 5|p.61 where we also provide details on their equivalence to DPDA and LR(1)-Parsers.

We now define CFG already using productions that are defined subsequently.

Definition 2.17: «Type of CFG and valid-cfg»

If N is a finite set of nonterminals, Σ is a finite set of events, S is an initial nonterminal in N , and P is a finite set of productions, then CFGs G that satisfy *valid-cfg* G are of the following form.

$$\langle \text{cfg-nonterminals} = N, \text{cfg-events} = \Sigma, \text{cfg-initial} = S, \text{cfg-productions} = P \rangle$$

As for EPDA and Parsers, we do not partition the event alphabet Σ into controllable events Σ_c and uncontrollable events Σ_{uc} by introducing for them two separate record fields to obtain a general formalization of CFGs.

The productions of CFGs, given above by the set P , have only one shape. In general, a production is given by a left-hand-side nonterminal and a right-hand-side word over nonterminals and events of the CFG. Formally, we distinguish between nonterminals and events by using words ranging over a custom datatype where the elements beA A and beA b represent nonterminals A and events b , respectively. Informally, we omit the constructors beA and beB and use capital and lower case letters for nonterminals and events, respectively. Given a CFG with a set of nonterminals N and a set of events Σ we define the set of elements of such words by *bi-elem-domain* $N \Sigma = \{ beA A \mid A . A \in N \} \cup \{ beB b \mid b . b \in \Sigma \}$.

Definition 2.18: «Type of Production of CFGs and valid-cfg-step-label»

If A is a nonterminal from N , w is a word over *bi-elem-domain* $N \Sigma$, then the productions p of a CFG G that satisfy *valid-cfg-step-label* $G p$ are of the following form.

$$\langle prod-lhs = A, prod-rhs = w \rangle$$

We introduce subsequently the three semantics *cfgSTD*, *cfgLM*, and *cfgRM* for CFG. They share identical definitions for the (initial) configurations, the generated unmarked language, and the generated marked language, but they differ in their step-relations as explained below.

A configuration of a CFG is given by a word over the set *bi-elem-domain* $N \Sigma$ and the initial configuration is constructed by taking only the initial nonterminal S .

Definition 2.19: «(Initial) Configurations of a CFG»

If w is a word over *bi-elem-domain* $N \Sigma$, then the configurations of a CFG are of the following form.

$$\langle cfg-conf=w \rangle$$

The unique initial configuration is $\langle [beA S] \rangle$.

The three CFG semantics are branching semantics as they have a unique initial configuration, which does not determine the unmarked word to be executed from this initial configuration.

The step relations of the three semantics determine how a given *Pre* configuration can be modified by a production *Production* into a configuration *Post*. We use the operation *lift-B* to apply the constructor beB to each element in a word over Σ .

Definition 2.20: «Step Relations *cfgSTD*-step-relation, *cfgLM*-step-relation, and *cfgRM*-step-relation»

If w_1 , w_2 , and w are words over *bi-elem-domain* $N \Sigma$ and v is a word over Σ , then steps are of the following form.

<i>cfgSTD</i>	Production:	$\langle A, w \rangle$
	Pre:	$\langle w_1 @ [A] @ w_2 \rangle$
	Post:	$\langle w_1 @ w @ w_2 \rangle$
	Kind:	replace any occurrence of the nonterminal A

<i>cfgLM</i>	Production:	$\langle A, w \rangle$
	Pre:	$\langle (lift-B\ v) @ [A] @ w_1 \rangle$
	Post:	$\langle (lift-B\ v) @ w @ w_1 \rangle$
	Kind:	replace an occurrence of the nonterminal A , if it not preceded by another nonterminal
<i>cfgRM</i>	Production:	$\langle A, w \rangle$
	Pre:	$\langle w_1 @ [A] @ (lift-B\ v) \rangle$
	Post:	$\langle w_1 @ w @ (lift-B\ v) \rangle$
	Kind:	replace an occurrence of the nonterminal A , if it not succeeded by another nonterminal

Because all three step relations depend on a nonterminal to be present in the Pre-configuration, there are no steps applicable to a configuration not containing any nonterminals.

The notion of determinism cannot be determined for CFG by checking local structural conditions as for EPDA and Parsers. Instead the definition depends on the notion of initial derivations. Consider the following simplified example from [325, Volume II, Figure 6.9, p. 49] of a CFG that we consider to be not deterministic.

Example 2.3: «A Nondeterministic CFG»

We consider the CFG with initial nonterminal S and the following four rules.

$$\rho_1 : \langle S, [A, B, b] \rangle \quad \rho_2 : \langle A, [a, b] \rangle \quad \rho_3 : \langle S, [a, B, b] \rangle \quad \rho_4 : \langle B, [b] \rangle$$

Discussion: There are initial derivations d_1 and d_2 that end in the configurations $c_1 = \langle [a, b, b, b] \rangle$ and $c_2 = \langle [a, b, b] \rangle$, respectively. Determinism would require that both derivations apply the same productions until reaching configurations having $[a, b, b]$ as a prefix. However, this is not the case for the given example where different productions are applied in the two derivations.

Note, the reasoning employed in this example corresponds closely to a similar explanation of determinism for EPDA. That is, if an EDPDA generates the unmarked words $[a, b, b, b]$ and $[a, b, b]$, then there is (using the semantics *epdaH*) a *unique* initial derivation that appends the event b to the history $[a, b]$ in its last step. Note that special attention is required when formalizing determinism along the lines of these intuitive explanations because EPDA and CFG allow for steps not executing events. Determinism for CFG by means of the notion of LR(1)-CFG is given in Definition 6.9|p.121 (where we analyze the relationship between DPDA and LR(1)-CFG w.r.t. determinism) and is considered in more detail also in subsection 5.2.1|p.69.

Finally, the marked and unmarked languages of a CFG are obtained from its initial derivations d . Each nonterminal-free prefix of the word given in a configuration c is an unmarked word of the CFG. Moreover, the marked language is given by the words contained in configurations that contain no further nonterminals.

Definition 2.21: «unmarked-language and marked-language for CFG»

The unmarked language of a CFG is given by all words v over Σ such that some configuration c is reachable by some initial derivation d such that $\text{lift-B } v$ is a prefix of $\text{cfg-conf } c$.

For a word to be in the marked language of the CFG we require that $\text{lift-B } v$ equals $\text{cfg-conf } c$.

Based on the unmarked language and the marked language we define the DES that corresponds to a given CFG.

Definition 2.22: «Conversion of CFG Using cfg-to-des to DES»

If G is a CFG, then $\text{cfg-to-des } G$ is a DES defined as

$$\text{DES } (\text{cfgLM.unmarked-language } G) \text{ } (\text{cfgLM.marked-language } G).$$

For the equivalence of the three semantics we note that initial derivations ending in configurations without nonterminals can be translated among the three semantics by reordering the application of productions. For cfgLM and cfgRM these initial derivations are also unique. However, initial derivations that end in configuration containing nonterminals cannot be translated among the semantics: a piecewise translation from initial cfgRM derivations to initial cfgLM derivations is considered for this case in the proof presented in section 6.3|p.116.

Finally, we end our introduction by giving an example of a CFG.

Example 2.4: «CFG Example»

The CFG with initial nonterminal S and the following two productions has the same unmarked and marked language as the EPDA and the Parser from Example 2.1|p.22 and Example 2.2|p.27, respectively.

$$\rho_1 : \langle S, [a, S, b] \rangle \quad \rho_2 : \langle S, [] \rangle$$

Chapter 2|p.15

(Abstract and Concrete Discrete Event Systems)

In this chapter, we introduced the abstract formalism of DES and the concrete formalisms of EPDA, Parsers, and CFG. A DES is a semantical abstraction of an instance of a concrete formalism and includes the marked and unmarked language of that instance.

DES are the foundation for our definitions of the abstract supervisory control problem and an abstract controller synthesis algorithm in chapter 3|p.31 and chapter 4|p.45, respectively. DPDA and DFA, which are subclasses of EPDA, are used to define a concrete supervisory control problem and a concrete controller synthesis algorithm in chapter 3|p.31 and chapter 5|p.61, respectively. The other concrete formalisms of Parsers and CFGs introduced here are used in this concrete controller synthesis algorithm as well to represent intermediate controller candidates.

3

Abstract and Concrete Supervisory Control Problems

In this chapter, we introduce an abstract and a concrete supervisory control problem to be solved by our abstract and concrete controller synthesis algorithms from chapter 4|p.45 and chapter 5|p.61, respectively. As a foundation for these problems, we use the complete lattice of DES and the automata formalisms from section 2.1|p.17 and section 2.2|p.18, respectively. In section 3.3|p.41, we show that both supervisory control problems are compatible under the expected limitations.

We employ DES for the characterization of the least restrictive satisfactory controller to be obtained as stated by the abstract supervisory control problem. While our definition of this abstract problem follows earlier definitions, we believe that it is an improvement compared to the characterization presented in [281] because it defines the unmarked and marked languages of the controller to be constructed together, which is important to state properties referring to both languages and, hence, also for the connection to the abstract controller synthesis algorithm and the concrete controller synthesis algorithm. Moreover, it cleanly separates the abstract and the concrete setting, which allows for a separate handling of aspects that are *only* relevant in the concrete setting such as for example the absence of livelocks. For comparison, the supremal controllable unmarked language contained in the unmarked language of the plant was defined based on the complete powerset lattice of unmarked words over the event alphabet in [281, Section 2, p. 1073] while the marked language and nonblockingness were handled separately in [282, Section 3, p. 482].

Note that the extension of controller synthesis algorithms to language classes beyond regular languages was mentioned in [281, Section 4, p. 1074] as a possible research goal already.

CONTENTS OF THIS CHAPTER

3.1/p.33 *Abstract Supervisory Control Problem for Discrete Event Systems*

We introduce the properties of controllability, nonblockingness, and satisfaction of specifications that are then used to define the abstract supervisory control problem for DES plants and DES specifications. Additionally, we consider minimality for DES controllers.

3.2/p.36 *Concrete Supervisory Control Problem for Deterministic Pushdown Automata*

We introduce the properties of operational controllability, operational nonblockingness, operational satisfaction of specifications, deadlock freedom, and livelock freedom that are then used to define the concrete supervisory control problem for DFA plants and DPDA specifications. Additionally, we consider two minimality notions and two accessibility notions for DPDA controllers.

3.3/p.41 *Correspondence between Abstract and Concrete Supervisory Control Problems*

We compare the two supervisory control problems and show that the concrete supervisory control problem is a refinement of the abstract supervisory control problem under the expected limitations.

3.1. ABSTRACT SUPERVISORY CONTROL PROBLEM FOR DISCRETE EVENT SYSTEMS

Following our introduction from chapter 1|p.1, we introduce the abstract supervisory control problem. For this purpose, we rely on the abstract formalism of DES from section 2.1|p.17 to model the controllers, plants, specifications, and closed loops and introduce required constructions and properties on DES.

—• Construction of the Closed Loop

Recall that the closed loop CL of a plant P and a controller C is defined by their synchronous composition. Because the two systems can only execute events together (that is, send and receive operations are coupled), only those words of events are preserved from P and C that are common to them. Hence, C can prevent parts of P , but it cannot add further sequences to the common behavior. The synchronous composition is constructed by the *inf* operation from the complete lattice of DES, which constructs the intersection of the (un)marked languages of P and C . That is, $\text{inf } P \ C = \text{DES } (L_{um}^{des} P \cap L_{um}^{des} C) \ (L_m^{des} P \cap L_m^{des} C)$.

—• Property of Controllability

For a proper synchronization between the two components, we require that the controller C must not refuse the reception of any event $u \in \Sigma_{uc}$ that is send by the plant P . Controllers not satisfying this property are generally not desirable as they are capable of ignoring events sent by the plant that e.g. unveil an undesired plant behavior. This property is also known in computer science where I/O automata [227] are required to be *input-enabled*.

Formally, we state that whenever C may execute w and P may execute $w @ [u]$ for an uncontrollable event u , then C must be able to execute $w @ [u]$ as well.

Definition 3.1: «DES-controllable»

definition $\text{DES-controllable} :: \Sigma \text{ des} \Rightarrow \Sigma \text{ set} \Rightarrow \Sigma \text{ des} \Rightarrow \text{bool}$

where $\text{DES-controllable } C \ \Sigma_{uc} \ P$

$$\equiv \forall w \ u. (w \in L_{um}^{des} C \wedge u \in \Sigma_{uc} \wedge w @ [u] \in L_{um}^{des} P) \longrightarrow w @ [u] \in L_{um}^{des} C$$

—• Property of Nonblockingness

The unmarked language and the marked language of the closed loop CL describe a safe region and a goal region, respectively. Alternatively, from a task-oriented perspective, they can be understood to describe *tasks* and *completed tasks*, respectively. Nonblockingness then states that every started task can be completed or, alternatively, that the goal region is permanently reachable from every part of the safe region. Formally, each unmarked word w of the closed loop must be extendable by some word v resulting in a marked word of the closed loop.

Definition 3.2: «DES-nonblocking»

definition $\text{DES-nonblocking} :: \Sigma \text{ des} \Rightarrow \text{bool}$

where $\text{DES-nonblocking } CL \equiv L_{um}^{des} CL \subseteq \text{prefix-closure } (L_m^{des} CL)$

→ *Property of Satisfaction of the Specification*

We state safety and permanent reachability properties by using specifications in the form of DES that are to be satisfied by the closed loop CL . On the one hand, the set $L_{um}^{des} S$ characterizes the safe region that is not to be exited throughout the execution of the closed loop. On the other hand, the set $L_m^{des} S$ characterizes the goal region that is desirable to be reached. We define that a closed loop CL satisfies a specification S whenever the unmarked and marked languages of CL are contained in those of S . This is stated for DES by making use of the *less-eq* relation of our complete lattice of DES from section 2.1|p.17.

Definition 3.3: «DES-satisfaction»

definition $DES\text{-satisfaction} :: \Sigma des \Rightarrow \Sigma des \Rightarrow bool$

where $DES\text{-satisfaction } S \ CL \equiv CL \leq S$

Note, nonblockingness and specification satisfaction together imply that $L_{um}^{des} C$ is a subset of *prefix-closure* ($L_m^{des} S$). This expresses, again from the task-oriented perspective, that each task of the closed loop can be extended to a completed task of the specification.

→ *Abstract Supervisory Control Problem for Discrete Event Systems*

The supervisory control problem describes, for a given plant P and a given specification S , a set of desirable controllers C . In this abstract setting, we use (valid) DES to describe the plant, specification, controller, and closed loop.

We determine the controllers that may be used to safely control the plant by collecting all valid DES satisfying the three properties introduced above.

Definition 3.4: «SCP-Controller-Satisfactory»

definition $SCP\text{-Controller-Satisfactory} :: \Sigma des \Rightarrow \Sigma des \Rightarrow \Sigma set \Rightarrow \Sigma des set$

where $SCP_{Sat}^C P S \Sigma_{uc}$
 $\equiv \{ C . \text{valid-des } C$
 $\quad \wedge DES\text{-controllable } \Sigma_{uc} P C$
 $\quad \wedge DES\text{-nonblocking } (inf C P)$
 $\quad \wedge DES\text{-satisfaction } S (inf C P) \}$

Based on these satisfactory controllers, we derive the satisfactory closed loops $inf C P$ for which a satisfactory controller C has been used.

Definition 3.5: «SCP-Closed-Loop-Satisfactory»

definition $SCP\text{-Closed-Loop-Satisfactory} :: \Sigma des \Rightarrow \Sigma des \Rightarrow \Sigma set \Rightarrow \Sigma des set$

where $SCP_{Sat}^{CL} P S \Sigma_{uc} \equiv \{ inf C P \mid C . C \in SCP_{Sat}^C P S \Sigma_{uc} \}$

Firstly, the set $SCP_{Sat}^C P S \Sigma_{uc}$ is never empty because $L_{um}^{des} C$ may be empty without ever violating controllability. While this can be understood to be a modelling problem (the non-emptiness could be included in the definition of a valid DES), we follow here the definitions in [281] (see also section 3.3|p.41 for further consequences). Secondly, all controllers contained in $SCP_{Sat}^C P S \Sigma_{uc}$ are

equally satisfactory w.r.t. the properties discussed and result therefore in a *safe* closed loop behavior by guaranteeing for example the absence of violations of the specification.

The various controllers differ in the extend to which they limit the plant behavior. For example, two controllers may be satisfactory while only one of them is desirable as it actually activates the plant from an initial idle state by sending a suitable start-up event. Hence, we restrict the set $SCP_{Sat}^C P S \Sigma_{uc}$ to its members that are least restrictive w.r.t. the plant P . A controller C is least restrictive if C is satisfactory, the closed loop of C and P is the supremal satisfactory closed loop, and this supremal closed loop is a satisfactory closed loop as well. The notion of least restrictiveness also ensures that further synchronous controllers can be used in parallel to restrict the behavior of the closed loop.

Definition 3.6: «SCP-Controller-Least-Restrictive»

definition *SCP-Controller-Least-Restrictive*

$$:: \Sigma \text{ des} \Rightarrow \Sigma \text{ des} \Rightarrow \Sigma \text{ set} \Rightarrow \Sigma \text{ des set}$$

where $SCP_{LR}^C P S \Sigma_{uc}$

$$\begin{aligned} \equiv \{ C . C \in SCP_{Sat}^C P S \Sigma_{uc} \\ \wedge \inf C P = \sup (SCP_{Sat}^{CL} P S \Sigma_{uc}) \\ \wedge \inf C P \in SCP_{Sat}^{CL} P S \Sigma_{uc} \} \end{aligned}$$

Indeed, this definition of the set of controllers $SCP_{LR}^C P S \Sigma_{uc}$ can be simplified to $\{ C . \text{valid-des } C \wedge \inf C P = \sup (SCP_{Sat}^{CL} P S \Sigma_{uc}) \}$ because the four properties used in SCP_{Sat}^C are closed under the supremum (that is, closed under arbitrary union), $\sup (SCP_{Sat}^{CL} P S \Sigma_{uc})$ is contained in $SCP_{Sat}^{CL} P S \Sigma_{uc}$, and controllability of C can be derived from the controllability of $\inf C P$.

With the definitions introduced and this final simplification, we state the abstract supervisory control problem for DES.

Definition 3.7: «Abstract Supervisory Control Problem for DES»

Given a set of events Σ , which is partitioned into controllable events Σ_c and uncontrollable events Σ_{uc} , a plant P in the form of a valid DES over Σ , and a specification S in the form of a valid DES over Σ , *determine* a controller in the form of a valid DES over Σ from $SCP_{LR}^C P S \Sigma_{uc}$.

We discussed and related further alternative notions of controllability and the abstract supervisory control problem in [310].

→ *Additional Property of Minimality for DES*

The set of solutions to the abstract supervisory control problem for DES is in general no singleton. For example, we may require the controller C to be the smallest solution, that is, we may require that the controller C equals $\inf (SCP_{LR}^C P S \Sigma_{uc})$. However, without a concrete description language for DES at hand, the infimal solution is not arguably more desirable than the supremal solution (see Par. *Additional Property of Minimality for DPDA* [p.39]).

3.2. CONCRETE SUPERVISORY CONTROL PROBLEM FOR DETERMINISTIC PUSHDOWN AUTOMATA

We introduce a characterization of the least restrictive satisfactory controllers for the setting of DFA plants and DPDA specifications. These controllers are DPDA and ought to satisfy operational properties to be introduced subsequently. However, the formal definitions for the properties are omitted here to increase readability and are contained in Appendix C|p.233. In the subsequent section, we compare these DPDA controllers with the least restrictive satisfactory DES controllers from the previous section.

— Construction of the Closed Loop

The synchronous execution of a DFA plant and a DPDA controller, where both components synchronize on their common events, is defined by the usual product construction $F_{DPDA-DFA-Product}$ of DFA and DPDA (see also [163, Theorem 7.27, p.286]) and is discussed in more detail in section 5.1|p.64. In fact, the observation of the feasibility of this construction triggered the entire research project at its earliest stage.

— Property of Determinism

We require controllers to be deterministic, that is, in this setting controllers should be DPDA rather than nondeterministic PDA. Determinism means here that the DPDA controller C always has a unique successor configuration for each possible extension of the current history variable (see Definition 2.7|p.20). Allowing nondeterministic PDA controllers is not suitable because the runtime environment has to keep track of all configurations that are compatible with the history variable. The set of these configurations should be as small as possible to reduce memory requirements and, ideally, contains only one configuration as guaranteed by determinism.

However, the controller may be allowed to execute different events from a configuration. Firstly, the controller must satisfy controllability: it must accept *any* given uncontrollable event that is nondeterministically chosen and sent by the plant. Secondly, the controller must be least restrictive: it must choose nondeterministically a controllable event only being limited by the other properties to be satisfied. These two steps of sending by plant and controller may not alternate in general: it is possible that the controller continuously receives events from the plant without sending events (thereby just monitoring the plant) and, as the other extreme, it may send events to the plant without receiving any events from the plant (thereby blindly controlling the plant in an *open loop*).

— Property of Controllability

A concrete DPDA controller must accept uncontrollable events that are sent by the plant as explained before. To define *epda-operational-controllable* $C \ P \ \Sigma_{uc}$ (see Definition C.1|p.233), we consider two initial derivations d_1 and d_2 of the controller C and the plant P , respectively, where both derivations execute the

events of a common word w of events in n_1 and n_2 steps, respectively, and where the derivation d_2 of the plant P executes an uncontrollable event u afterwards. Then, the controller must be able to extend the initial derivation d_1 by some derivation d_C also executing the event u .

This definition assumes determinism as a nondeterministic PDA controller could also use an initial derivation for the execution of $w @ [u]$ that is not an extension of d_1 .

—• *Property of Nonblockingness*

The DPDA closed loop CL must satisfy the nonblockingness property to ensure that the goal region can be reached continuously. The goal region is given by the initial derivations of the specification that satisfy the marking condition (i.e., initial derivation that contain a marking configuration, which in turn must contain a marking state). However, as for the DES-based setting, this additional relationship is established by additionally requiring the satisfaction of the specification by the closed loop as stated below.

To define *epda-operational-nonblockingness CL* (see Definition C.2|p.234), we state that each initial derivation d_h of the closed loop CL of length n_h can be extended by a finite-length derivation d_c such that the composition of d_h and d_c satisfies the marking condition of CL .

—• *Property of Satisfaction of the Specification*

The closed loop CL must satisfy the requirements stated by the DPDA specification S . That is, each initial derivation of CL must be equally feasible in the specification S .

To define *epda-operational-specification-satisfaction CL S* (see Definition C.3|p.234), we state that for each initial derivation d_1 of the closed loop CL of length n_1 there is a corresponding initial derivation d_2 of length n_2 of the specification S such that the last configurations of d_1 and d_2 agree on the events executed so far. Also, we require d_2 to satisfy the marking condition if d_1 satisfies the marking condition to ensure that the marking behavior of S is respected by the closed loop CL .

—• *Property of Deadlock Freedom*

The closed loop CL must not deadlock during its execution. A deadlock occurs if the closed loop is unable to perform another step without being in a marking configuration. That is, if no events are sent by the plant and the closed loop has just finished a task, it is allowed to remain in an idle state not performing any steps. It is an important observation that nonblockingness ensures deadlock freedom for DPDA because nonblockingness forces the closed loop to reach a marking configuration from any configuration at hand.

To define *epda-deadlock-freedom CL* (see Definition C.4|p.234), we state that for each initial derivation d of the closed loop CL of length n that does not satisfy the marking condition there is an applicable step from the last configuration of d .

—• *Property of Livelock Freedom*

The closed loop CL must not livelock during its execution. A livelock occurs if the closed loop is unable to execute a further event while it may perform arbitrarily

many internal steps. Since the plant is a DFA, the controller would perform these internal steps autonomously. The consequence of a livelock is that the controller is not able to respond to the plant in the sense of receiving or sending events. Note, a livelock that does not visit a marking configuration infinitely often is excluded for nonblocking DPDA as well.

To define *epda-livelock-freedom* CL (see Definition C.5|p.235), we state that there is no infinite initial derivation d of the closed loop CL for which the events executed so far are not changed after some index N .

→ *Concrete Supervisory Control Problem for DPDA*

We proceed similarly as for DES and collect the properties introduced above into a single definition. This set characterizes all DPDA controllers that would result in closed loops without undesirable behavior.

Definition 3.8: «SCP-Controller-Satisfactory-DPDA»

definition SCP-Controller-Satisfactory-DPDA

$:: (plant-state, event, unused-stack) epda$
 $\Rightarrow (specification-state, event, specification-stack) epda$
 $\Rightarrow event\ set$
 $\Rightarrow (controller-state, event, controller-stack) epda\ set$

where $SCP_{Sat}^{C,DPDA} P S \Sigma_{uc}$

$\equiv \{ C . valid-dpda\ C$
 $\wedge epda-operational-controllable\ \Sigma_{uc}\ P\ C$
 $\wedge epda-operational-nonblockingness\ (F_{DPDA-DFA-Product}\ C\ P)$
 $\wedge epda-operational-specification-satisfaction\ (F_{DPDA-DFA-Product}\ C\ P)\ S$
 $\wedge epda-deadlock-freedom\ (F_{DPDA-DFA-Product}\ C\ P)$
 $\wedge epda-livelock-freedom\ (F_{DPDA-DFA-Product}\ C\ P) \}$

Based on these satisfactory DPDA controllers and the DFA plant P , we now define the closed loops that are induced by them.

Definition 3.9: «SCP-Closed-Loop-Satisfactory-DPDA»

definition SCP-Closed-Loop-Satisfactory-DPDA

$:: (plant-state, event, unused-stack) epda$
 $\Rightarrow (specification-state, event, specification-stack) epda$
 $\Rightarrow event\ set$
 $\Rightarrow ((controller-state, plant-state)\ tuple2, event, controller-stack) epda\ set$

where $SCP_{Sat}^{CL,DPDA} P S \Sigma_{uc}$

$\equiv \{ F_{DPDA-DFA-Product}\ C\ P \mid C . C \in SCP_{Sat}^{C,DPDA} P S \Sigma_{uc} \}$

As for DES, we now determine the least restrictive satisfactory DPDA controllers. These DPDA controllers are satisfactory and result in satisfactory closed loops. However, for least restrictiveness, we cannot proceed as for DES because there is no complete (semi)lattice of DPDA as they do not allow for infinite joins. Instead, we reuse the notion of supremality from the DES-based complete lattice and

state that the closed loop obtained must be supremal w.r.t. (a) the satisfactory DPDA closed loops (that is, there is no better DPDA controller than C) and (b) the satisfactory DES closed loops (that is, there is no better DES controller than $epda\text{-}to\text{-}des\ C$). The second property is required to ensure that the concrete supervisory control problem refines the abstract supervisory control problem as explained later on in section 3.3|p.41

Definition 3.10: «SCP-Controller-Least-Restrictive-DPDA»

definition SCP-Controller-Least-Restrictive-DPDA

$:: (plant\text{-}state, event, unused\text{-}stack)\ epda$
 $\Rightarrow (specification\text{-}state, event, specification\text{-}stack)\ epda$
 $\Rightarrow event\ set$
 $\Rightarrow (controller\text{-}state, event, controller\text{-}stack)\ epda\ set$

where $SCP_{LR}^{C,DPDA} P\ S\ \Sigma_{uc}$

$\equiv \{ C . C \in SCP_{Sat}^{C,DPDA} P\ S\ \Sigma_{uc}$
 $\wedge F_{DPDA\text{-}DFA\text{-}Product} C\ P \in SCP_{Sat}^{CL,DPDA} P\ S\ \Sigma_{uc}$
 $\wedge epda\text{-}to\text{-}des (F_{DPDA\text{-}DFA\text{-}Product} C\ P)$
 $= Sup (epda\text{-}to\text{-}des \setminus (SCP_{Sat}^{CL,DPDA} P\ S\ \Sigma_{uc}))$
 $\wedge epda\text{-}to\text{-}des (F_{DPDA\text{-}DFA\text{-}Product} C\ P)$
 $= Sup (SCP_{Sat}^{CL} (epda\text{-}to\text{-}des\ P) (epda\text{-}to\text{-}des\ S)\ \Sigma_{uc}) \}$

With the definitions introduced, we now state the concrete supervisory control problem for DPDA specifications and DFA plants.

Definition 3.11: «Concrete Supervisory Control Problem for DPDA and DFA»

Given a set of events Σ , which is partitioned into controllable events Σ_c and uncontrollable events Σ_{uc} , a plant P in the form of a DFA over Σ , and a specification S in the form of a DPDA over Σ , *determine* a controller in the form of a DPDA over Σ from $SCP_{LR}^{C,DPDA} P\ S\ \Sigma_{uc}$ or *determine* that this set is empty.

This concrete control problem is solved up to nontermination by our concrete controller synthesis algorithm as presented in chapter 5|p.61. Also, in the next section, we relate the abstract and the concrete supervisory control problems.

→ *Additional Property of Minimality for DPDA*

The concrete control problem from above may have multiple solutions. We propose two notions of minimal DPDA controllers, which can be used to restrict the set of solutions. Firstly, a DPDA controller C is called *language-minimal* if it is the smallest solution w.r.t. the size of the marked and unmarked languages (that is, a DPDA controller C is *language-minimal* if $Inf (epda\text{-}to\text{-}des \setminus (SCP_{LR}^{C,DPDA} P\ S\ \Sigma_{uc}))$ equals $epda\text{-}to\text{-}des\ C$). Secondly, a DPDA controller C is called *size-minimal* if it is the smallest solution w.r.t. the number of states and edges.

We discuss in subsection 8.3.1|p.209 future work regarding these two notions and our concrete algorithm results in language-minimal controllers even though we would prefer size-minimal controllers to reduce the static memory requirements for the runtime environment while producing the identical closed loop.

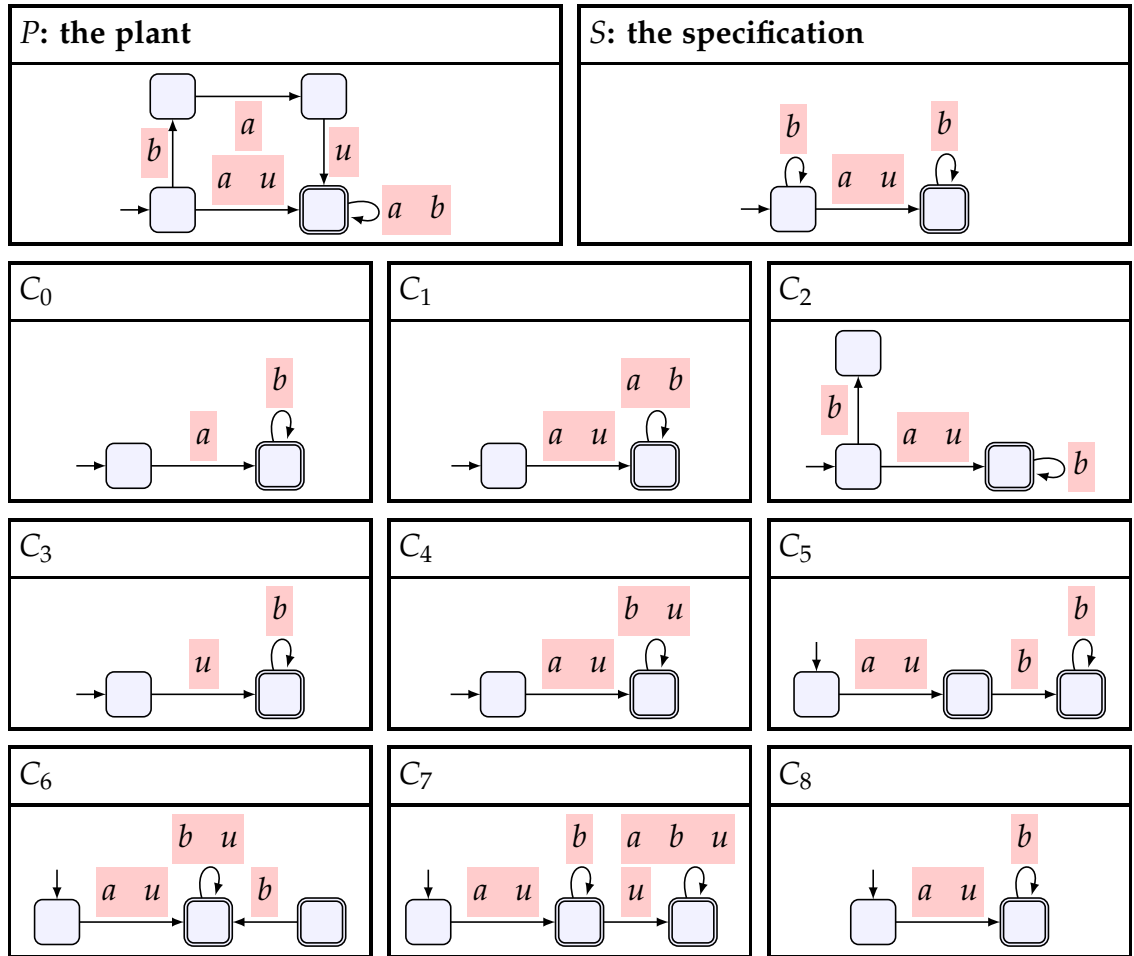
→ *Additional Property of Accessibility for DPDA*

Again, to identify a reasonable subclass of the solutions to the concrete control problem, we propose two notions of accessibility of controllers. Firstly, a controller C is *accessible* (by itself), formally written as *epda-accessible* C , if all of its elements (i.e., states and edges) are used in some initial derivation of it (see Definition C.6|p.235). Secondly, a controller C is *accessible in the closed loop* CL , formally written as *epda-accessible-in-closed-loop* C CL , if all of its elements are used (in the expected modified form due to the synchronous composition with the plant) in some initial derivation of the closed loop (see Definition C.7|p.235).

These two properties, which cannot be defined at the level of DES, are not required for a sound closed loop behavior in the concrete control problem. However, to obtain size-minimal controllers in the future, we may invalidate the property of accessibility in the closed loop on purpose (see Par. *Outputs of the Concrete Controller Synthesis Algorithm*|p.212). Note that both notions of accessibility are unrelated to the two previous notions of *language-minimal* and *size-minimal* controllers.

Example 3.1: «Examples of Controllers with Their Properties»

Events: $\Sigma_c = \{a, b\}$ and $\Sigma_{uc} = \{u\}$



3.3. Correspondence between Abstract and Concrete Supervisory Control Problems

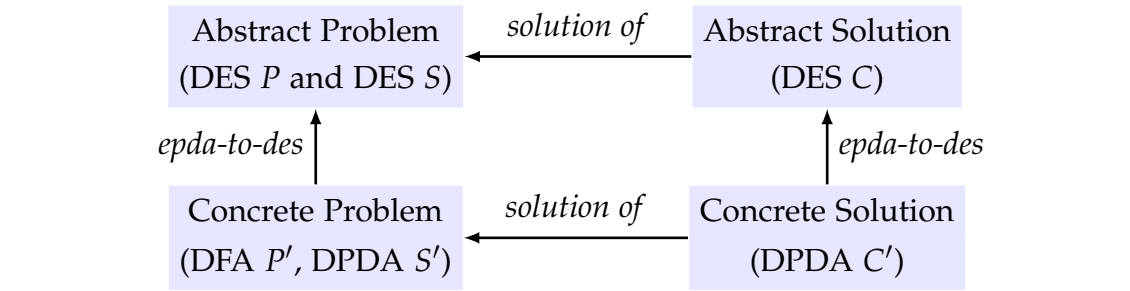
Legend:

Properties of Controller	C ₀	C ₁	C ₂	C ₃	C ₄	C ₅	C ₆	C ₇	C ₈
controllability	✗	✓	✓	✓	✓	✓	✓	✓	✓
specification satisfaction	✓	✗	✓	✓	✓	✓	✓	✓	✓
nonblockingness	✓	✓	✗	✓	✓	✓	✓	✓	✓
satisfactory	✗	✗	✗	✓	✓	✓	✓	✓	✓
+ least restrictive	✗	✗	✗	✗	✓	✓	✓	✓	✓
+ language minimal	✗	✗	✗	✗	✗	✓	✓	✗	✓
+ size minimal	✗	✗	✗	✗	✗	✗	✗	✗	✓
accessible	✓	✓	✓	✓	✓	✓	✗	✓	✓
accessible in the closed loop	✓	✓	✓	✓	✗	✓	✗	✗	✓

3.3. CORRESPONDENCE BETWEEN ABSTRACT AND CONCRETE SUPERVISORY CONTROL PROBLEMS

We compare the abstract supervisory control problem from Definition 3.7|p.35 and the concrete supervisory control problem from Definition 3.11|p.39. Note that we are not aware of previous formal considerations on the relationship between abstract suprema-based and concrete automata-based characterizations.

Figure 3.1: «Overview of Connection between Abstract and Concrete Problem»



→ Existence of Corresponding Instances

Not every instance of the abstract supervisory control problem (given by a DES plant P , a DES specification S , and a set of uncontrollable events Σ_{uc}) corresponds to some instance of the concrete supervisory control problem. This is due to the lack of expressiveness of DFA and DPDA used for plants and specifications. For example, the DES P may have the marked language $\{ a^n b^n \cdot n \in \mathbf{N} \}$ that cannot be realized by a DFA. Subsequently, we assume that P and S are realized by a DFA P' and a DPDA S' , respectively, in the sense of $epda\text{-}to\text{-}des P' = P$ and $epda\text{-}to\text{-}des S' = S$.

→ Existence of Abstract Solution for Abstract Problem

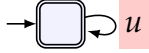

The abstract supervisory control problem has a solution for each of its instances. The DES solution C is formally given by $Sup (SCP_{Sat}^{CL} P S \Sigma_{uc})$ in all cases.

→ *Existence of Concrete Solution for Concrete Problem*

The concrete supervisory control problem does not have a solution for each of its instances: In the following example, we demonstrate that a satisfactory DPDA controller cannot be obtained for each input. Note, the DES solution $Sup (SCP_{Sat}^{CL} P S \Sigma_{uc})$ of the corresponding abstract instance, which is the element bot of the complete lattice of DES equal to $DES \{ \} \{ \}$ in this case, cannot be realized by a DPDA because every DPDA has the empty word in its unmarked language.

Example 3.2: «Unrealizability of the Solution bot of the Abstract Supervisory Control Problem»

Events: $\Sigma_c = \{ \}$ and $\Sigma_{uc} = \{ u \}$

P: the plant	S: the specification	C: the controller
$DES \{ [], [u] \} \{ [], [u] \}$	$DES \{ [] \} \{ \}$	$DES \{ \} \{ \}$
P': the plant	S': the specification	C': the controller
		<i>does not exist</i>

Besides this problem that originates from allowing empty unmarked languages in valid DES as discussed below of Definition 3.5|p.34, we also cannot exclude that further problem instances have no solution because, firstly, our concrete controller synthesis algorithm from chapter 5|p.61 does not terminate in general (see section 5.5|p.87) and, secondly, our second synthesis algorithm from subsection 8.2.2|p.193 has not been formally verified to be sound and terminating. Hence, there may be no DPDA solution to a particular concrete problem instance (that is, $SCP_{LR}^{C,DPDA} P S \Sigma_{uc}$ may be empty) because there may be no suitable DPDA C that realizes $Sup (SCP_{Sat}^{CL} P S \Sigma_{uc})$ and that also satisfies the additional required semantical properties. While the class of deterministic context free languages (DCFL), which is generated by DPDA, is not closed under arbitrary union, the satisfactory DPDA closed loops may be a sufficiently restricted subclass of DPDA to allow for arbitrary unions in general. Of course, the closure under union of satisfactory controllers is not required for the obvious termination of our second algorithm from subsection 8.2.2|p.193.

→ *Correspondence of Solutions for the Abstract and Concrete Instance*

The definitions for the abstract and the concrete setting correspond closely. Firstly, every satisfactory DPDA controller C translates to a satisfactory DES controller $epda\text{-}to\text{-}des C$ in terms of Definition 3.8|p.38 and Definition 3.4|p.34. Secondly, every satisfactory DPDA closed loop CL translates to a satisfactory DES closed loop $epda\text{-}to\text{-}des CL$ in terms of Definition 3.9|p.38 and Definition 3.5|p.34. Lastly, every least restrictive satisfactory DPDA controller C translates to a least restrictive satisfactory DES controller $epda\text{-}to\text{-}des C$ in terms of Definition 3.10|p.39 and Definition 3.6|p.35. For the last part to be satisfied, we have to require the

equality of $epda\text{-}to\text{-}des (F_{DPDA\text{-}DFA\text{-}Product} C P)$ (that is, the DPDA closed loop converted to a DES) and $Sup (SCP_{Sat}^{CL} (epda\text{-}to\text{-}des P) (epda\text{-}to\text{-}des S) \Sigma_{uc})$ (that is, the supremal DES closed loop) in Definition 3.10|p.39 to ensure that the least restrictive DPDA controller must not be more restrictive than any satisfactory DES controller that cannot be realized by a satisfactory DPDA controller. We conclude that the concrete supervisory control problem refines the abstract supervisory control problem as desired and that the reverse direction does not hold due to the increased expressiveness of DES compared to DPDA and DFA (and the minor unrealizability problem related to *bot*).

Chapter 3|p.31

(Abstract and Concrete Supervisory Control Problems)

In this chapter, we introduced the abstract and the concrete supervisory control problem and ensured that they are compatible with each other under the expected limitations. Also, the definition of the concrete supervisory control problem relied on the definition of the abstract supervisory control problem by reusing the notion of least restrictiveness stated on the foundation of DES.

In the next chapter, we determine for the abstract setting of DES an abstract fixed-point controller synthesis algorithm that relates to the abstract supervisory control problem. Afterwards, in chapter 5|p.61, we determine a concrete fixed-point controller synthesis algorithm that relates to the concrete supervisory control problem.

The correspondence between both supervisory control problems is due to the similarities of the involved properties. Also, both controller synthesis algorithms reestablish this correspondence by making use of corresponding building blocks.

4

Abstract Controller Synthesis Algorithm for Discrete Event Systems

We introduce an abstract controller synthesis algorithm, which is applied to a DES plant P , a DES specification S , and a set of uncontrollable events Σ_{uc} .

The least restrictive satisfactory controller (see Definition 3.6|p.35) that is synthesized is equal to a greatest fixed point of an endofunction on DES, which restricts the current controller candidate in each step least restrictively.

The used endofunction is constructed compositionally from multiple endofunctions, which we call fixed-point iterators or abstract building blocks. Each of these abstract building blocks then enforces a single property that is required for satisfactory controllers. This compositional construction of the fixed-point iterator simplifies reasoning at the abstract level of DES. Moreover, the concrete controller synthesis algorithm introduced in chapter 5|p.61 uses concrete building blocks operating on automata to implement the abstract building blocks. The abstract controller synthesis algorithm thereby enables compositional reasoning for the verification of the concrete controller synthesis algorithm and, moreover, the transfer of results from the abstract to the concrete level.

The connection between the least restrictive satisfactory controller and the greatest fixed point of a suitable endofunction operating on models of controller candidates has been introduced in [281] for more restrictive settings with different terminology and without considerations on composability of fixed-point iterators.

The abstract controller synthesis algorithm does not terminate in general if no further restrictions on the inputs P and S are assumed. Moreover, the algorithm is known to be terminating only for plants and specifications that are realizable by DFA as of now. By also relating to our LR(1)-CFG based concrete controller synthesis algorithm from subsection 8.2.2|p.193, we observe how the fixed-point iterator for enforcing controllability should be adapted to result in a terminating fixed-point computation also for specifications realizable by DPDA.

CONTENTS OF THIS CHAPTER

4.1|p.47 *Framework of Abstract Building Blocks for Fixed-point Computation*

We introduce a framework for the computation of greatest fixed points using abstract building blocks (given in the form of fixed-point iterators on DESs). We attach signatures to the fixed-point iterators to allow for results on suitable composition of fixed-point iterators.

4.2|p.50 *Abstract Building Blocks for Enforcing Properties on Discrete Event Systems Least Restrictively*

We introduce abstract building blocks for enforcing least restrictively the properties required for satisfactory controller such as for example the properties of nonblockingness and controllability.

4.3|p.53 *Abstract Controller Synthesis Algorithm for Discrete Event Systems*

To compute the least restrictive satisfactory controllers from Definition 3.6|p.35, we define an abstract controller synthesis algorithm by composing the abstract building blocks from section 4.2|p.50.

4.4|p.56 *On the Termination of the Abstract Controller Synthesis Algorithm*

We discuss classes of inputs for which the abstract controller synthesis algorithm does (or does not) terminate resulting in least restrictive satisfactory controllers.

4.1. FRAMEWORK OF ABSTRACT BUILDING BLOCKS FOR FIXED-POINT COMPUTATION

We introduce a framework of admissible abstract building blocks, which are used to compute the least restrictive satisfactory controllers from Definition 3.6|p.35. We recall the definition of the greatest fixed point of a function F , which operates on the elements of a complete lattice, because both, the least restrictive satisfactory controllers and greatest fixed points, are defined as certain suprema in the complete lattice of DESs. Note, the type variable α that occurs subsequently can be instantiated by Σdes because DESs constitute a complete lattice according to Theorem 2.1|p.17.

Definition 4.1: «gfp»

definition $gfp :: (\alpha :: complete-lattice \Rightarrow \alpha) \Rightarrow \alpha$
where $gfp F \equiv Sup \{ A . A \leq F A \}$

Monotonicity plays an important role for the existence of greatest fixed points according to [338]. This ground laying work is part of the standard Isabelle library and we can straightforwardly obtain the following result stating that the supremum $Sup \{ A . A = F A \}$ is the greatest fixed point of the monotone function F .

Corollary 4.1: «gfp Properties Using Monotonicity»

$mono (F :: \alpha :: complete-lattice \Rightarrow \alpha)$
 $\implies GFP F = Sup \{ A . A = F A \} \wedge GFP F = F (GFP F)$

To obtain an implementable algorithm, we introduce the function *Compute-FP*, which starts with the initial value D and applies the function F , called fixed-point iterator or abstract building block subsequently, until obtaining a fixed point.

Definition 4.2: «Compute-FP»

function (**domintros**) *Compute-FP*
 $:: (\alpha \Rightarrow \alpha) \Rightarrow \alpha \Rightarrow \alpha$
where *Compute-FP* $F D$
 $= (\text{if } D = F D \text{ then } D \text{ else } \text{Compute-FP } F (F D))$

We denote, using standard Isabelle notation, that *Compute-FP* terminates on F and D by *Compute-FP_dom* (F, D) . Note that we believe that it is not possible to provide properties on the fixed-point iterator F that ensure termination and that are satisfied by our fixed-point iterators later on (see Theorem 4.12|p.56).

The sequence of values obtained during the computation of *Compute-FP* $F D$ is guaranteed to be decreasing in each step (that is, $top \geq F top \geq F (F top) \geq \dots$) if D is *top*. To also allow for fixed-point computations that are not starting with *top* and, moreover, to allow for the proper composition of fixed-point iterators later on, we require that fixed-point iterators satisfy the property of *decreasing arguments*, that is, fixed-point iterators must satisfy $F C \leq C$ for every C .

Moreover, to ensure that we obtain a greatest fixed point satisfying additional properties (see the properties of satisfactory DES controllers in Definition 3.4|p.34), we introduce signatures of fixed-point iterators. These signatures consist of three sets describing the possible inputs Q_{inp} , outputs Q_{out} , and fixed points Q_{term} . Obviously, Q_{inp} and Q_{out} are similarly contained in the Hoare calculus [161]. In the following definition, we only require properties such as monotonicity and decreasing arguments to be satisfied based on these three sets.

Definition 4.3: «FP-Iterator»

definition *FP-Iterator*

$$:: (\alpha :: \text{complete-lattice} \Rightarrow \alpha) \Rightarrow \alpha \text{ set} \Rightarrow \alpha \text{ set} \Rightarrow \alpha \text{ set} \Rightarrow \text{bool}$$

where *FP-Iterator* $F \ Q_{inp} \ Q_{term} \ Q_{out}$

$$\begin{aligned} &\equiv \forall X \in Q_{inp}. \forall Y \in Q_{inp}. \\ &\quad F \ X \leq X \\ &\quad \wedge (X \leq Y \longrightarrow F \ X \leq F \ Y) \\ &\quad \wedge F \ X \in Q_{out} \\ &\quad \wedge (X \in Q_{term} \longleftrightarrow F \ X = X) \end{aligned}$$

We are now able to state that *Compute-FP* $F \ D$ computes a certain supremum by computing the corresponding greatest fixed point smaller than the initial DES D .

Theorem 4.1: «Compute-FP Computes Supremum»

$$\begin{aligned} &FP\text{-}Iterator \ F \ UNIV \ Q_{term} \ Q_{out} \\ &\implies \text{Compute-FP_dom} \ (F, D) \\ &\implies \text{Compute-FP} \ F \ D = \text{Sup} \ \{ X . X \in Q_{term} \wedge X \leq D \} \end{aligned}$$

A limitation of this theorem is that it requires the fixed-point iterator F to allow arbitrary inputs ($UNIV$ contains all DESs). This limitation is problematic because the fixed-point iterators introduced later have more restricted input sets. However, as stated in the next theorem, we can relate the fixed-point iterator F to be used for implementation with an equivalent technical fixed-point iterator G . Basically, the set Q_{inp} must include the initial value D , G must be closed on Q_{inp} , and F and G must be equal on Q_{inp} .

Theorem 4.2: «Compute-FP Computes Supremum Using Equivalent Iterator»

$$\begin{aligned} &FP\text{-}Iterator \ G \ UNIV \ Q_{term} \ Q_{out} \\ &\implies D \in Q_{inp} \\ &\implies Q_{out} \subseteq Q_{inp} \\ &\implies (\wedge C. C \in Q_{inp} \implies G \ C = F \ C) \\ &\implies \text{Compute-FP_dom} \ (F, D) \\ &\implies \text{Compute-FP} \ F \ D = \text{Sup} \ \{ X . X \in Q_{term} \wedge X \leq D \} \end{aligned}$$

This theorem is applied later on in section 4.3|p.53 by (a) constructing a fixed-point iterator F with its expected specific input set and (b) constructing a fixed-point iterator G that identifies the maximal element M contained in the input set of F that is smaller than the provided input D and that applies (a simplified alternative of) F on this element M .

We introduce an unconditional and a conditional sequential composition of fixed-point iterators. They allow to define a fixed-point iterator by means of several more basic abstract building blocks, which enforce properties such as for example nonblockingness and controllability on the DES. Firstly, unconditional composition is defined by function composition of the two fixed-point iterators F_1 and F_2 and by constructing the signature of the resulting fixed-point iterator $F_2 \circ F_1$ from the signatures of F_1 and F_2 . The composed fixed-point iterator then only has the fixed points that are fixed points of both fixed-point iterators.

Theorem 4.3: «Unconditional Composition of Fixed-point Iterators»

$$\begin{aligned} & \text{FP-Iterator } F_1 \ Q_{inp} \ Q_{term1} \ Q_{inter1} \\ \implies & \text{FP-Iterator } F_2 \ Q_{inter2} \ Q_{term2} \ Q_{out} \\ \implies & Q_{inter1} \subseteq Q_{inter2} \\ \implies & \text{FP-Iterator } (F_2 \circ F_1) \ Q_{inp} \ (Q_{term1} \cap Q_{term2}) \ Q_{out} \end{aligned}$$

Secondly, we define conditional composition where the second fixed-point iterator F_2 is skipped if the argument is a fixed point of F_1 . Intuitively, such a composition is useful if every fixed point of F_1 is a fixed point of F_2 and the computation of F_2 requires a non-negligible amount of resources.

Definition 4.4: «if-comp»

definition $if\text{-}comp :: (\alpha \Rightarrow \alpha) \Rightarrow (\alpha \Rightarrow \alpha) \Rightarrow (\alpha \Rightarrow \alpha)$
where $if\text{-}comp \ F_2 \ F_1 \equiv \lambda C. \text{ if } F_1 \ C = C \text{ then } C \text{ else } (F_2 \circ F_1) \ C$

The resulting sets of fixed points and outputs are more bulky in the following theorem compared to the case of unconditional composition above. However, in the concrete applications later on, the resulting sets usually collapse to much simpler forms.

Theorem 4.4: «Conditional Composition of Fixed-point Iterators»

$$\begin{aligned} & \text{FP-Iterator } F_1 \ Q_{inp} \ Q_{term1} \ Q_{inter1} \\ \implies & \text{FP-Iterator } F_2 \ Q_{inter2} \ Q_{term2} \ Q_{out} \\ \implies & Q_{inter1} \subseteq Q_{inter2} \\ \implies & Q_{inp} \cap Q_{term1} \subseteq Q_{term2} \\ \implies & \text{FP-Iterator } (if\text{-}comp \ F_2 \ F_1) \ Q_{inp} \\ & \quad \{ C . \text{ if } F_1 \ C = C \text{ then } C \in Q_{inp} \cap Q_{term1} \text{ else } C \in Q_{term1} \cap Q_{term2} \} \\ & \quad ((Q_{inp} \cap Q_{inter1} \cap Q_{term1}) \cup Q_{out}) \end{aligned}$$

We also make use of results on weakening and strengthening (of signatures) of fixed-point iterators but these additional results are omitted here for readability.

For comparison, the supremal controllable sublanguage (of the unmarked language) has been iteratively computed by a monotone function in [281] as well. However, in [281] nonblockingness is not considered, composition of monotone functions is not investigated, and it is based only on unmarked languages rather than on DESs, which consider the marked and unmarked language at once as required for the property of nonblockingness.

4.2. ABSTRACT BUILDING BLOCKS FOR ENFORCING PROPERTIES ON DISCRETE EVENT SYSTEMS LEAST RESTRICTIVELY

Based on the framework of fixed-point iterators introduced before, we present for each property that is required for satisfactory controllers (see Definition 3.4|p.34) one fixed-point iterator that enforces this property least restrictively. In particular, we present fixed-point iterators for restricting an arbitrary DES to a valid DES, for enforcing the satisfaction of the specification, for enforcing nonblockingness, and for enforcing controllability. We then compose these fixed-point iterators in the next section for the computation of least restrictive satisfactory controllers.

4.2.1. Fixed-point Iterator Enforce-Valid-DES

We introduce the fixed-point iterator *Enforce-Valid-DES* for obtaining the supremal *valid* DES that is smaller than an arbitrary (possibly not valid) element of the complete lattice of DESs. We define the two languages of the resulting DES in two steps: firstly, we determine the supremal prefix-closed sublanguage A_{um} of the unmarked language of the input DES, and, secondly, we restrict the marked language of the input DES to the marked words that are contained in A_{um} . This construction ensures that we do not add elements to either of the two sets, which would not be true if we would have maintained the marked language while using its prefix-closure as the unmarked language.

Definition 4.5: «Enforce-Valid-DES»

definition $Enforce-Valid-DES :: \Sigma des \Rightarrow \Sigma des$

where $Enforce-Valid-DES C$

$$\equiv \text{let } A_{um} = \text{Sup } \{ A . A \subseteq L_{um}^{des} C \wedge A = \text{prefix-closure } A \} \\ \text{in } DES A_{um} (A_{um} \cap L_m^{des} C)$$

We now state that *Enforce-Valid-DES* is a fixed-point iterator and produces the supremal satisfactory solution. We make use of the two variables *SAT* and *RES* to represent the satisfactory solutions and computed solution, respectively.

Theorem 4.5: «Enforce-Valid-DES is Sound and Least Restrictive»

$$\begin{aligned} SAT &= \{ A . A \leq C \wedge \text{valid-des } A \} \\ \implies RES &= Enforce-Valid-DES C \\ \implies RES &= \text{Sup } SAT \wedge RES \in SAT \\ &\wedge FP-Iterator Enforce-Valid-DES \text{ top } \{ C . \text{valid-des } C \} \{ C . \text{valid-des } C \} \end{aligned}$$

The fixed-point iterator *Enforce-Valid-DES* is only used later on in the fixed-point iterator *FP-Iterator-Step-Alt* (see Definition 4.12|p.55), which is used for the fixed-point iterator *G* occurring in Theorem 4.2|p.48. It also has, as opposed to the subsequently introduced fixed-point iterators enforcing the satisfaction of specifications, nonblockingness, and controllability, no correspondence in the concrete controller synthesis algorithm in chapter 5|p.61 where each occurring DPDA automatically generates a valid DES.

4.2.2. Fixed-point Iterator Enforce-Specification-Satisfaction

We enforce a specification S on a DES C by constructing their infimum. The usage of the infimum operation, which also implements the synchronous communication between plant and controller, relates to the *synchronous satisfaction check* between the specification S and the closed loop to be obtained.

Definition 4.6: «Enforce-Specification-Satisfaction»

definition *Enforce-Specification-Satisfaction* $:: \Sigma des \Rightarrow \Sigma des \Rightarrow \Sigma des$

where *Enforce-Specification-Satisfaction* $S C \equiv \inf S C$

This construction is sound, least restrictive, and generates valid DESs whenever the inputs are valid DESs as well.

Theorem 4.6: «Enforce-Specification-Satisfaction is Sound and Least Restrictive»

$valid-des S$
 $\Rightarrow valid-des C$
 $\Rightarrow SAT = \{ A . A \leq C \wedge DES-satisfaction S A \wedge valid-des A \}$
 $\Rightarrow RES = Enforce-Specification-Satisfaction S C$
 $\Rightarrow RES = Sup SAT \wedge RES \in SAT$
 $\wedge FP-Iterator Enforce-Specification-Satisfaction$
 $\{ C . valid-des C \}$
 $\{ C . valid-des C \wedge DES-satisfaction S C \}$
 $\{ C . valid-des C \wedge DES-satisfaction S C \}$

4.2.3. Fixed-point Iterator Enforce-Nonblocking-DES

Nonblockingness is enforced by the fixed-point iterator *Enforce-Nonblocking-DES*, which restricts the unmarked language of the input to the prefix closure of the preserved marked language of the input.

Definition 4.7: «Enforce-Nonblocking-DES»

definition *Enforce-Nonblocking-DES* $:: \Sigma des \Rightarrow \Sigma des$

where *Enforce-Nonblocking-DES* $C \equiv DES (prefix-closure (L_m^{des} C)) (L_m^{des} C)$

The following theorem now states that we construct the supremal, valid, and nonblocking DES contained in the valid DES provided.

Theorem 4.7: «Enforce-Nonblocking-DES is Sound and Least Restrictive»

$valid-des C$
 $\Rightarrow SAT = \{ A . A \leq C \wedge valid-des A \wedge DES-nonblocking A \}$
 $\Rightarrow RES = Enforce-Nonblocking-DES C$
 $\Rightarrow RES = Sup SAT \wedge RES \in SAT$
 $\wedge FP-Iterator Enforce-Nonblocking-DES$
 $\{ C . valid-des C \}$
 $\{ C . valid-des C \wedge DES-nonblocking C \}$
 $\{ C . valid-des C \wedge DES-nonblocking C \}$

4.2.4. Fixed-point Iterator Enforce-Marked-Controllable-Subset

We introduced controllability of a DES in Definition 3.1|p.33 to obtain a close correspondence to the standard notion of controllability. However, when also assuming nonblockingness we can equivalently define controllability as follows based on the *marked* language of C .

Definition 4.8: «DES-marked-controllable»

definition $DES\text{-marked-controllable} :: \Sigma \text{ des} \Rightarrow \Sigma \text{ set} \Rightarrow \Sigma \text{ des} \Rightarrow \text{bool}$
where $DES\text{-marked-controllable } C \Sigma_{uc} P$
 $\equiv \forall w' w u.$
 $(w' \in L_m^{des} C \wedge w \sqsubseteq w' \wedge u \in \Sigma_{uc} \wedge w @ [u] \in L_{um}^{des} P)$
 $\longrightarrow w @ [u] \in L_{um}^{des} C$

We now provide the fixed-point iterator *Enforce-Marked-Controllable-Subset* (see section A|p.225 for a remark on coauthorship), which removes from the marked language all words with controllability problems and from the unmarked language all words that have a strict prefix with a controllability problem.

Definition 4.9: «Enforce-Marked-Controllable-Subset»

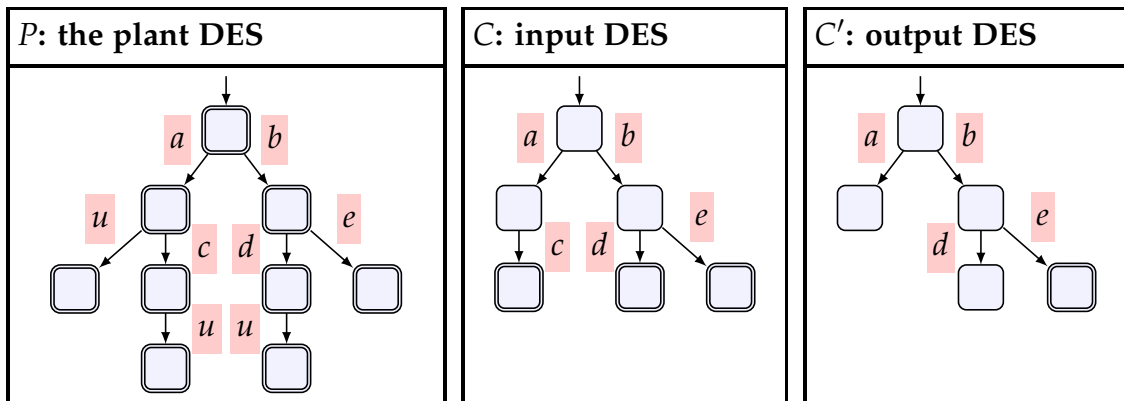
definition $Enforce\text{-Marked-Controllable-Subset}$
 $:: \Sigma \text{ set} \Rightarrow \Sigma \text{ list set} \Rightarrow \Sigma \text{ des} \Rightarrow \Sigma \text{ des}$
where $Enforce\text{-Marked-Controllable-Subset } \Sigma_{uc} P_{um} C$
 $\equiv DES \{ w' \in L_{um}^{des} C . \forall w u. (w \sqsubseteq w' \wedge u \in \Sigma_{uc} \wedge w @ [u] \in P_{um})$
 $\longrightarrow w @ [u] \in L_{um}^{des} C \}$
 $\{ w' \in L_m^{des} C . \forall w u. (w \sqsubseteq w' \wedge u \in \Sigma_{uc} \wedge w @ [u] \in P_{um})$
 $\longrightarrow w @ [u] \in L_{um}^{des} C \}$

We now give an example of this fixed-point iterator.

Example 4.1: «Enforce-Marked-Controllable-Subset»

Events: $\Sigma_c = \{a, b, c, d, e\}$ and $\Sigma_{uc} = \{u\}$

Application: $Enforce\text{-Marked-Controllable-Subset } \Sigma_{uc} (L_{um}^{des} P) C = C'$



Explanation: The marked words $[a, c]$ and $[b, d]$ are removed while the marked word $[b, e]$ without controllability problem is preserved. The unmarked word $[a, c]$ is removed because it has the strict prefix $[a]$ with a controllability problem.

From the example above, we conclude that this fixed-point iterator, when applied to a nonblocking DES, may return a DES that is neither controllable (in the sense of *DES-controllable* from Definition 3.1[p.33]) nor nonblocking. However, the fixed-point iterator is useful because, when applied to a nonblocking DES, (a) it returns the provided DES C when C is controllable and (b) it returns a DES C' that is strictly smaller than C when C is not controllable.

These observations result in the following theorem.

**Corollary 4.2: «Enforce-Marked-Controllable-Subset
is Sound and Least Restrictive»**

$$\begin{aligned}
 & \text{valid-des } C \\
 \implies & \text{valid-des } P \\
 \implies & \text{DES-nonblocking } C \\
 \implies & \text{SAT}_1 = \{ A . A \leq C \wedge \text{valid-des } A \\
 & \quad \wedge \text{DES-controllable } \Sigma_{uc} P A \wedge \text{DES-nonblocking } A \} \\
 \implies & \text{SAT}_2 = \{ A . A \leq C \wedge \text{valid-des } A \} \\
 \implies & \text{RES} = \text{Enforce-Marked-Controllable-Subset } \Sigma_{uc} (L_{um}^{des} P) C \\
 \implies & (\text{RES} = \text{Sup SAT}_1 \vee (C > \text{RES} \wedge \text{RES} > \text{Sup SAT}_1)) \wedge \text{RES} \in \text{SAT}_2 \\
 & \wedge \text{FP-Iterator Enforce-Marked-Controllable-Subset} \\
 & \quad \{ C . \text{valid-des } C \wedge \text{DES-nonblocking } C \} \\
 & \quad \{ C . \text{valid-des } C \wedge \text{DES-nonblocking } C \wedge \text{DES-controllable } \Sigma_{uc} P C \} \\
 & \quad \{ C . \text{valid-des } C \}
 \end{aligned}$$

Since the removal of controllability problems may invalidate the nonblockingness property of the input, the nonblockingness is not a part of the output set of the signature of the fixed-point iterator.

4.3. ABSTRACT CONTROLLER SYNTHESIS ALGORITHM FOR DISCRETE EVENT SYSTEMS

The fixed-point computation of the least-restrictive satisfactory controller (see Definition 3.6[p.35]) is executed by applying *Compute-FP* (see Definition 4.2[p.47]) using an initial value D and a fixed-point iterator F . Firstly, we define D using *FP-Iterator-Init*, which is the (parameterized) initial value to be used. Secondly, we define F by composing the fixed-point iterators introduced in the previous section into the (parameterized) fixed-point iterator *FP-Iterator-Step*.

The initial value D is given by *FP-Iterator-Init* $P S top$ where P is the plant, S is the specification, and top is the largest (valid) DES (see Theorem 2.1[p.17] for *top*). We enforce the satisfaction of the adapted specification *inf* $P S$ (see below) on the (later inserted) input top and then enforce nonblockingness on the result.

Definition 4.10: «FP-Iterator-Init»

definition *FP-Iterator-Init* :: $\Sigma des \Rightarrow \Sigma des \Rightarrow (\Sigma des \Rightarrow \Sigma des)$

where *FP-Iterator-Init* $P S$

$$\equiv \text{Enforce-Nonblocking-DES} \circ (\text{Enforce-Specification-Satisfaction } (\text{inf } P S))$$

Note, by changing the used specification from S into $\inf P S$ we compute the infimal controller that results in the supremal satisfactory closed loop (see Par. *Additional Property of Minimality for DPDA*|p.39). Intuitively, we may change the specification this way because the obtained controller is synchronized with the plant to obtain the closed loop and it is not relevant for obtaining the least restrictive satisfactory controller whether we discard elements forbidden by P already here or as late as in that final synchronization.

Even though *FP-Iterator-Init* is applied only once during the fixed-point computation, we define it to be a fixed-point iterator with a signature by replacing *top* by an arbitrary valid DES. The following theorem (obtained using Theorem 4.6|p.51, Theorem 4.7|p.51, and Theorem 4.3|p.49) then states that *FP-Iterator-Init* immediately (i.e., $Q_{term} = Q_{out}$) enforces least restrictively (least restrictiveness follows trivially also from the first two mentioned theorems) the satisfaction of the properties of nonblockingness and specification satisfaction. However, the initial parameter $D = \text{FP-Iterator-Init } P S \text{ top}$ is not controllable in general.

Theorem 4.8: «Fixed-point Iterator FP-Iterator-Init»

$$\begin{aligned}
 & \text{valid-des } P \\
 \implies & \text{valid-des } S \\
 \implies & \text{FP-Iterator } (\text{FP-Iterator-Init } P S) \\
 & \{ C . \text{valid-des } C \} \\
 & \{ C . \text{valid-des } C \wedge \text{DES-satisfaction } (\inf P S) C \wedge \text{DES-nonblocking } C \} \\
 & \{ C . \text{valid-des } C \wedge \text{DES-satisfaction } (\inf P S) C \wedge \text{DES-nonblocking } C \}
 \end{aligned}$$

Subsequently, we introduce the fixed-point iterator *FP-Iterator-Step*, which executes two steps. Firstly, using *Enforce-Marked-Controllable-Subset* certain uncontrollable words are removed from the DESs' languages. Secondly, if and only if uncontrollable words have been removed, nonblockingness is reinstated by applying the fixed-point iterator *Enforce-Nonblocking-DES*. Note, in the corresponding step of the concrete controller synthesis algorithm we must be able to decide whether uncontrollable words have been removed in the first step to allow for the termination of the controller synthesis algorithm (see section 5.4|p.85).

Definition 4.11: «FP-Iterator-Step»

$$\begin{aligned}
 & \text{definition } \text{FP-Iterator-Step} \\
 & :: \Sigma \text{ set} \Rightarrow \Sigma \text{ des} \Rightarrow (\Sigma \text{ des} \Rightarrow \Sigma \text{ des}) \\
 & \text{where } \text{FP-Iterator-Step } \Sigma_{uc} P \\
 & \quad \equiv \text{if-comp} \\
 & \quad \quad \text{Enforce-Nonblocking-DES} \\
 & \quad \quad (\text{Enforce-Marked-Controllable-Subset } \Sigma_{uc} (L_{um}^{des} P))
 \end{aligned}$$

Again, using the results on composability and strengthening of fixed-point iterators we derive the desired signature of *FP-Iterator-Step*. This signature shows that the fixed points of *FP-Iterator-Step* are satisfactory controllers.

Theorem 4.9: «Fixed-point Iterator FP-Iterator-Step»

$$\begin{aligned}
 & \text{valid-des } P \\
 \implies & \text{valid-des } S \\
 \implies & \text{FP-Iterator } (\text{FP-Iterator-Step } \Sigma_{uc} P) \\
 & \{ C . \text{valid-des } C \wedge \text{DES-satisfaction } (\inf P S) C \wedge \text{DES-nonblocking } C \} \\
 & \{ C . \text{valid-des } C \wedge \text{DES-satisfaction } (\inf P S) C \wedge \text{DES-nonblocking } C \\
 & \quad \wedge \text{DES-controllable } \Sigma_{uc} P C \} \\
 & \{ C . \text{valid-des } C \wedge \text{DES-satisfaction } (\inf P S) C \wedge \text{DES-nonblocking } C \}
 \end{aligned}$$

To obtain least restrictiveness of *FP-Iterator-Step*, we cannot apply Theorem 4.1|p.48 because the input set of *FP-Iterator-Step* is not *UNIV*. We solve this problem by using Theorem 4.2|p.48 instead as described before. Firstly, we define the technical fixed-point iterator *FP-Iterator-Step-Alt* with the input set *UNIV*. Secondly, we show that *FP-Iterator-Step-Alt* and *FP-Iterator-Step* are equivalent.

As the first step, we define the fixed-point iterator *FP-Iterator-Step-Alt*, which least restrictively enforces the input set of *FP-Iterator-Step* before executing a simplified unconditional form of it.

Definition 4.12: «FP-Iterator-Step-Alt»

$$\begin{aligned}
 & \text{definition } \text{FP-Iterator-Step-Alt} :: \Sigma \text{ set} \Rightarrow \Sigma \text{ des} \Rightarrow (\Sigma \text{ des} \Rightarrow \Sigma \text{ des}) \\
 & \text{where } \text{FP-Iterator-Step-Alt } \Sigma_{uc} P \\
 & \quad \equiv \text{Enforce-Nonblocking-DES} \circ \\
 & \quad \quad (\text{Enforce-Marked-Controllable-Subset } \Sigma_{uc} (L_{um}^{des} P)) \circ \\
 & \quad \quad \text{Enforce-Nonblocking-DES} \circ \\
 & \quad \quad \text{Enforce-Valid-DES}
 \end{aligned}$$

The signature of *FP-Iterator-Step-Alt* with *UNIV* as input set is given as follows. Note, *FP-Iterator-Step-Alt* does not need to enforce the satisfaction of the specification according to Theorem 4.2|p.48; rather it only has to be equivalent to *FP-Iterator-Step* on the input set of *FP-Iterator-Step*.

Theorem 4.10: «Fixed-point Iterator FP-Iterator-Step-Alt»

$$\begin{aligned}
 & \text{valid-des } P \\
 \implies & \text{valid-des } S \\
 \implies & \text{FP-Iterator } (\text{FP-Iterator-Step-Alt } \Sigma_{uc} P) \\
 & \text{top} \\
 & \{ C . \text{valid-des } C \wedge \text{DES-nonblocking } C \wedge \text{DES-controllable } \Sigma_{uc} P C \} \\
 & \{ C . \text{valid-des } C \wedge \text{DES-nonblocking } C \}
 \end{aligned}$$

As the second step, we state in the following theorem that the two fixed-point iterators *FP-Iterator-Step-Alt* and *FP-Iterator-Step* are equivalent.

Theorem 4.11: «Equivalence of FP-Iterator-Step-Alt and FP-Iterator-Step»

$$\begin{aligned}
 & \text{valid-des } P \\
 \implies & \text{DES-nonblocking } C \\
 \implies & \text{FP-Iterator-Step-Alt } \Sigma_{uc} P C = \text{FP-Iterator-Step } \Sigma_{uc} P C
 \end{aligned}$$

Lastly, we conclude using Theorem 4.2|p.48 that, provided the termination of *Compute-FP*, we obtain the infimal, least restrictive satisfactory controller when executing *Compute-FP* with the fixed-point iterator *FP-Iterator-Step* $\Sigma_{uc} P$ and the initial value *FP-Iterator-Init* $P S top$.

Theorem 4.12: «Computation of the Least Restrictive Satisfactory Controller»

$$\begin{aligned}
 & valid\text{-}des P \\
 \implies & valid\text{-}des S \\
 \implies & C = \text{Compute-FP} (FP\text{-}Iterator\text{-}Step \Sigma_{uc} P) (FP\text{-}Iterator\text{-}Init P S top) \\
 \implies & \text{Compute-FP_dom} (FP\text{-}Iterator\text{-}Step \Sigma_{uc} P, FP\text{-}Iterator\text{-}Init P S top) \\
 \implies & C \in SCP_{LR}^C P S \Sigma_{uc} \\
 \wedge & C = \text{Inf} (SCP_{LR}^C P S \Sigma_{uc})
 \end{aligned}$$

4.4. ON THE TERMINATION OF THE ABSTRACT CONTROLLER SYNTHESIS ALGORITHM

The abstract controller synthesis algorithm as given in Theorem 4.12|p.56 by *Compute-FP* (*FP-Iterator-Step* $\Sigma_{uc} P$) (*FP-Iterator-Init* $P S top$) terminates for some but not for all two valid DES P (the plant) and S (the specification). That is, the assumption *Compute-FP_dom* (*FP-Iterator-Step* $\Sigma_{uc} P$, *FP-Iterator-Init* $P S top$) stating the termination of the computation is necessary for Theorem 4.12|p.56.

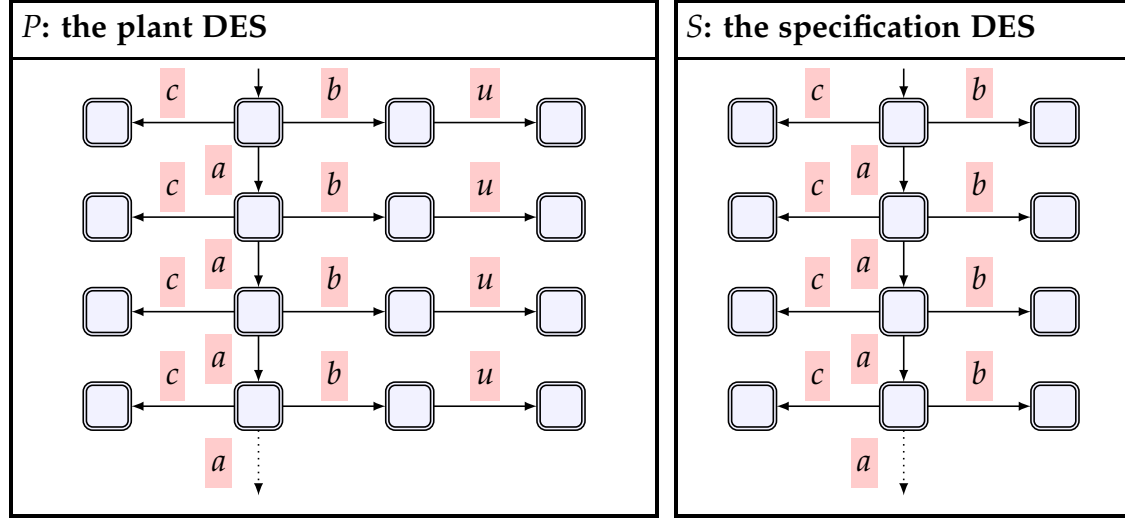
Determining further classes of languages for which the abstract synthesis algorithm terminates and the definition of corresponding controller synthesis algorithms is a challenge for the future. We consider only two relevant classes of problem instances in the following two subsections. These classes correspond to the two settings where the specification is realizable by a DFA and a DPDA, respectively, while the plant is realizable as a DFA in each case.

4.4.1. Plants Realizable by DFA and Specifications Realizable by DFA

In this subsection we assume that the plant P and the specification S are both realizable by DFA. For this problem instance, it is well known that the abstract controller synthesis algorithm always terminates [281, Theorem 3.1, p. 1074]. The argument for termination is based on the Myhill-Nerode-Equivalence: for a nonblocking DES D , we define this equivalence on $L_{um}^{des} D$, which is denoted \sim subsequently, as follows: $w_1 \sim w_2$ iff $\{ w . w_1 @ w \in L_m^{des} D \} = \{ w . w_2 @ w \in L_m^{des} D \}$. In [281, Theorem 3.1, p. 1074] it is stated that termination then follows from the fact that the number of Myhill-Nerode-Equivalence classes of the intermediate controller candidates decrease suitably fast during the fixed-point computation with the obvious lower bound 0. Also, a terminating concrete synthesis algorithm, which operates on these DFA realizations, has been presented already in [281]. Consider the following example where $S \leq P$ holds and where the first controller candidate C_0 is therefore equal to S .

Example 4.2: «Termination for a Plant Realizable by a DFA and a Specification Realizable by a DFA»

Events: $\Sigma_c = \{a, b, c\}$ and $\Sigma_{uc} = \{u\}$



Myhill-Nerode-Equivalence classes:

DES	Myhill-Nerode-Equivalence classes		
P	a^*	a^*b	$a^*c + a^*bu$
$S = C_0$	a^*	$a^*b + a^*c$	
C_1	a^*	a^*c	

Discussion: In the first step of the fixed-point computation, only the words a^*b would be removed from the marked language of the first controller candidate $C_0 = S$ to obtain the controller candidate C_1 . Note that the number of Myhill-Nerode-Equivalence classes of two is not strictly decreased in this step.

However, from inspecting the concrete controller synthesis algorithm from chapter 5 [p.61 or [281] for this setting, we observe that the controller candidate C_0 is realized by a DFA where the states are not in one-to-one correspondence with its Myhill-Nerode-Equivalence classes due to the product construction that implements the *inf* operation on two DFA, which results in a *disambiguation* as follows. The states of the product DFA correspond then to all binary intersections of one Myhill-Nerode-Equivalence class of P and one Myhill-Nerode-Equivalence class of S . That is, besides some empty intersections, we would obtain the sets a^* , a^*b , and a^*c corresponding to the three states of the realization of C_0 . Hence, the removal of a^*b reduces the number of states of C_0 .

We conjecture that, if P and S have n_P and n_S Myhill-Nerode-Equivalence classes, respectively, then the fixed-point computation terminates after at most $n_P \times n_S$ steps because the words from at least one of the remaining nonempty binary intersections are removed in each step.

Determining a suitable *disambiguation* (here using the product of the realizations of P and S) is a key step for controller synthesis throughout this thesis.

4.4.2. Plants Realizable by DFA and Specifications Realizable by DPDA

The abstract controller synthesis algorithm terminates on many problem instances when the plant is realizable by a DFA and the specification is realizable by a DPDA: see [280, Example 2, page 644] and the examples in chapter 7|p.141.

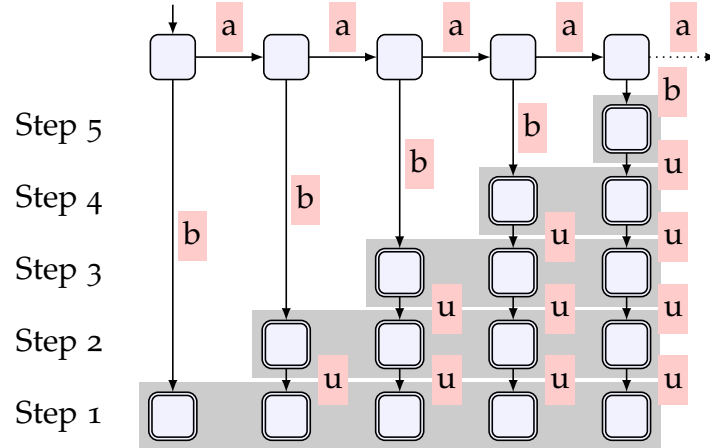
However, as demonstrated by the following example, which is similar to [280, Example 1, page 643], there are problem instances where the abstract controller synthesis algorithm does not terminate. For these problem instances, the fixed-point computation generates an infinite sequence of controller candidates that are all not controllable.

Example 4.3: «Nontermination for a Plant Realizable by a DFA and a Specification Realizable by a DPDA»

Inputs: $\Sigma_{uc} = \{u\}$, nonblocking plant P with $L_m^{des} P = \{a^i b u^j \mid i, j \in \mathbf{N}\}$, and nonblocking specification S with $L_m^{des} S = \{a^i b u^i \mid i \in \mathbf{N}\}$.

Explanation: Firstly, the initial controller candidate C_0 is equal to S because $S \leq P$ and S is nonblocking. Secondly, in step $k \geq 1$, to obtain the controller candidate C_k from the controller candidate C_{k-1} , we remove the infinite set of words $\{a^{k+i} b u^i \mid i \in \mathbf{N}\}$ from the marked and unmarked language of C_{k-1} because each of them prevents the event u that is possibly sent by the plant. This removal of words is also visualized in the picture below where only a small part of the infinitely many words are included. Note, *bot* is the least restrictive satisfactory controller, which is never obtained.

Application: Compute-FP (FP-Iterator-Step $\Sigma_{uc} P$) (FP-Iterator-Init $P S$ top)



In each step, the (infinite set of) words given by one row in the picture is removed.

We believe that the incompleteness of the abstract synthesis algorithm is not relevant in the context of controller synthesis because all problem instances where the abstract synthesis algorithm exhibits nontermination commonly require that the execution of a certain number of uncontrollable events is specified in S . Such specifications are not reasonable in the first place because the controller cannot enforce the execution of (a certain number of) uncontrollable events. Consider the

example above where the specification requires that, once the word $a^n b$ has been executed, it must be possible to subsequently execute up to n times the event u but not $n + 1$ times.

Similarly to the example of the previous subsection, we can observe that one Myhill-Nerode-Equivalence class is removed from the marked set of controller candidates in each step: in step i precisely those words are removed that can be extended with the words $\{ u^k . k \leq i \}$. However, the number of Myhill-Nerode-Equivalence classes is not bounded in this DPDA-oriented setting in contrast to the DFA-oriented setting from the previous subsection.

The fixed-point iterator *Enforce-Nonblocking-DES* for enforcing nonblockingness does not play a relevant role in the example above and, hence, we believe that it is not the source of the nontermination. We rather conjecture that the second fixed-point iterator *Enforce-Marked-Controllable-Subset* is the source of nontermination because it is not sufficiently strong in the sense that it does not detect large enough patterns of words to be removed in each step of the fixed-point computation.

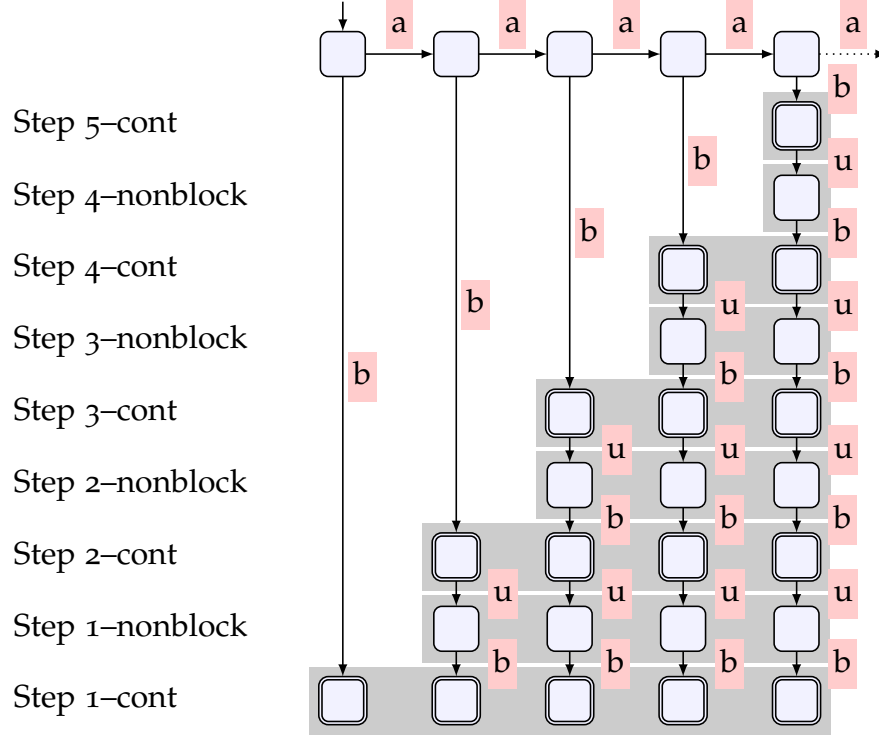
To obtain a terminating fixed-point computation, we considered a stronger fixed-point iterator for enforcing controllability, which does not only check for a single missing uncontrollable event but for an arbitrary long nonempty sequence of uncontrollable events. The underlying condition of controllability has been introduced in [201, Theorem 3.2, page 6] to allow for a faster execution of the synthesis algorithm [201, Algorithm 3.10, page 8]: basically, more uncontrollable words are removed in each step, which usually results in fewer computation steps to be executed. This stronger fixed-point iterator is helpful because it detects for every $a^n b$ that the sequence u^{n+1} is missing for the example above. This results in the removal of all words $a^n b$ in the first step of the fixed-point algorithm. However, in the following more elaborate example, the property of nonblockingness is involved more prominently also resulting in a nonterminating computation when using the stronger fixed-point iterator.

Example 4.4: «Nontermination for a Plant Realizable by a DFA and a Specification Realizable by a DPDA with Nonblockingness»

Inputs: $\Sigma_{uc} = \{ u \}$, nonblocking plant P with $L_m^{des} P = \{ a^i b (ub)^j . i, j \in \mathbf{N} \}$, and nonblocking specification S with $L_m^{des} S = \{ a^i b (ub)^i . i \in \mathbf{N} \}$.

Explanation: Firstly, the initial controller candidate C_0 is equal to S because $S \leq P$ and S is nonblocking. Secondly, in step $k \geq 1$, to obtain the controller candidate C_k from the controller candidate C_{k-1} , we remove in (Step k -cont) the infinite set of words $\{ a^{k+i} b (ub)^i . i \in \mathbf{N} \}$ from the marked and unmarked language of C_{k-1} because each of them prevents the event u that is possibly sent by the plant and in (Step k -nonblock) the infinite set of words $\{ a^{k+i+1} b (ub)^i u . i \geq 0 \}$ from the unmarked language of C_{k-1} to reestablish nonblockingness. This removal of words is also visualized in the picture below where only a small part of the infinitely many words are included. Note, *bot* is the least restrictive satisfactory controller, which is never obtained.

Application: Compute-FP (FP-Iterator-Step $\Sigma_{uc} P$) (FP-Iterator-Init $P S top$)



In each step, the (infinite set of) words given by one row in the picture is removed.

We conclude that the previously mentioned stronger fixed-point iterator for enforcing controllability is also not sufficient to ensure termination.

When considering the similarities between the removed words in the example above, we conjecture that a sufficient fixed-point iterator for enforcing controllability must incorporate the inherent bracketing/stack-behavior of DPDA to prune enough words at once. We present initial ideas on this in subsection 8.2.2|p.193 in our second concrete controller synthesis algorithm.

Chapter 4|p.45

(Abstract Controller Synthesis Algorithm for Discrete Event Systems)

We introduced a framework of fixed-point iterators. This framework enables a compositional approach for providing fixed-point iterators for enforcing the desired properties of satisfactory controllers. We determined for these fixed-point iterators suitable signatures and properties and constructed a fixed-point algorithm from these fixed-point iterators for solving the abstract supervisory control problem.

The presented approach lays the foundation for the automata-based construction of the concrete controller synthesis algorithm introduced in the next chapter. In this concrete controller synthesis algorithm, we implement the abstract building blocks (given by the abstract fixed-point iterators) by concrete building blocks using automata-based operations. These concrete building blocks then satisfy concretizations of the signatures of the fixed-point iterators introduced in this chapter.

5

Concrete Controller Synthesis Algorithm for Deterministic Pushdown Automata

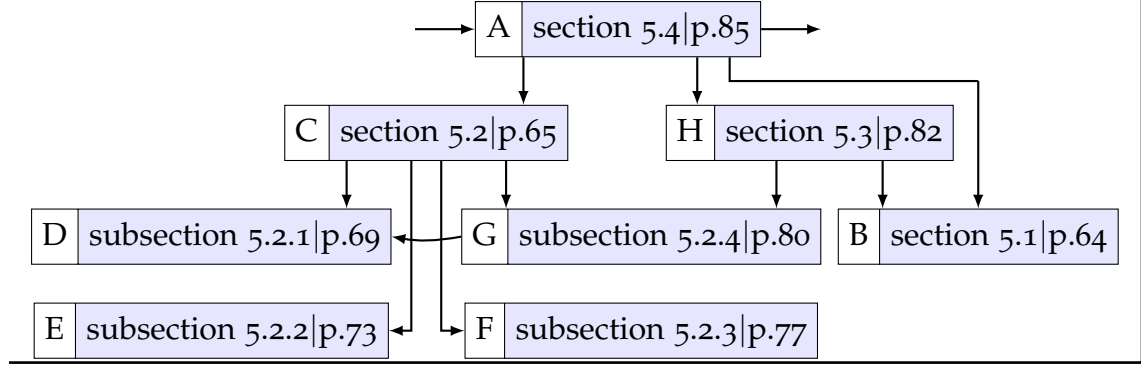
We introduce our automata-based concrete controller synthesis algorithm as an instantiation of the DES-based abstract controller synthesis algorithm verified in Theorem 4.12|p.56. This instantiation is given in two steps. Firstly, we implement the DES-based fixed-point iterators that are employed in the abstract controller synthesis algorithm by concrete building blocks operating on automata, CFGs, and Parsers. Secondly, we define the concrete controller synthesis algorithm by resembling the combination of the fixed-point iterators in the abstract controller synthesis algorithm by combining the concrete building blocks analogously. We include three major building blocks in our concrete controller synthesis algorithm: we construct the initial DPDA controller candidate satisfying the specification, we enforce nonblockingness on a DPDA controller candidate, and we reduce controllability of DPDA to the nonblockingness of DPDA.

The algorithm is sound in the sense that every DPDA controller obtained upon a terminating fixed-point computation is a solution to the concrete supervisory control problem given in Definition 3.11|p.39. Moreover, if the algorithm terminates and returns no controller candidate, it is guaranteed that there is no DPDA controller that is able to enforce the specification on the plant satisfactorily.

The algorithm does not terminate in general as explained in section 4.4|p.56 because some specifications may result in an infinite sequence of controller candidates. Two different approaches to this problem are discussed as future and ongoing work in subsection 8.3.1|p.209 and subsection 8.2.2|p.193, respectively. Moreover, in chapter 7|p.141, we provide various examples where the algorithm terminates to support our claim that the specifications leading to nonterminating computations are always unreasonable specifications from a control theoretic perspective.

Consider the overview of the building blocks of the concrete controller synthesis algorithm in the following figure. The top-level building block A has input and output edges attached on the left and right side, respectively. The edges between the building blocks declare the usage of the target building block by the source building block.

Figure 5.1: «Structure of the Concrete Controller Synthesis Algorithm»



CONTENTS OF THIS CHAPTER

5.1|p.64 Concrete Building Block for the Synchronous Composition of Deterministic Pushdown Automata with Deterministic Finite Automata

The introduced building block B represents the well-known automata-based operation $F_{DPDA-DFA-Product}$, which implements the DES-based operation *inf* used for different purposes in the abstract controller synthesis algorithm. The construction implements the well-known synchronous-product automaton of a given DPDA and a given DFA.

5.2|p.65 Concrete Building Block for Enforcing Nonblockingness for Deterministic Pushdown Automata

The second introduced building block C represents the DPDA-based operation $F_{DPDA-Enforce-Nonblocking,Opt}$, which implements the DES-based operation *Enforce-Nonblocking-DES* from Definition 4.7|p.51 for enforcing the nonblockingness property on a controller candidate. The operation $F_{DPDA-Enforce-Nonblocking,Opt}$ also enforces the properties of accessibility, livelock freedom, and deadlock freedom for DPDA and is based on the building blocks D, E, F, and G. These building blocks are used to translate the given input DPDA into an LR(1)-CFG, which satisfies the nonblockingness property, to translate this LR(1)-CFG back into a DPDA, and to enforce the accessibility property on the resulting DPDA.

5.3|p.82 *Concrete Building Block for Reducing Controllability to Nonblockingness for Deterministic Pushdown Automata*

The third introduced building block H represents the DPDA-based operation $F_{DPDA-Reduce-Controllable}$, which implements the DES-based operation *Enforce-Marked-Controllable-Subset* from Definition 4.9|p.52 for reducing the controllability property to the nonblockingness property for a given DPDA controller candidate. The operation also relies on the building block G for enforcing the satisfaction of the accessibility property from section 5.2|p.65.

5.4|p.85 *Concrete Synthesis Algorithm as an Instantiation of the Abstract Synthesis Algorithm*

Finally, the introduced building block A represents the DPDA-based concrete controller synthesis algorithm, which is given by the operation $F_{DPDA-DFA-Construct-Controller}$. The operation $F_{DPDA-DFA-Construct-Controller}$ implements the DES-based controller synthesis algorithm as given in Theorem 4.12|p.56. Both algorithms are fixed-point algorithms that are started on a carefully obtained initial value. As explained, the fixed-point iterators employed in the abstract controller synthesis algorithm correspond directly to the concrete building blocks presented in this chapter. The central operations are $F_{DPDA-DFA-Product}$ for constructing the adapted specification and for explicitly reintroducing the state-correspondence with the plant as explained in section 5.1|p.64, $F_{DPDA-Enforce-Nonblocking, Opt}$ for enforcing (amongst others) the satisfaction of the nonblockingness property, and the operation $F_{DPDA-Reduce-Controllable}$ for reducing the satisfaction of the controllability problem to the satisfaction of the nonblockingness property.

5.5|p.87 *On the Termination of the Concrete Controller Synthesis Algorithm*

As for the abstract controller synthesis algorithm, we discuss the viable nontermination of the concrete controller synthesis algorithm. For this cause, we repeat an example for nontermination discussed in subsection 4.4.2|p.58 by discussing the automata realizations of the involved DPDA and DFA. This example demonstrates how the formalism of DPDA can be misused to define specifications resulting in nonterminating fixed-point computations. Potential resolutions of such nonterminating computations resulting from these artificial specifications are discussed as ongoing and future work in subsection 8.2.2|p.193 and subsection 8.3.1|p.209, respectively.

5.1. CONCRETE BUILDING BLOCK FOR THE SYNCHRONOUS COMPOSITION OF DETERMINISTIC PUSHDOWN AUTOMATA WITH DETERMINISTIC FINITE AUTOMATA

Before discussing the implementation of the DES-based operation inf (see Theorem 2.1|p.17), we recall its three central applications in the two previous chapters. Firstly, given a DES controller C' and a DES plant P' , we constructed their closed loop $\text{inf } C' P'$ (see Par. *Construction of the Closed Loop*|p.33). Secondly, given a DES controller candidate C' and a DES specification S' , we enforced the satisfaction of S' by constructing the next controller candidate $\text{inf } S' C'$, which is entirely contained in S' (see Definition 4.6|p.51). Thirdly, given a DES plant P' and a DES specification S' , we constructed an adapted specification $\text{inf } S' P'$, which is entirely contained in P' while imposing the same requirements on the behavior of the closed loop (see Definition 4.10|p.53).

We introduce with building block B the usual automata-based product construction $F_{\text{DPDA-DFA-Product}}$ (see for example [163, Theorem 7.27, p.286]). It constructs for a given DPDA and a given DFA their synchronous execution as a DPDA where both automata synchronize on their events and where the DPDA performs its internal steps autonomously. $F_{\text{DPDA-DFA-Product}}$ implements inf in the sense of $\text{inf}(\text{epda-to-des } C)(\text{epda-to-des } P) = \text{epda-to-des}(F_{\text{DPDA-DFA-Product}} C P)$.

For the first case from above: we have already mentioned (see Par. *Construction of the Closed Loop*|p.36) that we construct the closed loop $F_{\text{DPDA-DFA-Product}} C P$ for a given DPDA controller C and a given DFA plant P . For the second case from above: we enforce the satisfaction of the adapted DES specification $\text{inf } P S$ in Theorem 4.12|p.56 only on the DES *top* and we do not need to perform this step in our concrete controller synthesis algorithm since $\text{inf } S P \leq S$ holds from the beginning. For the third case from above: we adapt the DPDA specification S using the DFA plant P into $F_{\text{DPDA-DFA-Product}} S P$ (see section 5.4|p.85).

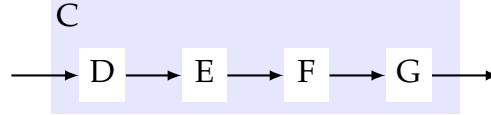
For a fourth case, which does not occur in the abstract controller synthesis algorithm: we construct for a DPDA controller candidate C and the DFA plant P the DPDA controller candidate $F_{\text{DPDA-DFA-Product}} C P$ (see section 5.3|p.82). We know that C and $F_{\text{DPDA-DFA-Product}} C P$ are equal w.r.t. their corresponding DES counterparts because (a) the DPDA controller candidates are monotonically decreasing w.r.t. their corresponding DES counterparts and (b) the initial controller candidate $\text{inf } S P$ is smaller than P . However, we call this step a *labelling step* because (a) $F_{\text{DPDA-DFA-Product}} C P$ has states of the form (q_1, q_2) where q_1 is a state of C and q_2 is a state of P and (b) every initial derivation d of $F_{\text{DPDA-DFA-Product}} C P$ can be projected to an initial derivation d' of P (possibly of shorter length because internal steps of C , which are not synchronously performed with P , are removed during the projection) executing the same events. Moreover, we call this step a *disambiguation step* because $F_{\text{DPDA-DFA-Product}} C P$ has at least as many accessible states as C ; intuitively, the synchronous composition results in the splitting of a

state q_1 contained in C into multiple states (q_1, q_2) for each state q_2 of plant P . In section 5.3|p.82, we rely on the fact that the application of $F_{DPDA-DFA-Product}$ preserves operational nonblockingness, livelock freedom, deadlock freedom, and operational controllability. For example, $F_{DPDA-DFA-Product} C P$ is known to be operational nonblocking in section 5.3|p.82 because C is known to be operational nonblocking and $epda\text{-}to\text{-}des C \leq epda\text{-}to\text{-}des P$.

5.2. CONCRETE BUILDING BLOCK FOR ENFORCING NONBLOCKINGNESS FOR DETERMINISTIC PUSHDOWN AUTOMATA

The building block C represents the operation $F_{DPDA-Enforce-Nonblocking, Opt}$, which implements *Enforce-Nonblocking-DES* from subsection 4.2.3|p.51. It transforms a DPDA controller C into another DPDA controller C' , which satisfies the properties of nonblockingness, livelock freedom, deadlock freedom, accessibility, and which generates the same marked language. The inner structure of C with its four steps is given in the following figure and explained in the subsequent subsections.

Figure 5.2: «Overview of Enforcing Nonblockingness on DPDA»



We discuss how the nonblockingness and accessibility properties can be enforced for DFA, CFG, regular grammars (a strict subset of CFG, which are equivalent to DFA), and LR(1)-CFG before considering the case of DPDA.

→ Enforcing the Nonblockingness and Accessibility Properties for DFA

The properties of nonblockingness and accessibility can be enforced for DFA using the following static fixed-point backward and forward analysis algorithms.

Definition 5.1: «Enforce the Nonblockingness Property for DFA»

1. Define Q to be the set of all marking states of the input DFA.
2. Until a fixed-point is reached: if $\langle q_1, a, s_1, s_2, q_2 \rangle$ is an edge of the input DFA and $q_2 \in Q$, then add q_1 to Q .
3. Restrict the input DFA to the states contained in Q
(side case: Q may not contain the initial state of the input DFA).

Definition 5.2: «Enforce the Accessibility Property for DFA»

1. Define Q to be the singleton set of the initial state of the input DFA.
2. Until a fixed-point is reached: if $\langle q_1, a, s_1, s_2, q_2 \rangle$ is an edge of the input DFA and $q_1 \in Q$, then add q_2 to Q .
3. Restrict the input DFA to the states contained in Q .

→ *Enforcing the Nonblockingness and Accessibility Properties for CFG*

Enforcing the nonblockingness and accessibility properties for CFG can be done analogously to the case for DFA using the following static fixed-point backward and forward analysis algorithms.

Definition 5.3: «Enforce the Nonblockingness Property for CFG»

1. Define Q to be the empty set of nonterminals of the input CFG.
2. Until a fixed-point is reached: if $\langle prod-lhs = A, prod-rhs = w \rangle$ is a production of the input CFG and every nonterminal contained in w is in Q , then add A to Q .
3. Restrict the input CFG to the nonterminals contained in Q (side case: Q may not contain the initial nonterminal of the input CFG).

Definition 5.4: «Enforce the Accessibility Property for CFG»

1. Define Q to be the singleton set of the initial nonterminal of the input CFG.
2. Until a fixed-point is reached: if $\langle prod-lhs = A, prod-rhs = w \rangle$ is a production of the input CFG and $A \in Q$, then add every nonterminal contained in w to Q .
3. Restrict the input CFG to the nonterminals contained in Q .

Moreover, we can enforce the nonblockingness property for regular grammars and LR(1)-CFG by using the algorithm for arbitrary CFG because these subclasses are closed under the removal of nonterminals and productions.

→ *Enforcing the Nonblockingness and Accessibility Properties for DPDA*

Enforcing the nonblockingness and accessibility properties on DPDA cannot proceed analogously as for DFA or CFG because of the additional stack variable contained in the configurations of a DPDA. This stack variable is a *contextual information* on the state that has to be considered during analysis. For DFA and CFG there is no such context on which the step relation additionally depends.

To correctly identify and to remove violations of nonblockingness from a DPDA, we translate the DPDA controller candidate at hand into a CFG for which we can identify and remove these elements in the form of certain CFG productions obtaining an LR(1)-CFG without blocking problems. Adapting results from parsing theory and by defining and employing additional operations, we translate the obtained LR(1)-CFG back into a DPDA controller candidate while maintaining the satisfaction of nonblockingness and the other mentioned properties. Note, the translation of the DPDA into an LR(1)-CFG disambiguates the model to ensure that *precisely* the required productions are removed/retained and subsequent constructions preserve this disambiguation (e.g., the LR(1)-Parser covers the possible future steps using the stack).

The notions of LR(1)-CFG and LR(1)-Parsers originate from [189] and many subsequent developments have been later collected in [325] in consistent notation. For the building blocks D and E, we adopted and formalized the constructions

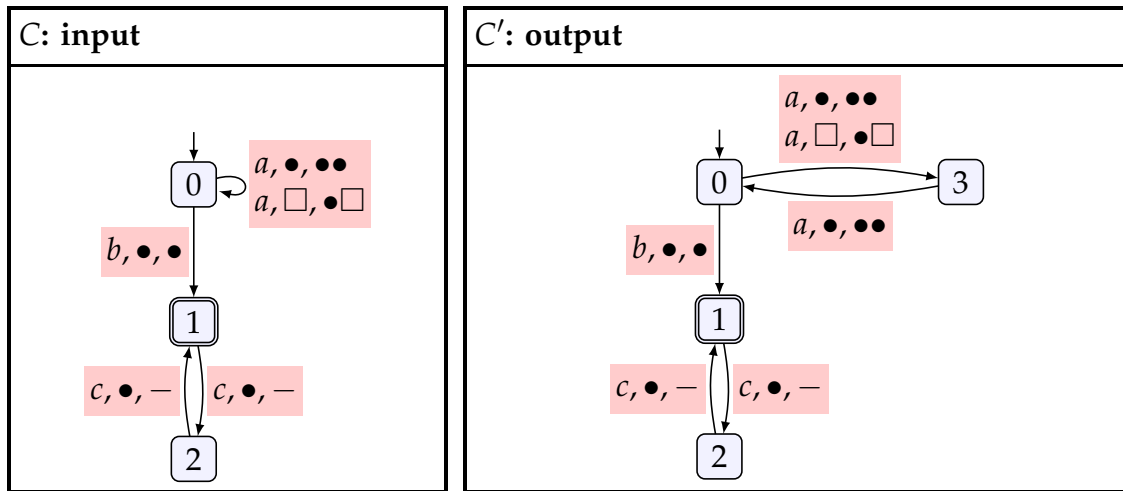
(which have been partially given as pseudo-code) from [189] and [325], respectively. Besides fixing a minor error in the constructions from [189] (not leading to a valid DPDA), we simplified some definitions (for example, the rules of an LR(k)-Parser from [325, Volume II, Section 6.3, p. 31]).

The concrete controller synthesis algorithm returns a controller in the form of a DPDA but not in the form of an LR(1)-CFG or an LR(1)-Parser because DPDA seem to be more intuitive (in particular w.r.t. determinism) and more easily implementable. We have to perform the translation between DPDA and LR(1)-CFG in every iteration of the fixed-point computation because the building block C (presented here) for enforcing the nonblockingness property operates (at its core) on LR(1)-CFG and the building block H (see section 5.3|p.82) for reducing the controllability property to the nonblockingness property operates on DPDA. See subsection 8.2.2|p.193 for our second concrete controller synthesis algorithm for enforcing controllability on LR(1)-CFG directly.

It is not possible to enforce the nonblockingness property on a DPDA by simply removing certain states and edges from it. Multiple productions of the resulting CFG may be constructed for one edge of the DPDA and a nonempty strict subset of these productions may be contained in the resulting LR(1)-CFG. A production that is not in the resulting LR(1)-CFG would then imply that the edge should be removed to enforce the nonblockingness property and a production that is in the resulting LR(1)-CFG would imply that the edge should not be removed to preserve the marked language. This possible ambiguity is resolved by the construction of the LR(1)-Parser in our algorithm.

Finally, consider the DPDA in the following two examples, which demonstrate that the task of enforcing the nonblockingness property is nontrivial for DPDA because localizing violations of the nonblockingness property and the removal of such violations are two nontrivial tasks.

Example 5.1: «Enforcing the Nonblockingness Property for DPDA»



Discussion: Every initial derivation of C that reaches state 2 with no further • symbols on the stack violates the nonblockingness property. In the possible

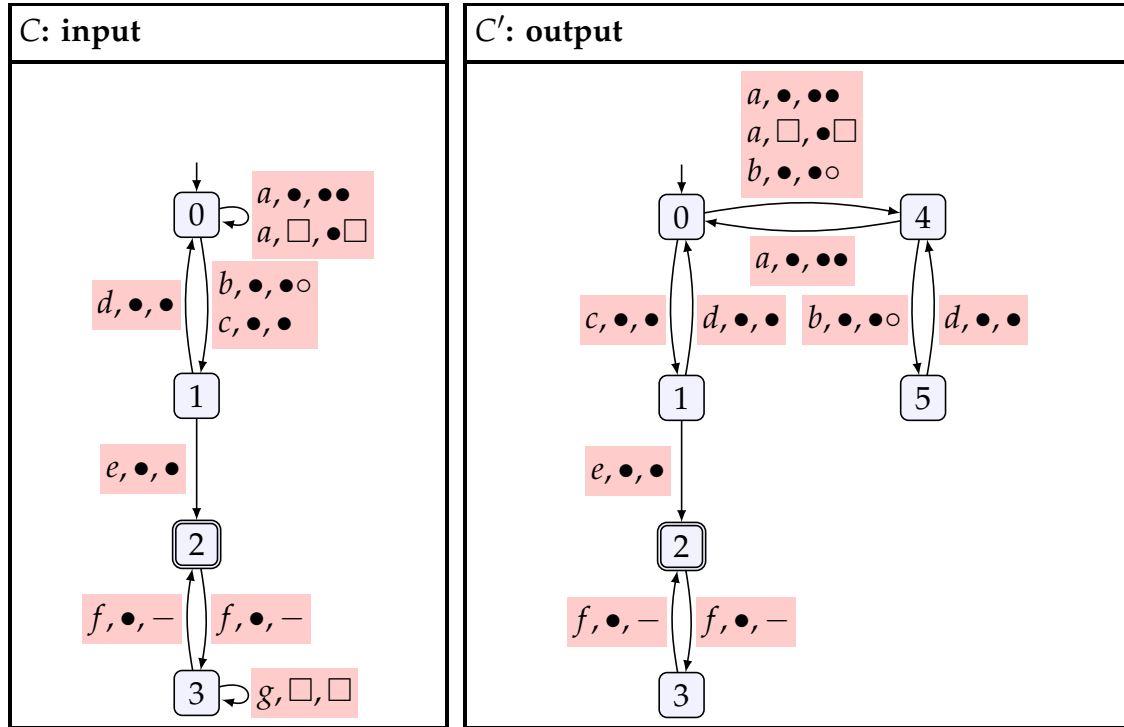
solution C' , we ensure that this does not happen by enforcing that an even number of events a has been executed once the event b is executed. In C' the number of \bullet symbols on the stack is always odd for the states 2 and 3 and even for the other states.

Firstly, detecting violations of the nonblockingness property requires the representation of the (possibly infinite) set of all reachable configurations (with their unbounded stack variables) from which no marking configuration is reachable. It is obviously not sufficient to detect deadlocks and we are not aware of a simple solution to the localization problem.

Secondly, removing violations is complex because the removal may depend on the possibly many and arbitrarily long paths of edges in the DPDA from the cause (execution of an odd number of events a in the example) and the earliest point where the nonblockingness property is violated (when reaching state 2 with no further \bullet symbols on the stack in the example). We are also not aware of a simple solution to this non-local removal problem.

The following example demonstrates that the localization and the removal of violations of nonblockingness can be complex and that deadlock detection is insufficient for localization.

Example 5.2: «Enforcing the Nonblockingness Property for DPDA»

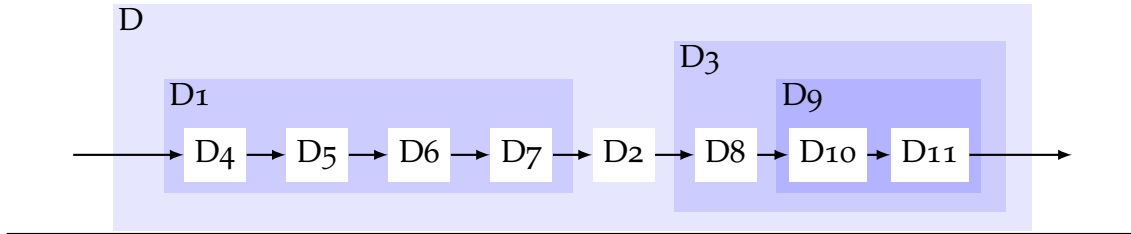


Discussion: Every initial derivation of C that reaches state 3 with a top stack of \circ or \square violates the nonblockingness property. In C' the length of the maximal sequence of \bullet symbols on top of the stack is always odd for the states 3, 4, and 5 and even for the other states.

5.2.1. Convert DPDA into Equivalent LR(1)-CFG

In building block D, we convert a given DPDA into an LR(1)-CFG that satisfies the properties of nonblockingness, accessibility, deadlock freedom, and livelock freedom, and which generates the same marked language. The inner structure of this building block D is given in the following figure and explained subsequently.

Figure 5.3: «Convert DPDA Into Equivalent LR(1)-CFG»



Building block D consists of three steps.

1. In building block D₁, we apply $F_{DPDA \rightarrow SDPDA}$ to convert the given DPDA into an equivalent SDPDA, which is a DPDA with only three kinds of edges.
2. In building block D₂, we apply $F_{SDPDA \rightarrow \text{Enforce-Unique-Marking-Early}}$ to convert the obtained SDPDA into an equivalent SDPDA satisfying a certain nonambiguity property.
3. In building block D₃, we apply $F_{SDPDA \rightarrow LR(1), Opt}$ to convert the obtained SDPDA into the desired LR(1)-CFG.

For the constructions used in these three steps, we relied on [189]. However, for the first building block D₁, we had to fix a minor mistake. Moreover, for the last building block D₃, we introduce an optimized version $F_{SDPDA \rightarrow LR(1), Opt}$ of the standard operation $F_{SDPDA \rightarrow LR(1), Std}$ to increase the performance of the entire concrete controller synthesis algorithm.

For the required proofs, we note that [189] contains no reasoning of correctness for D₁ and D₂ and only a proof idea for the preservation of the marked language for the operation $F_{SDPDA \rightarrow LR(1), Std}$. The fact that the constructed CFG satisfies the LR(1)-property (confer [325, Volume II, Proposition 6.41, p. 53]) was the most challenging single property that we verified. We sketch our proof of this property in section 6.3|p.116 because we are not aware of a previous proof.

Convert DPDA into Equivalent SDPDA

In building block D₁, we apply the operation $F_{DPDA \rightarrow SDPDA}$ to the given DPDA and obtain an SDPDA. The successive execution of the four steps of D₁ ensures that there are three remaining kinds of edges: executing edges, popping edges, and pushing edges. Most importantly, the operation $F_{DPDA \rightarrow SDPDA}$ preserves the marked language between the input DPDA and the output SDPDA. The obtained

SDPDA allows for simpler definitions and proofs of subsequently applied operations. In particular, the operations $F_{SDPDA \rightarrow LR(1), Std}$ and $F_{SDPDA \rightarrow LR(1), Opt}$ are defined over the three kinds of edges occurring in SDPDA.

→ *Building Block D4: $F_{DPDA-Seperate-Executing-Edges}$*

We separate the execution of events from the modifications of the stack. This results in executing edges $\langle q_s, Some\ r, s, s, q_t \rangle$ that do not modify the stack and non-executing edges $\langle q_s, None, po, pu, q_t \rangle$ that may modify the stack.

→ *Building Block D5: $F_{DPDA-Remove-Neutral-Edges}$*

We remove neutral edges of the form $\langle q_s, None, s, s, q_t \rangle$. These edges are replaced by two edges, which, when executed in succession, add an element to the stack and remove the element afterwards again. Hence, after this step, only edges remain that are either executing an event or that modify the stack.

Note that we had to fix a minor error contained in [189] where it is not ensured that the *epda-eos* symbol always remains on the stack.

→ *Building Block D6: $F_{DPDA-Seperate-Push-Pop-Edges}$*

We ensure that the stack is modified in only two different kinds of ways. The resulting automaton is then either popping a single element from the stack by edges of the form $\langle q_s, None, po, [], q_t \rangle$ or it is pushing elements onto the stack by edges of the form $\langle q_s, None, po, pu @ po, q_t \rangle$.

→ *Building Block D7: $F_{DPDA-Remove-Mass-Pushing-Edges}$*

We ensure that at most one element is pushed onto the stack in each step. For this purpose, we split the edges that push more than one element onto the stack into a sequence of such admissible edges.

Enforce Unique Marking Early for SDPDA

We apply the operation $F_{SDPDA-Enforce-Unique-Marking-Early}$, which is implemented by building block D2, to remove a certain ambiguity from the SDPDA prior to the subsequent translation to CFG. In particular, we ensure that the resulting SDPDA has a *unique* initial marking derivation ending in a marking state for every marked word. That is, once a marking state has been accessed in an initial derivation, further steps cannot enter marking states without prior execution of an additional event. Thereby, the automaton is no longer able to mark a word twice in a single initial derivation.

Also note that this operation ensures that livelocks (see Par. *Property of Livelock Freedom* |p.37), which could potentially exist in the input SDPDA, are no longer accessing marking states infinitely often. Livelocks are given by infinite initial derivations d that are not executing events after some index N . The output SDPDA then guarantees that d enters at most one marking state at index $n > N$. The removal of livelocks is then trivial after the translation to CFG where the nonterminals corresponding to the configurations beyond index n are removed as they violate the nonblockingness property.

Convert SDPDA into Equivalent LR(1)-CFG

In the building block D3, we convert the SDPDA obtained from before into an LR(1)-CFG in two steps. In building block D8, we use either the operation $F_{SDPDA \rightarrow CFG, Std}$ from [189] or its optimized version $F_{SDPDA \rightarrow CFG, Opt}$ to convert the SDPDA obtained from before into a CFG. Then, in building block D9, we enforce the satisfaction of the properties of nonblockingness and accessibility for this CFG by applying $F_{CFG-Enforce-Nonblocking}$ and $F_{CFG-Enforce-Accessible}^{cfgSTD}$ (see Definition 5.3|p.66 and Definition 5.4|p.66) to obtain the resulting LR(1)-CFG in building blocks D10 and D11, respectively. The operation used in D3 is called $F_{SDPDA \rightarrow LR(1), Std}$ or $F_{SDPDA \rightarrow LR(1), Opt}$ depending on whether $F_{SDPDA \rightarrow CFG, Std}$ or $F_{SDPDA \rightarrow CFG, Opt}$ is used.

→ Conversion Procedure

We construct the CFG such that its initial derivations resemble the initial derivations of the input SDPDA as follows (see Example 5.3|p.72 for an example of the discussed conversion procedure). Recall that the input SDPDA has a unique initial marking derivation d for every marking word w due to D2. Moreover, for simplicity, assume that all elements that are pushed onto the stack are removed later on in d . Hence, the marking configuration of the derivation d looks like $\langle q, w, [\square] \rangle$. Let the term $\beta_{i,j}$ then denote the state of the configuration in which the j th stack element contained in the stack of configuration i has been removed later on in d . Given some configuration $c_i = \langle q, w, [v_1, v_2, \dots, v_{k-1}, v_k] \rangle$ contained in d at index i , we define the corresponding configuration of the resulting CFG as follows: $\langle w @ [L_{q,v_1,\beta_{i,1}}, L_{\beta_{i,1},v_2,\beta_{i,2}}, \dots, L_{\beta_{i,k-2},v_{k-1},\beta_{i,k-1}}, L_{\beta_{i,k-1},\square}] \rangle$. Hence, the configurations of the CFG that belong to the initial and marking configurations c_0 and c_n are $\langle [L_{q_0,\square}] \rangle$ and $\langle w @ [L_{q,\square}] \rangle$ for some q , respectively. Note, the CFG performs one further step from the configuration $\langle w @ [L_{q,\square}] \rangle$ to the configuration $\langle w \rangle$ to actually insert w into its marked language. Moreover, if the simplifying assumption from above stating that the stack is $[\square]$ in the marking configuration c_n is not satisfied, that is, if the stack is of the form $[v_1] @ s$, then $\beta_{i,j}$ is not only undefined for the last stack element \square . In such a situation, the list of nonterminals ends with the last $L_{\beta_{i,j-1},v_j,\beta_{i,j}}$ where $\beta_{i,j}$ is defined. From these explanations, it is straightforward to determine the productions of the resulting CFG to allow for this configuration-based correspondence.

→ Efficiency of the Conversion Procedure

The construction of the CFG has a cubic complexity because we generate a production for the CFG from any edge of the SDPDA that is pushing a stack element onto the stack and any two states. The size of the resulting CFG is problematic for two reasons: the space and time required for its construction. However, the operation $F_{SDPDA \rightarrow CFG, Opt}$ from [189] produces many nonterminals and productions that are removed in the second step (that is, nonterminals and productions never occurring in any initial marking derivation of the resulting

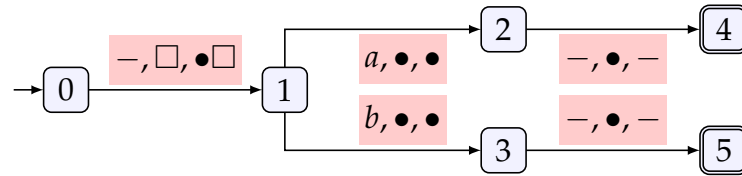
CFG). To reduce the resources required for the translation, we introduced an optimized version $F_{SDPDA \rightarrow CFG, Opt}$. This optimization relies on an over-approximation of the required sets of nonterminals and productions, which relies on an over-approximation of the inter-configuration accessibility for the input SDPDA. Then, only the nonterminals and productions that could not be excluded by this analysis are generated for the resulting CFG. For the over-approximation of the inter-configuration accessibility, we compute an over-approximation of the reachability graph of the SDPDA. The nodes of this reachability graph are essentially given by a state and a stack of bounded length k ; hence, two actual configurations of the SDPDA are represented by the same node in the reachability graph if they coincide in their state and the first k elements of their stack-variable. Typically, even using $k = 0$ improves the performance significantly because the input SDPDA is seldom a strongly connected graph (there is no directed path of edges between any pair of vertices). The two alternative operations are equivalent in the sense that, after the application of the second step, the resulting LR(1)-CFG coincide. In subsection 8.2.1|p.188, we also present an even more efficient recursive conversion procedure that lacks a formal verification as of the writing of this thesis.

→ *Nondeterminism of the Resulting LR(1)-CFG*

In addition to the preservation of the marked language from the input SDPDA, we also preserve the determinism of the SDPDA: the resulting CFG satisfies the LR(1)-property. As stated before, this property has not been verified before (neither in [189] nor in [325]). We sketch our proof of this property in section 6.3|p.116. However, from our explanations above, it is obvious that the resulting CFG does not have a right-unique step-relation as demonstrated in the following example where the SDPDA only marks words when having an empty stack.

Example 5.3: «Expected Nondeterminism of LR(1)-CFG»

Input SDPDA:



Productions of output LR(1)-CFG with initial nonterminal $L_{0,\square}$:

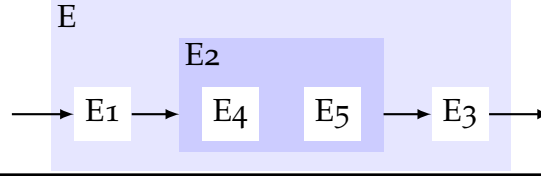
$L_{0,\square} \longrightarrow L_{1,\bullet,4} L_{4,\square}$	$L_{0,\square} \longrightarrow L_{1,\bullet,5} L_{5,\square}$
$L_{1,\bullet,4} \longrightarrow a L_{2,\bullet,4}$	$L_{1,\bullet,5} \longrightarrow b L_{3,\bullet,5}$
$L_{2,\bullet,4} \longrightarrow$	$L_{3,\bullet,5} \longrightarrow$
$L_{4,\square} \longrightarrow$	$L_{5,\square} \longrightarrow$

Discussion: The deterministic step from state 1 is translated into a nondeterministic step in $L_{0,\square}$ in the resulting LR(1)-CFG. However, given a first event to be executed there is a unique step of the LR(1)-CFG to be executed from $L_{0,\square}$; the required static analysis of the LR(1)-CFG to make this deterministic decision is explained subsequently when we construct the LR(1)-Parser for the LR(1)-CFG.

5.2.2. Convert LR(1)-CFG into Equivalent LR(1)-Parser

In building block E, we translate the obtained LR(1)-CFG into an LR(1)-Parser.

Figure 5.4: «Convert LR(1)-CFG Into Equivalent LR(1)-Parser»



The constructions are based on the initial work in [189] where the more general conversion of LR(k)-CFGs into LR(k)-Parsers was introduced. However, we opted to rely on the presentation of these constructions in [325], which in turn is based on [5]. While [325] provides a more thorough presentation including various results, the algorithms are given in pseudo-code. Thus, one of our contributions is also the formal definition of the algorithm employed. In fact, as the proofs are mostly unaffected in terms of proof-complexity by the generalization to the LR(k) setting, we have also carried out the relevant proofs for arbitrary look-ahead lengths k instead of requiring $k = 1$.

The building block E consists of the following steps according to the figure given above.

→ *Building Block E1: $F_{CFG-Augment}$*

Using a fresh nonterminal S and a fresh event $\$,$ we add the production $\langle S, \$A\$ \rangle$ to the LR(1)-CFG and change the initial nonterminal from A to S . Thereby, we wrap the executed words into a leading and trailing event $\$.$ This step simplifies the definition of the subsequently applied operations because we have a unique construction procedure throughout the application of $F_{LR(k)-Machine}$ below instead of having a distinguished construction procedure for the initial production of the LR(1)-CFG. Firstly, the leading $\$$ event is removed during the construction of the LR(1)-Parser in the building block E3 below. Secondly, the trailing $\$$ event ends up to be the processing terminator of the LR(1)-Parser. Hence, the marked and the unmarked languages are not permanently altered by adding the leading and trailing $\$$ event.

→ *Building Blocks E2 and E3: $F_{LR(k)-Machine}$ and $F_{LR(k)-Parser}$*

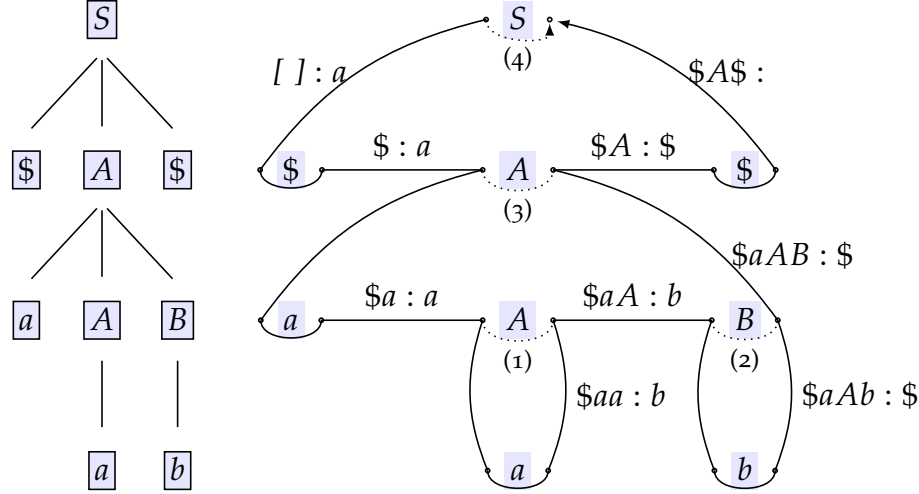
In Example 5.4 p.74, we discuss the construction procedure for the LR(1)-Machine and for the LR(1)-Parser by an example. Firstly, the LR(1)-Machine is constructed in the form of a DFA from the input LR(1)-CFG using two further operations: (a) for a given configuration of the LR(1)-CFG, we compute the first events of all accessible, nonterminal-free configurations in building block E4 (using $F_{CFG-First}$) and (b) we compute successor states and successor state sequences based on events and sequences of events in the LR(1)-Machine in building block E5. Secondly, we obtain the LR(1)-Parser by constructing the rules based on the previously constructed LR(1)-Machine.

Example 5.4: « $F_{LR(k)\text{-Machine}}$ and $F_{LR(k)\text{-Parser}}$ for Nonregular $LR(1)\text{-CFG}$ »

Input: Consider the following $LR(1)\text{-CFG}$ with initial nonterminal S , which is already the result of an application of the operation $F_{CFG\text{-Augment}}$.

$$\begin{aligned}\rho_1 : A &\longrightarrow aAB \\ \rho_2 : A &\longrightarrow a \\ \rho_3 : B &\longrightarrow b \\ \rho_4 : B &\longrightarrow \\ \rho_5 : S &\longrightarrow \$A\$ \end{aligned}$$

Derivation Tree (Left) and Parsing Derivation (Right):



The derivation tree on the left side represents an initial marking derivation of the input $LR(1)\text{-CFG}$ (in any of the three semantics). The corresponding initial marking derivation of the $LR(1)\text{-Parser}$ is depicted in a custom notation on the right side. Intuitively, we apply a post-order traversal to the derivation tree from the left side to obtain a traversal of the parsing derivation on the right side. The obtained traversal starts in the small dot left to the nonterminal S and ends in the small dot right to the nonterminal S . A node is called *traversed* if the small dot on its right side has been reached/passed. Also, each edge in the parsing derivation is labelled by (a) the list of node values visited on the shortest way to it making use of the additional dotted connections below of nonterminals, and (b) the next event that is reached subsequently. For example, the label $\$A : \$$ is obtained by using the dotted edge below the nonterminal A essentially skipping the derivation tree below of it and by using $\$$ because it is the next event to be reached.

When only following the non-dotted lines in the traversal, we obtain the sequence of stacks of the $LR(1)\text{-Parser}$ in the described initial marking derivation (because the $\$$ events are omitted in the Parser, the first and last step will be removed) and, alternatively, we observe the generated events in the order of a left-most initial marking derivation.

Using the dotted lines, we obtain incomplete derivation trees in the right-most semantics. That is, we may fold subtrees one after another by using the dotted lines in the order (1), (2), (3), and (4). Each such folding step corresponds to what the LR(1)-Parser is doing to its stack when reaching the small dot on the right side of a nonterminal: it folds the subtree it just finished by using the dotted line changing the stack using a so called *reduce* rule. The other kind of rules are *shift* rules, which are used to execute events. Both kinds of rules may depend on the current look-ahead symbol next to the separator : in the parsing tree above.

Note, the rule applicability depends on a finite suffix of the current stack, and, hence, two reachable stack-values may be equivalent to the rule applicability. Consequently, it is important that, whenever two rules are applicable, no non-determinism arises. This may happen for CFG not satisfying the LR(1)-property in the form of *shift-reduce* and *reduce-reduce* conflicts.

The stack-values used above are called *viable-prefixes* and they are given for each CFG by a prefix-closed regular language, that is, they can be represented by the unmarked language of a DFA, which is the LR(1)-Machine. Hence, the LR(1)-Parser operates on finite, unbounded initial derivations of the LR(1)-Machine and extends such a current derivation by one further step in shift-rules and exchanges a finite suffix of such a current derivation in reduce-rules.

Intuitively, the LR(1)-Machine unfolds leading nonterminals by applying their productions. This allows for the *immediate descent* using shift-rules (for example, the two *a* events are executed in two successive steps and the production $\langle A, a \rangle$ is not unfolded by the LR(1)-Parser in between) and, moreover, the LR(1)-Machine keeps track of the unfolding steps using different states to allow for the *step-wise ascent* using reduce rules (for example, after execution of the second event *a*). Because the ascent is step-wise, it is important (for determinism) that the next event to be executed is known already in the form of the look-ahead before finishing the ascent.

The LR(1)-Parser with initial stack symbol \$ then uses the following rules in the order given to parse the word *aab*\$. Note, we omit the other rules of the LR(1)-Parser here.

$$\begin{array}{ll}
 \$ \mid a \longrightarrow & \$, \$a \mid \\
 \$a \mid a \longrightarrow & \$a, \$aa \mid \\
 \$a, \$aa \mid b \longrightarrow & \$a, \$aA \mid b \\
 \$a, \$aA \mid b \longrightarrow & \$a, \$aA, \$aAb \mid \\
 \$aA, \$aAb \mid \$ \longrightarrow & \$aA, \$aAB \mid \$ \\
 \$, \$a, \$aA, \$aAB \mid \$ \longrightarrow & \$, \$A \mid \$
 \end{array}$$

The construction of the LR(1)-Parser preserves the (un)marked languages as well as the satisfaction of the properties of determinism (stated by the LR(1)-property for the input LR(1)-CFG) and nonblockingness. We believe that the LR(1)-Parser satisfies the property of livelock freedom, but we do not verify its satisfaction before building block F where we derive its satisfaction by also using nonblockingness to avoid further preservation proofs along the way.

For the preservation of the marked language by the construction procedure, we followed the informal verification given in [325] in which the theorems, lemmas, and proofs turned out to be sufficiently sound and detailed. Also, the construction process is introduced in [325] in a decomposed way, which eased the verification. On the abstract level, viable-prefixes of the input CFG are introduced and used to reason about the parsing steps of the LR(1)-Parser to be constructed. On the concrete level, this connection is implemented by the LR(1)-Machine (based on viable prefixes) and the corresponding LR(1)-Parser with the necessary rules. However, we believe that our explanations contained in the example above are an important additional contribution for supporting the intuition of the workflow of the LR(1)-Parser construction.

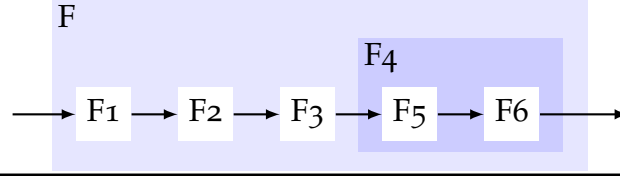
For the preservation of determinism from the LR(1)-CFG to the LR(1)-Parser, we followed the steps described in [325, Volume II, Theorem 6.39, p. 52]. This theorem states that the obtained LR(1)-Parser is deterministic iff the LR(1)-Parser has no reduce-reduce or shift-reduce rule-conflicts iff the LR(1)-Machine has no item-conflicts between items leading to rules of the LR(1)-Parser iff the CFG is an LR(1)-CFG. It is to be noted that the LR(1)-property is given differently in the various sources and our work connects the characterization of the LR(1)-property from [325] with the one in [189] by verifying that the construction from [189] actually enforces the characterization of the LR(1)-property from [325]. This fact has been stated without proof as [325, Volume II, Proposition 6.41, p. 53]. We present our original proof in section 6.3|p.116.

The construction of the LR(1)-Parser preserves the satisfaction of the non-blockingness property, which has been established for the input LR(1)-CFG by applying the algorithm from Definition 5.3|p.66. The notion of nonblockingness is also of special interest in the field of parsing theory where errors contained in the input should be detected as early as possible [325, Volume II, Section 9, p. 289] to be able to provide useful error messages. As a first attempt to handle nonblockingness, we considered the notion of *correct-prefix-parsers* [325, Volume II, Theorem 9.1, p. 291] and [218], which states that whenever a Parser reaches a configuration c , the history *parserHF-conf-history* c is a prefix of some marked word of the Parser. This definition of the correct-prefix-property states a weaker condition than the nonblockingness property because it does not take the fixed scheduler *parserHF-conf-fixed* c into account. The word *parserHF-conf-history* c @ *parserHF-conf-fixed* c (where the possibly trailing \$ is dropped) is an unmarked word according to our definition of the unmarked language and is therefore required, by the language based nonblockingness, to be extendable to a marked word of the Parser. Apparently, in [325] the fixed scheduler is not considered to be a part of the unmarked word, which is not problematic in the field of parsing theory because fixing a part of the input has no relevant side-effect there. To verify the stronger property of nonblockingness for the constructed LR(1)-Parser, we strengthened [325, Volume II, Theorem 9.1, p. 289] as well as the required lemmas and proofs. In fact, we started our attempts to apply parsing theory for enforcing the nonblockingness property when realizing the similarity of the nonblockingness property and the correct-prefix-property.

5.2.3. Convert LR(1)-Parser into Equivalent DPDA

In building block F, we translate the obtained LR(1)-Parser into a DPDA.

Figure 5.5: «Convert LR(1)-Parser Into Equivalent DPDA»



We introduced the three formalisms of Parsers, EDPDA, and DPDA in chapter 2|p.15 where we have also explained that Parsers have additional capabilities compared to EDPDA (that is, they can determine the end of the input by fixing the processing terminator, they can execute more than one event in one step, and they are allowed to fix a prefix of the input without consuming it immediately in so called top rules) and EDPDA have additional capabilities compared to DPDA (that is, the stack-pop component of an edge may not be a word of length 1).

Conceptually, we execute the following three steps in this building block. Firstly, we translate the input Parser into a Parser not making use of these additional capabilities in the building blocks F1 and F2. Secondly, we convert the obtained Parser into an EDPDA in building block F3. Finally, we translate the resulting EDPDA into a DPDA in building blocks F5 and F6.

Subsequently, we introduce these building blocks in more detail.

— *Building Block F1: $F_{\text{Parser-Remove-Input-Terminator-Usage}}$*

We translate the input LR(1)-Parser into a Parser that does not ever fix the processing terminator. That is, we ensure that the processing terminator does not occur in any rule of the resulting Parser. Technically, we simply remove all rules containing the processing terminator, which preserves the important properties because, whenever the input LR(1)-Parser can apply such a removed rule, it will not execute further events and will mark the word executed so far after a finite number of steps. Basically, once the LR(1)-Parsers fixes the processing terminator, it is guaranteed to only execute further reduce rules afterwards. Consider for example the rules given in Example 5.4|p.74 where the first four rules would remain and the last two rules would be removed in this step; these last two rules only reduce the stack of the Parser into the final stack $\$A$. The marking stack values of the resulting Parser are given by those elements q such that some rule r has been removed satisfying *rule-stack-pop* $r = w @ [q]$. That is, q is a marking stack value iff the finishing reduce sequence can start in q . Note that it is sufficient to test for q instead of testing whether the removed rule r is applicable by testing whether the current stack starts with *rule-stack-pop* r .

To the best of our knowledge, we are the first to observe that the LR(1)-Parser can be simplified this way.

The operation preserves the marked and unmarked language as well as the satisfaction of the properties of determinism and nonblockingness.

→ Building Block F2: $F_{\text{Parser-Remove-Top-Rules}}$

We ensure that every rule has an empty *rule-scheduler-push* component resulting in empty fixed scheduler components throughout any initial derivation of the resulting Parser. This is achieved by adapting the nonterminals on the stack such that they also record the fixed scheduler of length at most 1 (that is, the events observed but not executed). Additionally, we construct new rules such that the behavior is preserved as follows. When the input Parser applies a rule $w_1 @ [q_1] | a \rightarrow w_2 @ [q_2] |$ (executing the event a), we create the rules (a) $w_1 @ [(q_1, \text{None})] | a \rightarrow w_2 @ [(q_2, \text{None})] |$ for the case where the event *was not fixed* already and (b) $w_1 @ [(q_1, a)] | \rightarrow w_2 @ [(q_2, \text{None})] |$ for the case where the event *was fixed* already. When the input Parser applies a rule $w_1 @ [q_1] | a \rightarrow w_2 @ [q_2] | a$ (possibly fixing the event a), we create the rules (a) $w_1 @ [(q_1, \text{None})] | a \rightarrow w_2 @ [(q_2, a)] |$ for the case where the event *was not fixed* already and (b) $w_1 @ [(q_1, a)] | \rightarrow w_2 @ [(q_2, a)] |$ for the case where the event *was fixed* already.

The operation preserves the marked and unmarked language as well as the satisfaction of the properties of determinism and nonblockingness. Furthermore, only executing rules are applicable to marking configurations in the output. This will be exploited later on for establishing the satisfaction of livelock freedom.

→ Building Block F3: $F_{\text{Parser} \rightarrow \text{EDPDA}}$

We translate the Parser, which is essentially an EDPDA after the application of the two previous steps, into an EDPDA. The edges of the resulting EDPDA correspond bijectively to the rules of the input Parser as follows using the operation *rev* for the reversal of a word: an executing rule of the form $w_1 @ [q_1] | a \rightarrow w_2 @ [q_2] |$ results in an edge $\langle q_1, a, \text{rev } w_1, \text{rev } w_2, q_2 \rangle$ and, similarly, a nonexecuting rule of the form $w_1 @ [q_1] | \rightarrow w_2 @ [q_2] |$ results in an edge $\langle q_1, \text{None}, \text{rev } w_1, \text{rev } w_2, q_2 \rangle$.

The operation preserves the marked and unmarked language as well as the satisfaction of the properties of determinism, nonblockingness, and nonexistence of nonexecuting edges applicable to marking configurations.

→ Building Block F4: $F_{\text{EDPDA} \rightarrow \text{DPDA}}$

We convert the input EDPDA obtained from before into a DPDA by restricting the possible patterns of edges to those that are allowed in DPDA. The conversion is executed using $F_{\text{EDPDA} \rightarrow \text{DPDA}}$, which applies the operations from the two following building blocks F5 and F6 in succession.

The operation preserves the marked and unmarked language as well as the satisfaction of the properties of determinism and nonblockingness. Also, we establish in F6 that the resulting DPDA has no livelocks from marking states, that is, there is no infinite derivation starting in a marking state not executing events.

→ Building Block F5: $F_{\text{EDPDA-Remove-Nil-Popping-Edges}}$

We translate the input EDPDA into an output EDPDA without edge with empty *edge-pop* component. Such edges originate from rules that are primarily created

in our algorithm from shift rules of the LR(1)-Parser. Reduce rules on the other hand have (besides the top-most nonterminal) further nonterminals in their *rule-stack-pop* component unless they are created from productions with empty right hand side. We make use of the fact that the *epda-eos* element is never removed from the bottom of the stack and that, hence, there is always a top-element on the stack that can be tested for. For each edge with an empty *edge-pop* component and for each element x from the stack-alphabet *epda-gamma* an edge is added to the resulting EDPDA. The resulting edge is adapted to pop and push the element x from and to the stack, that is, x is an element borrowed from the stack during the application of such a replacement edge.

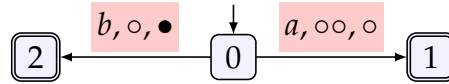
The operation preserves the marked and unmarked language as well as the satisfaction of the properties of determinism, nonblockingness, and nonexistence of nonexecuting edges applicable to marking configurations. However, note that many of the edges added will be inaccessible because for a given state q only a small subset of stack-symbols may occur in reachable configurations together with state q as a top-stack. Hence, similarly to section 5.2.1|p.71, we may construct an approximation of the state space to obtain for each state an overapproximation of the possible top-stacks to reduce the number of inaccessible generated edges.

→ *Building Block F6: $F_{EDPDA-Remove-Mass-Popping-Edges}$*

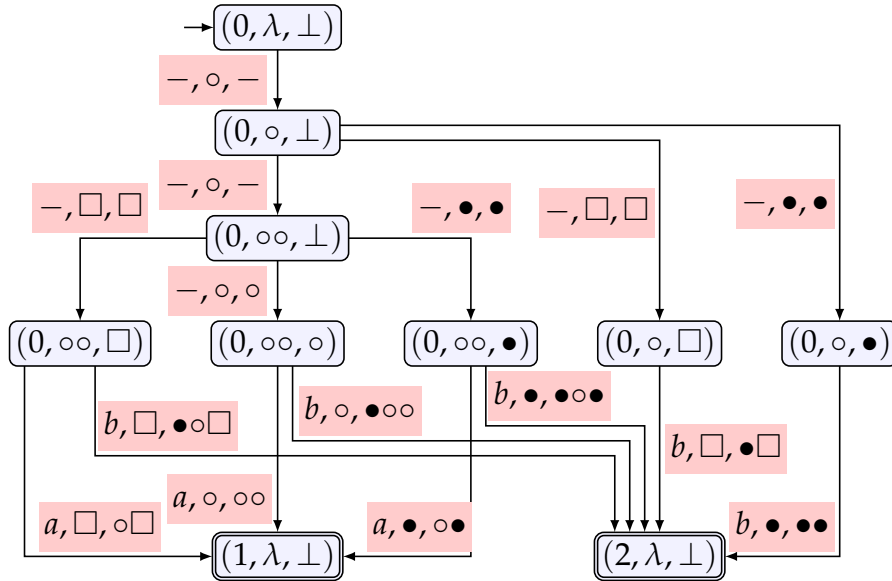
We translate the input EDPDA into an output EDPDA where every edge has an *edge-pop* component of length one. The construction procedure, as explained in the following example, is much more involved compared to the operation $F_{EDPDA-Remove-Mass-Pushing-Edges}$ for replacing edges that are *pushing* more than two elements at once.

Example 5.5: «Removal of Edges Popping More Than One Stack-symbol»

Input:



Output:



Discussion: The naive construction idea is to replace every edge with an *edge-pop* component of length greater than one by a corresponding sequence of edges with *edge-pop* components of length one. However, to preserve determinism, different edges leaving a common source state must be replaced collectively to ensure that two applicable edges with common source state still make incompatible requirements on either the event to be executed or the element to be popped from the stack. The construction also needs to ensure that the *epda-eos* element is never removed from the bottom of the stack.

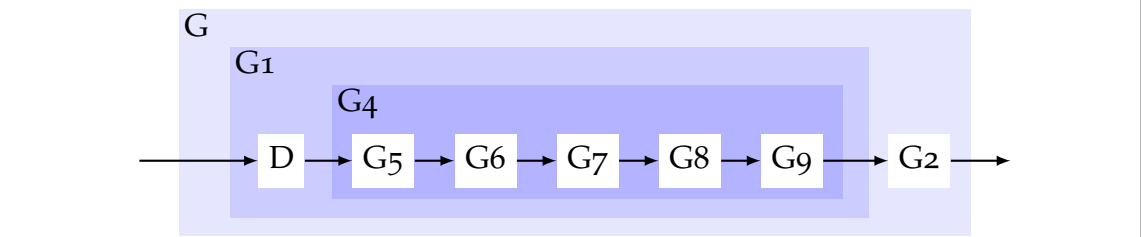
The operation preserves the marked and unmarked language as well as the satisfaction of the properties of determinism and nonblockingness. Also, we establish that the resulting DPDA has no livelocks from marking states. However, also in this step, we add edges of which many are likely to be inaccessible due to stack-prefixes that never occur in accessible configurations. As in the previous building block, we may make use of a state space approximation to reduce the number of inaccessible edges generated.

Overall, in the concrete building block F, we translated the LR(1)-Parsers obtained from before into DPDA preserving their marked language and enforcing the satisfaction of the nonblockingness property. This conversion is the last required step of the conversion procedure from DPDA via LR(1)-CFG via LR(1)-Parsers to DPDA showing the equivalence of these formalisms. In the next subsection, we solve the problem of possibly inaccessible edges generated in the building blocks F5 and F6.

5.2.4. Enforce Accessibility on DPDA

In building block G, we remove all inaccessible states and edges from the input DPDA using the operation $F_{DPDA-Enforce-Accessible, Opt}$.

Figure 5.6: «Enforce Accessibility for DPDA»



In building block G1 (described below), we determine the edges of the input DPDA that are required to preserve the marked language. Then, in building block G2, we (a) remove all edges from the input DPDA that are not determined to be required and (b) also remove all states (besides the initial state, which is always retained) that are not the source or target of a retained edge. That is, we reduce the problem of enforcing accessibility for states and edges to the problem of determining the required edges of the input DPDA.

→ *Building Block G1: $F_{DPDA-Determine-Accessible-Edges}$*

We identify the required edges of the input DPDA by translating it using building block D (see subsection 5.2.1|p.69) into an LR(1)-CFG, which is accessible by construction. Actually, we skip the building block D2 implementing the function $F_{SDPDA-Enforce-Unique-Marking-Early}$ for enforcing a unique marking during this step.

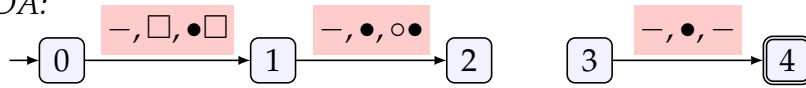
Then, in building block G4, we apply a reversal procedure, which relies on the fact that each initial marking derivation d_1 of the LR(1)-CFG using the semantics $cfgLM$ corresponds to an initial marking derivation d_2 of the input DPDA. Moreover, the edges used in d_2 can be computed from the productions used in d_1 . This correspondence is exploited by our reversal operations, which convert the productions of the obtained LR(1)-CFG back into edges of the original input DPDA. Again, recall that such a reversal cannot be pursued to obtain a restriction of the input DPDA that satisfies the nonblockingness property because there may be edges of the input DPDA that are required for maintaining the marked language while they also occur in initial derivations that cannot be extended to marking derivations. See Example 5.1|p.67 where every edge of the DPDA C participates in an initial marking derivation as well as in an initial derivation that is eventually deadlocked in a non-marking state. However, for enforcing accessibility, it is sufficient to determine those edges that are relevant to the marked language: for these edges at least one corresponding production remains in the obtained LR(1)-CFG.

In each of the building blocks G5, G6, G7, G8, and G9, we revert one step from the conversion of DPDA into LR(1)-CFG as given by the building blocks D3, D7, D6, D5, and D4, respectively. In each case, the reversal is possible because two distinct edges of the input DPDA resulted in disjoint sets of edges/productions of the output DPDA/CFG of that step.

For the conversion of the SDPDA into a CFG, we may either use the operation $F_{SDPDA \rightarrow LR(1), Opt}$ or the operation $F_{SDPDA \rightarrow CFG, Std}$. However, as the following example demonstrates, we have to make sure that we do not consider productions that are not accessible in the obtained CFG for the used $cfgLM$ semantics.

Example 5.6: «Necessity of Left-Most Accessibility for Productions»

Input SDPDA:



CFG obtained from applying $F_{SDPDA \rightarrow CFG, Std}$ with initial nonterminal $L_{0, \square}$:
(excerpt of productions)

$$L_{0, \square} \longrightarrow L_{1, \bullet, 4} \quad L_{1, \bullet, 4} \longrightarrow L_{2, \circ, 3} \quad L_{3, \bullet, 4} \longrightarrow$$

Discussion: We included only some of the productions actually generated by $F_{SDPDA \rightarrow CFG, Std}$. The nonterminal $L_{2, \circ, 3}$ cannot be replaced by a list of events because there is no derivation of the input DPDA from state 2 to state 3 popping \circ in the last step. Hence, we detect that $L_{3, \bullet, 4}$ is not accessible in a left-most derivation in this CFG and, moreover, we detect that states 3 and 4 are inaccessible because also no other accessible production makes use of them.

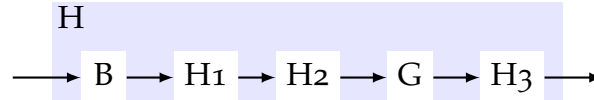
When using the operation $F_{SDPDA \rightarrow LR(1), Opt}$ as mentioned above, we obtained a CFG that contains only left-most accessible productions in this sense. However, when using the operation $F_{SDPDA \rightarrow CFG, Std}$, we have to compute the left-most accessible nonterminals in an additional step.

5.3. CONCRETE BUILDING BLOCK FOR REDUCING CONTROLLABILITY TO NONBLOCKINGNESS FOR DETERMINISTIC PUSHDOWN AUTOMATA

In building block H, we reduce the problem of enforcing the satisfaction of controllability on DPDA to the problem of enforcing the satisfaction of the nonblockingness property using the operation $F_{DPDA-Reduce-Controllable}$. Note, we do not use the terminology of enforcing the satisfaction of controllability because the operation does not preserve the *required* satisfaction of nonblockingness from its input DPDA to the output DPDA (see also Corollary 4.2|p.53). The well-known DFA-based operation for enforcing controllability behaves similarly in terms of not preserving the satisfaction of the nonblockingness property, but its satisfaction is not a required assumption for soundness in this restricted DFA-based setting.

In the presented reduction, we make use of three previously introduced building blocks: building block B from section 5.1|p.64 is used to construct the product automaton of a DPDA and a DFA, building block G from subsection 5.2.4|p.80 is used to remove inaccessible states and edges from a DPDA, and building block D2 from section 5.2.1|p.70 is used to ensure uniqueness of initial marking derivations for every marked word of a DPDA.

Figure 5.7: «Enforcing Controllability for DPDA»



The presented reduction follows the construction presented in [301, 302, 309] (see section A|p.225 for a remark on coauthorship). However, in this thesis, we deviate from these earlier monolithic definitions and provide a modular construction, which is also slightly modified and which allows for modular proofs and explanations. Also note that Christopher Griffin worked on enforcing controllability earlier and was able to determine controllability problems in a similar way, but he removed them not least restrictively as discussed in detail in subsection 8.1.7|p.180 where we also compare his approach to the reduction presented here.

We adapt the input DPDA C_0 in the four following steps into an equivalent DPDA before being able to remove states with controllability problems without undesired side-effects in the final step. That is, the first four steps remove ambiguity in the states of the DPDA w.r.t. the property of controllability.

→ *Building Block B: $F_{DPDA-DFA-Product}$*

We apply the operation $F_{DPDA-DFA-Product}$ from section 5.1|p.64 to obtain the DPDA C_1 by constructing the product automaton of the DPDA C_0 and the plant DFA P . By construction, the states of C_1 are of the form (q, p) where p is a state of the plant P and, moreover, whenever C_1 can enter a state (q, p) using an initial derivation d_1 , there is a corresponding initial derivation d_2 of P entering p .

A state (q, p) in C_1 *may* have a controllability problem if some edge exiting p can execute an event $u \in \Sigma_{uc}$ while for some stack-element X there is no edge exiting (q, p) executing u and popping X from the stack. However, we should only consider stack-elements X with which (q, p) is actually accessible as a top-stack. Consequently, we refine the analysis of missing edges in the next steps.

Note, in the context of our overall controller synthesis algorithm this step does not modify the marked language because we restrict the controller candidate's marked language already in the step preceding the fixed-point computation to a subset of the marked language of the plant P .

→ *Building Block H1: $F_{DPDA-Observe-Top-Stack}$*

We apply the operation $F_{DPDA-Observe-Top-Stack}$ on C_1 and obtain the DPDA C_2 by distinguishing in each state between the subsequent behavior for the different top-stacks. Recall that in DPDA, the edges have an *edge-pop* component of length precisely one, that is, they consider in their step-relation precisely the single top-stack element.

Technically, the resulting DPDA C_2 is a bipartite automaton consisting of *main* states, which are the states of C_1 , and additional *auxiliary* states, which are pairs $((q, p), X)$ containing a state of C_1 and a stack-element X . Firstly, edges of the form $\langle ((q, p), None, [X], [X]), ((q, p), X) \rangle$ test in C_2 at runtime for the current top-stack X in state (q, p) saving it explicitly in the target state $((q, p), X)$. Secondly, all edges of C_1 starting in state (q, p) requiring a top-stack X are changed to start in the auxiliary state $((q, p), X)$ in C_2 . Finally, main states are not marking and an auxiliary state $((q, p), X)$ is marking if (q, p) was marking in C_1 . As a consequence, a state $((q, p), X)$ is only reachable with a top-stack of X and every edge exiting that state is always applicable in a branching semantics where the next event to be executed is not given by schedule.

We identify a state $((q, p), X)$ in C_2 to have *potentially* a controllability problem if some edge exiting p can execute an event $u \in \Sigma_{uc}$ in P while there is no edge exiting $((q, p), X)$ executing u (hence, we were able to drop the condition that the edge is popping a certain element from the stack). However, this procedure is not sufficient because it would be admissible for $((q, p), X)$ to execute this event after an application of any finite number of non-executing edges. Consequently, we refine the analysis of missing edges in the next steps to ensure that localization of controllability problems requires only the analysis of a single state with its adjacent edges.

This building block H1 was the starting point for the development of building block H where the following two additional steps are required for soundness.

→ *Building Block H2: $F_{DPDA-Enforce-Unique-Marking-Late}$*

We apply the operation $F_{DPDA-Enforce-Unique-Marking-Late}$ to obtain from the DPDA C_2 the DPDA C_3 ensuring that only auxiliary states that are only exited by executing edges (called stable states) must be checked for missing edges.

Technically, we ensure that there is a unique initial marking derivation for every marked word and that any sequence of non-executing steps cannot pass through a marking state, that is, a marking state can only be exited by an executing edge. For this purpose, we facilitate the building block D2 with its operation $F_{SDPDA-Enforce-Unique-Marking-Early}$ from section 5.2.1|p.70, but we adapt the set of marking states to only contain stable states. In fact, after applying $F_{SDPDA-Enforce-Unique-Marking-Early}$ each state q is flagged by whether any initial derivation reaching a configuration containing q has been in a marking state before reaching q only executing internal steps afterwards. Hence, we obtain all marking states of the DPDA, all states that can be visited from such a state not executing further events and, finally, remove from this set all non-stable states. While the operation $F_{SDPDA-Enforce-Unique-Marking-Early}$ ensures that any word is not marked twice, the operation $F_{DPDA-Enforce-Unique-Marking-Late}$ then also ensures that every word is marked as late as possible.

We identify a stable auxiliary state $(m, ((q, p), X))$ (where m is an additional annotation introduced by $F_{DPDA-Enforce-Unique-Marking-Late}$ that is not relevant subsequently) in C_3 to have *potentially* a controllability problem if some edge exiting p can execute an event $u \in \Sigma_{uc}$ while there is no edge exiting $(m, ((q, p), X))$ executing u (that is, compared to before, we limited our analysis to the stable auxiliary states). However, this procedure is not sufficient because inaccessible states $(m, ((q, p), X))$ cannot have controllability problems.

→ *Building Block G: $F_{DPDA-Enforce-Accessible, Opt}$*

We apply the operation $F_{DPDA-Enforce-Accessible, Opt}$ from subsection 5.2.4|p.80 to obtain from the DPDA C_3 the DPDA C_4 removing all states and edges that are inaccessible in C_3 .

For our controller synthesis algorithm, this step is necessary to reliably detect that the fixed point has been obtained. This detection is based on the information whether the operation $F_{DPDA-Restrict-to-Controllable-States}$ from the next paragraph removed at least one state where the DPDA had a controllability problem. Hence, we must ensure that the DPDA C_0 has a controllability problem if and only if some state is removed in the next step.

Similarly, when enforcing controllability for DFA, any removal of a state guarantees the modification of the given controller candidate because (a) accessibility of the input is assumed as well and (b) inaccessibility is not introduced prior to the detection and removal of states with controllability problems.

Also note, inaccessible states and edges that are removed in this step have been added to the DPDA using the operation $F_{DPDA-Observe-Top-Stack}$. With our state space approximation (also used in F6 in section 5.2.3|p.79, D3 in section 5.2.1|p.71, and F5 in section 5.2.3|p.78), we may also decrease the number of inaccessible

states and edges added there, which would improve the performance of this step in the future. Also note, due to the accessibility of the input DPDA, we know that the set of possible top-stacks of non-marking states is given by the set of top-stacks tested for in exiting edges.

→ *Building Block H3: $F_{DPDA-Restrict-to-Controllable-States}$*

Finally, we apply the operation $F_{DPDA-Restrict-to-Controllable-States}$ to obtain from the DPDA C_4 the resulting DPDA C_5 by detecting and removing accessible states with controllability problems with their adjacent edges. Such states are selected by considering auxiliary stable states $(m, ((q, p), X))$ for which some edge exiting p can execute an event $u \in \Sigma_{uc}$ while there is no edge exiting $(m, ((q, p), X))$ executing u . The suitability of this detection and removal mechanism follows from the previously applied operations.

→ *Building Block H: $F_{DPDA-Reduce-Controllable}$*

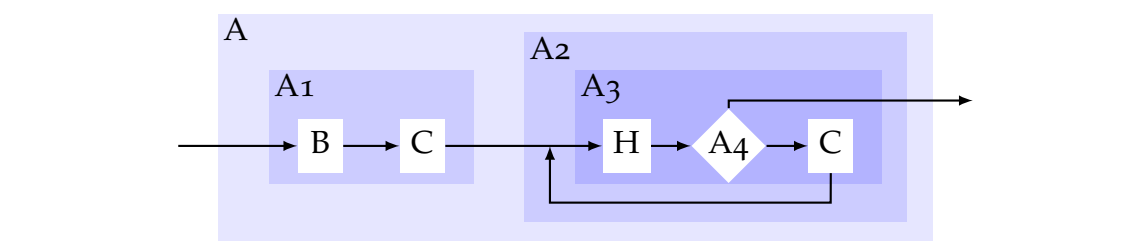
The operation $F_{DPDA-Reduce-Controllable}$ given by building block H combines the operations used in these five steps and reduces controllability for a nonblocking DPDA possibly invalidating the nonblockingness property. By only considering stable auxiliary states, we do not ensure that all unmarked words with a controllability problem are removed, which is achieved by the DFA-based algorithms. From the perspective of the DES generated from the DPDA C_5 , we implement *Enforce-Marked-Controllable-Subset* from Definition 4.9|p.52.

We conclude that the disambiguation carried out in the building blocks B, H1, and H2 is necessary to ensure that states with controllability problems can be removed without preventing initial marking derivations without controllability problems as well. Also, the accessibility established in building block G and the nonblockingness as assumed for the initial input DPDA C_0 are necessary to ensure that the removal of a state implies that the marked language has been restricted (that is, the controllability problem identified is accessible and relevant to the marked language).

5.4. CONCRETE SYNTHESIS ALGORITHM AS AN INSTANTIATION OF THE ABSTRACT SYNTHESIS ALGORITHM

The overall concrete controller synthesis algorithm for DPDA specifications and DFA plants is now given by $F_{DPDA-DFA-Construct-Controller}$ as a combination of the previously introduced building blocks.

Figure 5.8: «Overview of Controller Construction»



In building block A, we apply the operation $F_{DPDA-DFA-Construct-Controller}$ to construct the desired DPDA controller for a given DPDA specification S , a given DFA plant P , and a given set of uncontrollable events Σ_{uc} along the lines of the DES based controller synthesis algorithm presented in Theorem 4.12|p.56.

In building block A1, we apply the operation $F_{DPDA-DFA-Construct-Controller}^{fp-start}$ which corresponds to the DES-based operation $FP-Iterator-Init$ from Definition 4.10|p.53. In this first step, we apply building blocks B and C to obtain the initial controller candidate in the form of a DPDA, which satisfies the specification and the nonblockingness property but which does not satisfy controllability in general. In building block B we use $F_{DPDA-DFA-Product}$ to combine the DPDA specification S and the DFA plant P into a DPDA generating $inf(epda-to-des S)(epda-to-des P)$. Then, in building block C we use $F_{DPDA-Enforce-Nonblocking, Opt}$ to enforce the satisfaction of the nonblockingness property on this DPDA as in $FP-Iterator-Init$.

In building block A2, we apply the operation $F_{DPDA-DFA-Construct-Controller}^{fp}$ which corresponds to the DES-based fixed-point iterator $Compute-FP$ from Definition 4.2|p.47, to obtain the desired DPDA controller. In this operation, we apply in building block A3 the one-step operation $F_{DPDA-DFA-Construct-Controller}^{fp-one}$ which corresponds to the DES-based operation $FP-Iterator-Step$ from Definition 4.11|p.54, until the termination of the fixed-point computation is detected. This detection is recognized in building block A4 if and only if no state has been removed in the building block H3 contained in building block H. Analogously to the fixed-point iterator $Compute-FP$, we reduce controllability to nonblockingness in building block H and enforce nonblockingness in building block C alternatingly with the test for termination in between. Recall that re-establishing nonblockingness in the second step may invalidate controllability of the obtained controller candidate.

By implementing the fixed-point iterators used in the DES-based controller synthesis algorithm from Theorem 4.12|p.56 in the described way, we obtain as a direct result that the DES generated by the obtained DPDA controller is a solution to the DES-based supervisory control problem from Definition 3.7|p.35. Moreover, we state that the operation $F_{DPDA-DFA-Construct-Controller}$ solves the concrete supervisory control problem from Definition 3.11|p.39 up to termination.

Theorem 5.1: «Soundness of the DPDA-Based Synthesis Algorithm»

Whenever the operation $F_{DPDA-DFA-Construct-Controller}$ terminates and returns a DPDA, then this DPDA is accessible, accessible in the closed-loop, language-minimal, and a least restrictive satisfactory controller and, hence, a specific solution to the automata-based concrete supervisory control problem (see section 3.2|p.36). Moreover, whenever the operation $F_{DPDA-DFA-Construct-Controller}$ terminates and returns no DPDA, then there is no solution to the automata-based concrete supervisory control problem.

However, the presented concrete controller synthesis algorithm does not terminate in general as already discussed in section 4.4|p.56 for the abstract controller synthesis algorithm.

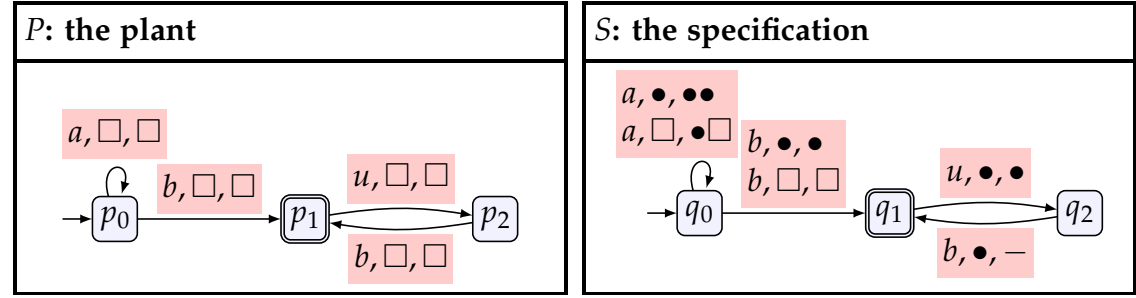
5.5. ON THE TERMINATION OF THE CONCRETE CONTROLLER SYNTHESIS ALGORITHM

Recall that the abstract controller synthesis algorithm from Theorem 4.12|p.56 does not terminate on every DES plant, DES specification, and set of uncontrollable events Σ_{uc} as discussed in section 4.4|p.56. In general, the nontermination of the abstract controller synthesis algorithm can be expected because the formalism of DESs is expressive enough to also serve as a model for more complex formalisms such as Turing machines, which are intractable according to well-known undecidability results. On the one hand, the DES inputs S and P used in the examples in section 4.4|p.56 are realizable by DPDA and DFA, respectively. On the other hand, each fixed-point iterator employed in the abstract controller synthesis algorithm is implemented by the corresponding building block in our concrete controller synthesis algorithm in terms of the DES generated by *epda-to-des*. Hence, we conclude that the concrete controller synthesis algorithm does not terminate on all inputs as well. However, we state that the concrete controller synthesis algorithm does not introduce additional partiality of the algorithm by not terminating on automata realizing DES for which the abstract controller synthesis algorithm terminates. Also note that this property is obtained from the fact that all suboperations of the building block A terminate on all inputs that occur during any application of the controller synthesis procedure.

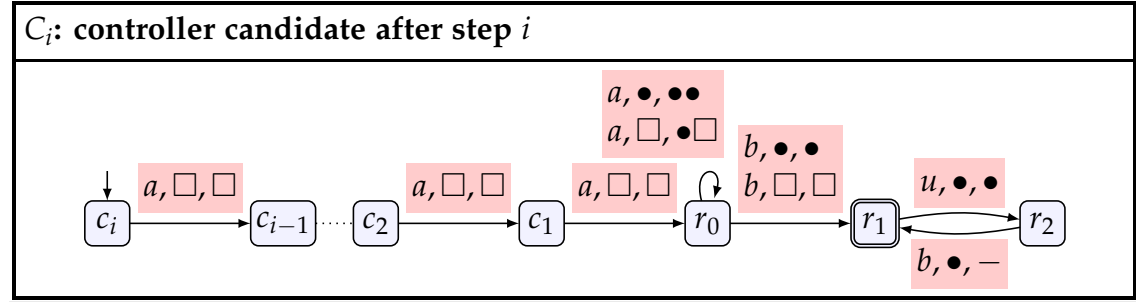
As a first step, we observed that the abstract controller synthesis algorithm terminates, as discussed in subsection 4.4.1|p.56, for the restricted class of DFA plants and DFA specifications. Due to the direct correspondence of the concrete controller synthesis algorithm presented in this chapter (see Theorem 5.1|p.86 above) with the abstract controller synthesis algorithm, we can conclude that our concrete controller synthesis algorithm terminates in this restricted setting (assuming that the informal proofs for termination from related work that are cited in subsection 4.4.1|p.56 are valid).

As a second step, we discussed in subsection 4.4.2|p.58 two examples demonstrating that the abstract and the concrete controller synthesis algorithms do not terminate for arbitrary DPDA specifications. To the best of our knowledge and supported by these two examples, the nontermination stems from specifications stating unrealistic requirements, which can never be satisfied by controllers due to controllability such as a lower/upper bound on uncontrollable events to be executed. The abstract and the concrete controller synthesis algorithms do not terminate for the examples provided because they fail to discard parts of the controller candidate that violate the controllability property because they rely on an iterated one-step unfolding procedure that is not sufficient for DPDA specifications. We now reconsider the second more involved Example 4.4|p.59 by presenting the DPDA realizations of the intermediate controller candidates. Note, the actual DPDA controller candidates obtained from applying our concrete controller synthesis algorithm are equivalent but typically slightly larger.

Figure 5.9: «DPDA Realizing Controller Candidates of Example 4.4|p.59»

 Inputs: $\Sigma_{uc} = \{u\}$ and $\Sigma_c = \{a, b\}$


Discussion: From Example 4.4|p.59, we conclude that the first controller candidate C_0 obtained when evaluating $F_{DPDA-DFA-CC} S P \Sigma_{uc}$ is equivalent to S . The concrete controller synthesis algorithm then generates an infinite sequence of controller candidates C_i (see below). Instead of expanding the number of events a executed before reaching the event b it should prevent the event b , which would lead to an empty marked language.



Chapter 5|p.61

(Concrete Controller Synthesis Algorithm for DPDA)

We instantiated the compositional abstract controller synthesis algorithm from chapter 4|p.45 to obtain an automata-based concrete controller synthesis algorithm. This algorithm is also a fixed-point algorithm and consists of building blocks implementing the corresponding fixed-point iterators of the abstract controller synthesis algorithm. We verified its soundness w.r.t. the abstract and the concrete supervisory control problem as stated in Definition 3.7|p.35 and Definition 3.11|p.39, respectively. Under the condition of termination, the obtained DPDA controller is a least restrictive satisfactory controller, which is, in addition, accessible, accessible in the closed loop, and has the infimal marked language among such solutions. The verification of the algorithm is based on its modularity given by the various building blocks for which we determined and verified suitable and composable input-output specifications.

In the next chapter, we detail on our usage of Isabelle for quality assurance.

6

Isabelle-based Formal Quality Assurance

In the two previous chapters, we introduced our abstract and concrete controller synthesis algorithms at an informal level. We have only provided basic intuitions about the involved constructions and the ultimately obtained results. In this chapter, we focus on our workflow, which is based on the theorem prover Isabelle and which thereby guarantees soundness for the algorithms on the one side and, up to programming errors, for the prototype implementation CoSy, discussed in the next chapter, on the other side.

For readers not familiar with Isabelle, we provide a discussion on the advantages and disadvantages of using theorem provers such as Isabelle for ensuring quality. In the trade-off discussed, we gain trustworthy (that is, error-free and omission-free) proofs as a great benefit, but we have to face the theorem prover Isabelle as an infallible adversary that only accepts and also requires the most detailed proofs. However, the ongoing development has reached a state where the tedious manual creation of such highly detailed proofs is partially alleviated by additional heuristics, external tools, and visualizations of intermediate proof states.

We also provide some selected aspects of our Isabelle-based framework established for the formalization of the correctness proofs of the abstract and concrete controller synthesis algorithms available at [303]. However, we only present the structure of our framework and formalization also providing some proof-code measures, discuss some general purpose proof rules, and provide the basic ideas of some parts of our framework for semantics.

Since Isabelle proofs with their high level of detail often lack accessibility, we end this chapter by providing the proof idea of one of our more central proofs. In particular, the outlined proof shows that the determinism of a given SDPDA is properly transferred to LR(1)-CFG, which is an important step during the verification of our concrete controller synthesis algorithm.

CONTENTS OF THIS CHAPTER

6.1|p.91 *Formal Methods for Quality Assurance*

We discuss the usage of the theorem prover Isabelle, which we employ to ensure quality, and compare this tool-based approach with conservative manual approaches. We use Isabelle to ensure quality in terms of, firstly, the theorems that state the soundness of our abstract and concrete controller synthesis algorithms (see Theorem 4.12|p.56 and Theorem 5.1|p.86) and, secondly, the fundamental software quality of our prototype implementation discussed in chapter 7|p.141.

6.2|p.96 *Isabelle-based Framework of Definitions and Properties*

We address important aspects of our Isabelle-based framework used heavily in our formalization of our two central theorems. For the framework and formalization, we present information on their general structure and their size. We then focus on our framework, present further additional general purpose results, and then detail on our locale-based framework of branching and linear semantics in which we also cover various aspects of semantics relevant in this thesis. Moreover, we discuss custom techniques based on locales that allow for the connection of semantics of possibly different formalisms. Finally, we introduce for EPDA and Parsers further semantics and discuss their advantages and disadvantages for certain verification tasks.

6.3|p.116 *Isabelle-based Verification of the Translation of Deterministic Pushdown Automata into LR(1)-context-free Grammars*

We provide a proof sketch for the property that an SDPDA can be converted into a language equivalent LR(1)-CFG by means of the operation $F_{SDPDA \rightarrow LR(1), Std}$ discussed in subsection 5.2.1|p.69. In particular, we consider the preservation of the determinism of the given SDPDA to the LR(1)-CFG. This single proof sketch demonstrates (in connection to our explanations in section 6.1|p.91) how proofs that have been formalized in theorem provers such as Isabelle can be prepared to allow readers their comprehension with minimal effort. In this presentation, we follow the style of classical presentations of pen-and-paper proofs. We are not aware of alternate presentations of (even informal) proofs of this property, which is required for the overall soundness of our concrete controller synthesis algorithm.

6.1. FORMAL METHODS FOR QUALITY ASSURANCE

In this section, we briefly discuss the steps of modelling, specifying, and verifying that we followed to obtain the formal results presented in this thesis. By focussing on the concrete controller synthesis algorithm from chapter 5|p.61, we motivate the requirement for formal methods in general to ensure software quality. We then provide a general discussion on the benefits of applying formal methods for this aim. Finally, we focus on the interactive theorem prover Isabelle [265], which we employ throughout the presented workflow to ultimately obtain a formally verified algorithm.

—• *Quality Assurance by Formalization in This Thesis*

In this thesis, we developed the concrete controller synthesis algorithm presented in chapter 5|p.61. However, our Java-based implementation of this algorithm, as used in section 7.5|p.157, may contain any number of bugs. This gap between verified mathematical constructions and unverified implementations has been noticed before. However, because we verified the underlying algorithm, we can be sure that every bug can be fixed once it is detected. Moreover, the removal of these bugs will, due to the compositional construction of our algorithm and the closely corresponding implementation, require the modification of a small number of lines in each case. This is not the case for arbitrary programs where detected bugs may result in realizing that the entire approach is incorrect and that the problem can either not be solved at all or that the entire implementation has to be changed. Also see subsection 8.3.1|p.209 for a discussion on the usage of code-generation techniques in the future.

We tested our implementation thoroughly using unit tests with code coverage and also considered various examples, like those reasonably realistic examples discussed in section 7.3|p.147, for integration testing. Hence, even if the translation from the presented algorithm to Java-code, the compilation of this Java-code to byte-code, and the execution of this byte-code in a virtual machine may introduce errors, we excluded the most fundamental kinds of bugs already by formalizing and verifying the correctness of our algorithm. The exclusion of these kinds of errors is an immediate achievement of formal verification and a central contribution of this thesis.

—• *Quality Assurance by Formalization Using a Three-step Workflow*

The exclusion of fundamental errors is the key motivation for formal verification as just pointed out. We established this kind of quality following the subsequently discussed three-step workflow of modelling the actual behavior, specifying the expected behavior, and verifying the correspondence between actual and expected behavior.

Providing formal definitions (that is, in a language with predefined syntax and semantics) results in a nonambiguous model. This nonambiguity then allows for *team-based joint inspection* as well as for the further steps towards specification

and verification. Definitions in natural language or pseudocode are, by this explanation, not formal and ambiguous. In this thesis, we omitted our formal Isabelle-based constructions and introduced the building blocks only informally in chapter 5|p.61 instead. Note, the Java-based implementation of our concrete controller synthesis algorithm was only obtained from these definitions.

To obtain a specification, we express desired properties of our definitions in a formal way in the form of theorems using a formal language, which is typically based on the language for definitions and a suitable logic such as higher order logic (HOL). Extending the definitions with such formal theorems then results in a description of the expected behavior of the given definitions. In this thesis, we have ultimately described the expected behavior of the concrete controller synthesis algorithm by means of Theorem 5.1|p.86. However, this theorem depends on various other definitions of which some have been discussed when introducing the automata-based supervisory control problem in Definition 3.11|p.39. Note, the process up to this point resulting in definitions and theorems, even without a follow-up verification of these theorems, constitutes a reasonably nonambiguous specification of our concrete controller synthesis algorithm that allows for a *team-based joint inspection* and that is sufficient for determining the suitability of the algorithm. That is, by inspecting Theorem 5.1|p.86 together with the notions this theorem depends on (such as the supervisory control problem from Definition 3.11|p.39) is sufficient for determining that the given algorithm is *meant* to solve (i.e., a verification is called for) the *expected* problem (i.e., the problem that is to be solved).

Finally, the verification of the correspondence of actual behavior (given by the definitions) and the expected behavior (given by the theorems) serves multiple purposes. Firstly, a *trustworthy proof* guarantees the absence of fundamental errors in the definitions (of course only w.r.t. the properties stated in the theorems) and may serve as a starting point for further developments in the future if the employed syntax and semantics are sufficiently established. Secondly, an *easily accessible proof* can be inspected to deepen the understanding of both, the definitions and the theorems. Thirdly, an *adaptable proof* allows for good maintainability when definitions and theorems are altered. These considerations constitute our opinion on the purpose and the quality of proofs. However, to the best of our knowledge there is an inherent trade-off between some of these criteria as argued subsequently for three cases.

— Isabelle as a Proof Assistant

So called *pen-and-paper proofs* are not trustworthy because they are prone to human error. Moreover, these proofs are often not given in a predefined language resulting in poor maintainability. However, by leaving out many details they are often easily accessible to a certain audience by having a level of detail that fits the expertise of the audience. Finally, building upon such untrustworthy proofs can only result in untrustworthy results down the line.

Formal proof-calculi have been introduced by Hilbert and Gentzen such as natural deduction systems and the sequent calculus. In such calculi a proof of a property S_n is given by a finite sequence of proof steps between properties such as in $True \vdash S_1 \vdash S_2 \vdash \dots \vdash S_n$. Backward proofs are constructed from the final property S_n backwards by applying proof rules such as the modus ponens $P \longrightarrow Q \implies P \implies Q$. However, many proof rules such as the modus ponens have multiple assumptions and, hence, the sequence of proof steps is better represented by a traversal of a proof-tree where the root is the property to be verified, the inner nodes are intermediate properties, and the leaf nodes are assumptions. See Example 6.1|p.95 for an example of such a proof-tree. The linear representation obtained from a traversal of this proof-tree then contains proof states S_k of the finite shape $\bigwedge_i (\phi_{i_0} \implies \dots \implies \phi_{i_j} \implies \phi_i)$, which is a conjunction of a finite number of consequences where each consequence may have a finite number of assumptions ϕ_{i_k} . Then, a single proof step replaces one of these consequences, which may be considered to be pending proof goals, by a finite possibly empty set of consequences. A complete proof then ends with an empty list of pending goals. *Pen-and-paper* proofs given in these proof calculi are not trustworthy, but, by being stated in a formal language, they allow for a *team-based joint inspection* analyzing their correctness in terms of fault- and omission-freeness. A drawback of proofs given in these proof calculi is that the level of detail is fixed and maximal and is henceforth not suitably accessible to typical audiences due to the resulting size of the proofs. Moreover, maintainability of such proofs is troublesome because most modifications to definitions and theorems require a modification of the proofs in many places. Since providing such a high level of detail also requires a tremendous amount of resources, these proof calculi are not widely used for verification purposes.

The ongoing development of the interactive theorem prover Isabelle [265] dates back to 1994. It implements a formal proof calculus as discussed in the previous paragraph. The goal of the Isabelle developers is to remedy the problems of proof-calculi that hinder their application by providing adequate tool-support. Isabelle comes along with a syntax, that is, a formal language, for definitions, theorems, and proofs. Definitions are obviously primarily given by terms, which are close to the mathematical functional notation and the typed λ -calculus using a syntax following the functional programming language ML. Theorems are based on higher order logic (HOL) and on expressions as employed in definitions.

Isabelle has automatic support in the sense that theories (comprising definitions, theorems, and proofs) are checked by Isabelle for syntactical correctness, type-correctness (that is, operations are never applied to arguments of a wrong type or arity), and, for the proofs, also for correctness in the sense of fault-freeness and omission-freeness as discussed above. Note, the statements that can be expressed in the input-language are too complex to allow for automatic derivation of proofs in general. However, attached SAT (Satisfiability for Propositional Logic) and SMT (Satisfiability Modulo Theories) solvers such as *e* [312], *spass* [171], and

vampire [353] as well as various built-in heuristic proof strategies such as **auto**, **force**, **clarify**, **simp**, and **blast** can be issued to obtain proof fragments that are then checked by Isabelle in the usual way.

Internally, Isabelle translates every provided proof step into a low-level sequence of proof steps that are then checked by the Isabelle kernel. The *trustworthiness* of Isabelle-checked proofs is a commonly accepted fact and, hence, using Isabelle for proof-checking ensures the highest confidence in proofs currently available. Due to the assumed trustworthiness it is, in our opinion, neither necessary nor advised to inspect Isabelle proofs to determine their correctness. Moreover, the usage of SAT and SMT solvers and heuristic proof strategies reduces the granularity of proofs, which typically increases readability of proofs and reduces the resources required for entering proofs. However, it hampers the possibility for step-wise analysis of the proofs because multiple proof steps are executed internally by a single provided command.

An alternative input language for proofs called *isar* has been released with the aim of increasing the *accessibility* and the *adaptability* of proofs. However, in our opinion, this goal has not been achieved because the overwhelming amount of proof-code of many theories (like the ones produced for this thesis) principally precludes human-based inspection. To increase accessibility and adaptability, we propose the usage of horizontal and vertical decomposition techniques as follows. Firstly, theory files should be split where appropriate into smaller theory files solving separate problems. Secondly, information hiding should be used to expose only a small set of definitions and theorems to other theory files. Thirdly, proofs should be split by introducing lemmas where appropriate to obtain smaller proof-blocks. Fourthly, theories should be inspected by considering definitions and theorems and the relationships among them as established by their proofs (not considering the proofs) in a first step. Fifthly, proofs should be inspected with the interactive support of Isabelle to display the intermediate lists of proof goals. However, as for big software products, additional documentation such as this thesis, may be required to describe the overall proof-strategy at the required different levels of detail to reasonably support accessibility of Isabelle theories. Moreover, while the proof-trees of the mentioned proof calculi nicely state the application of rules (by precisely stating the employed matching and the used assumptions), there is no suitable support in Isabelle yet for this matter. The adaptability of proof-code is also at a very early stage (basically, after adaptation of used definitions or theorems, Isabelle is able to localize points that require modifications by throwing proof-checking errors in these lines) and reasonable refactoring-techniques are called for. On the positive side, Isabelle allows for the usage of abstract theories that can be instantiated to prevent duplications of definitions, theorems, and proofs. Contradicting our own perception of how progress should be made in this matter, the key strategy of the *isar* language is to make the user enter as much data as possible to not only guide Isabelle to be able to check the proof but also to obtain a human readable proof-script: we believe, as mentioned above, that Isabelle proof scripts should not be inspected without

the support of Isabelle, which renders the additional annotations unnecessary as more relevant information can be derived automatically.

We conclude that we use the general purpose proof assistant Isabelle to ensure trustworthiness of our formal definitions, theorems, and proofs to exclude fundamental problems from our concrete controller synthesis algorithm.

Consider the following example where we have given a very simple lemma (which can be automatically solved by any of the built-in heuristic tactics) with its proof in two different notations.

Example 6.1: «Formal Proofs and Isabelle»

Examples of Proof Rules: The proof rules *mp* and *conjI* are given by listing their premises above the horizontal line and the conclusion below the horizontal line.

$$\text{mp} \frac{P \longrightarrow Q \quad P}{Q} \quad \text{conjI} \frac{P \quad Q}{P \wedge Q}$$

Example of a Proof: The two given proof rules *mp* and *conjI* are applied in the following proof. The lemma verified by the proof states that the three top-most properties in the proof-tree $p \wedge q \longrightarrow r$, p , and q jointly imply the property r . In the proof-tree, each proof step corresponds to one horizontal line where the applied rule is given on the left side and where the instantiation of the proof rule is given on the right side.

$$\text{mp} \frac{p \wedge q \longrightarrow r \quad \text{conjI} \frac{p \quad q}{p \wedge q} (P = p \text{ AND } Q = q)}{r} (P = p \wedge q \text{ AND } Q = r)$$

Example of a Proof in Isabelle-Syntax: The same proof from above is entered in Isabelle in the following syntax showing that the same two rules are applied with the same variable substitutions as before. Also, the three leaf nodes of the proof-tree are handled by stating that they occur among the premises of the lemma by using the command **assumption**.

```
lemma "p ∧ q ⟶ r ⟹ p ⟹ q ⟹ r"
apply (rule-tac P = "p ∧ q" and Q = "r" in mp)
apply (assumption)
apply (rule-tac P = "p" and Q = "q" in conjI)
apply (assumption)
apply (assumption)
done
```

Interactive Inspection of Proofs in Isabelle: As stated above, we recommend to inspect proofs by using the interactive mode of Isabelle to display the intermediate lists of pending proof goals. To give an idea of the benefits, we added in the following listing this additional information generated by Isabelle during the step-wise analysis. Note, commands are applied on the first pending goal **1**.

```

|lemma "p ∧ q → r ⇒ p ⇒ q ⇒ r"
  1 p ∧ q → r ⇒ p ⇒ q ⇒ r
|apply (rule-tac P = "p ∧ q" and Q = "r" in mp)
  1 p ∧ q → r ⇒ p ⇒ q ⇒ p ∧ q → r
  2 p ∧ q → r ⇒ p ⇒ q ⇒ p ∧ q
|apply (assumption)
  1 p ∧ q → r ⇒ p ⇒ q ⇒ p ∧ q
|apply (rule-tac P = "p" and Q = "q" in conjI)
  1 p ∧ q → r ⇒ p ⇒ q ⇒ p
  2 p ∧ q → r ⇒ p ⇒ q ⇒ q
|apply (assumption)
  1 p ∧ q → r ⇒ p ⇒ q ⇒ q
|apply (assumption)
done

```

6.2. ISABELLE-BASED FRAMEWORK OF DEFINITIONS AND PROPERTIES

In this section, we focus on the overall structure of our Isabelle-based formalization [303] and on parts reusable in similar contexts in the future.

Firstly, to extend the built-in support of Isabelle for the verification of partial recursive functions, we introduced a custom structured workflow, which is based on definitions and theorems and which is presented in greater detail in subsection 6.2.2|p.100.

Secondly, to support the compositional verification of functional algorithms, we have followed a workflow using interfaces and decomposition, which is discussed in subsection 6.2.1|p.99.

Thirdly, to formalize semantics for our concrete formalisms, we developed and interpreted custom abstract parameterized theories (based on locales in Isabelle), which encompass various general important definitions and theorems. This framework of semantics allows for the application of techniques that are well-known in the domain of computer science such as bisimulation. We provide details on these abstract parameterized theories and their interpretation for the concrete formalisms in subsection 6.2.3|p.102.

As additional contributions (discussed at an earlier stage in [310]), we formalized equivalent notions of controllability for languages and discrete event systems, introduced fixed-point iterators based on these notions as in subsection 4.2.4|p.52, and defined equivalent sets of desirable solutions as in section 4.3|p.53. However, these fixed point iterators have not been included in this thesis because we are not aware of reasonable implementations of them based on DPDA.

On the next pages, we present two figures on the structure and size of our formalization and on the size of proofs.

Figure 6.1: «Structure and Size of the Isabelle-based Formalization»

We present the basic structure of our Isabelle-based formalization consisting of four major and 14 minor parts with their number of proofs and lines of the proof scripts. The largest minor parts are: the locales for semantics and their relationship (minor part 6), the instantiation of these locales for EPDA, Parsers, and CFGs (minor parts 7–9), and the verification of our constructions for enforcing nonblockingness on a given DPDA (minor part 12). The other minor parts sum up 9.3 % of the lines of the overall proof script.

#	Content	Page	Proofs	%	Lines	%
<i>foundations</i>						
1	foundational and built-in operations		721	11.8	11 461	2.0
2	words and languages		224	3.6	4370	0.7
<i>abstract controller synthesis</i>						
3	discrete event systems	p. 17	0	0.0	265	0.0
4	abstract supervisory control problem	p. 33	164	2.6	5402	0.9
5	abstract controller synthesis algorithm	p. 45	206	3.3	10 135	1.8
<i>concrete formalisms</i>						
6	locales for transition systems		434	7.1	49 527	8.8
7	interpretation of locales for EPDA	p. 18	468	7.6	29 485	5.2
8	interpretation of locales for Parsers	p. 22	630	10.3	53 015	9.4
9	interpretation of locales for CFGs	p. 27	784	12.8	66 373	11.8
<i>concrete controller synthesis</i>						
10	construction of closed loop	p. 64	23	0.3	3204	0.5
11	concrete supervisory control problem	p. 36	16	0.2	781	0.1
12	enforcing nonblockingness	p. 65	2299	37.6	307 362	54.8
13	removing noncontrollability	p. 82	95	1.5	15 865	2.8
14	concrete controller synthesis algorithm	p. 85	40	0.6	3299	0.5
			Σ 6104		Σ 560 544	

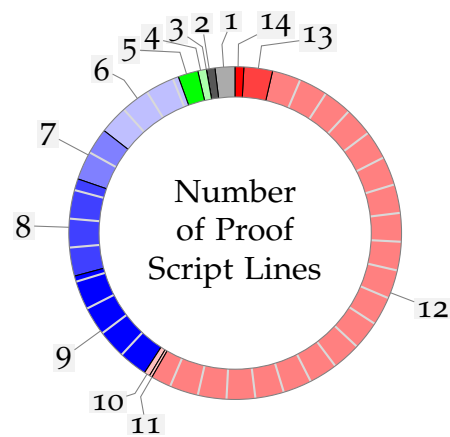
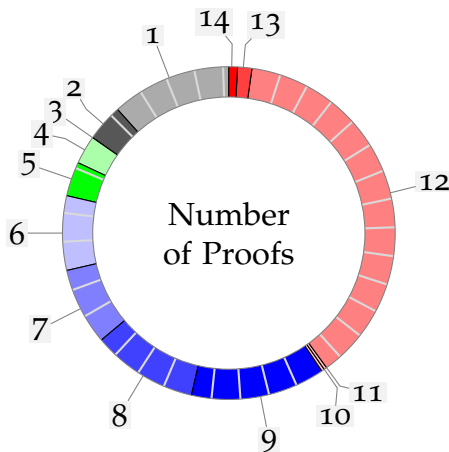
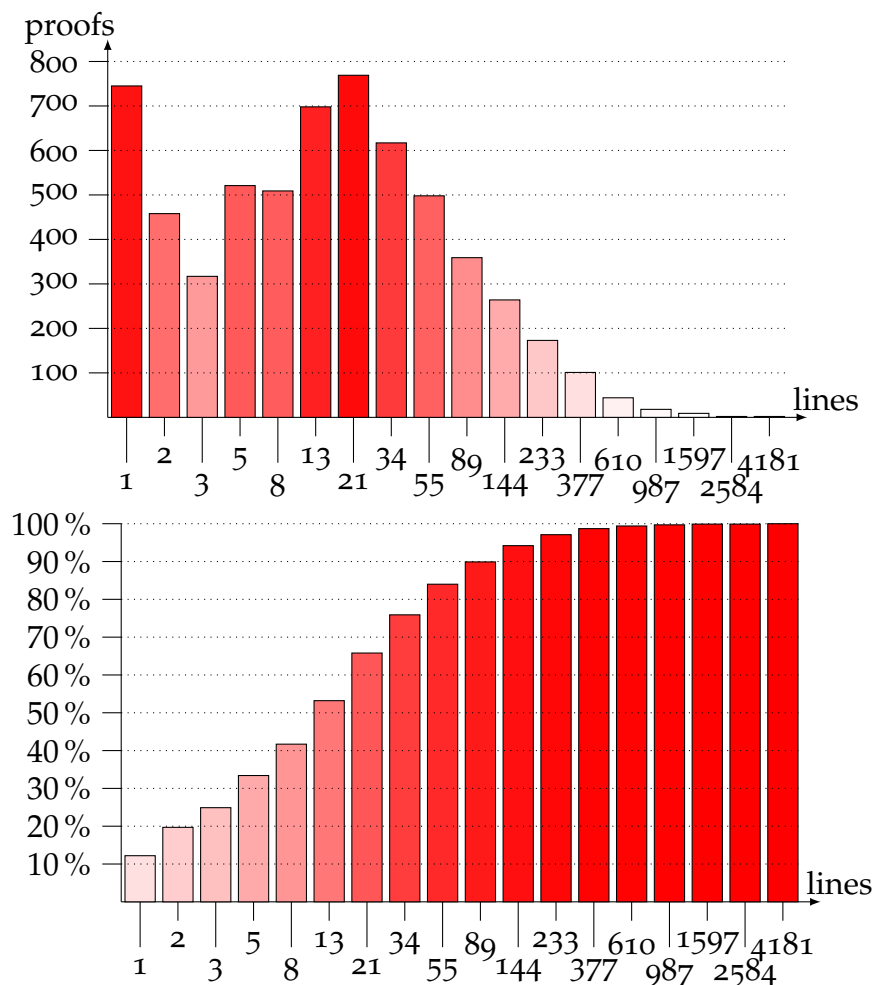


Figure 6.2: «Distribution of Proof Length of the Isabelle-based Formalization»

We analyze our Isabelle-based proofs by the number of lines that contain an application of the **apply()** command (where we disregard all instances of variable renamings). A fine-grained modularity of the relevant lemmas, which results from short proofs, is preferable for maintenance tasks such as refactoring and adaptation to changed definitions as well as for easing composition in future applications. The numbers (visualized below) indicate that our proofs are rather short with an average of 39.1 lines and a mean of 12 lines.

Major Part	Proof Length	
	Average	Mean
<i>foundations</i>	6.0	3
<i>abstract controller synthesis</i>	17.6	9
<i>concrete formalisms</i>	34.0	13
<i>concrete controller synthesis</i>	59.7	17



6.2.1. Verification of Composed Concrete Building Blocks

The concrete controller synthesis algorithm from chapter 5|p.61 is defined by composition of various building blocks. For most operations, we defined input/output-specifications to determine clean interfaces to ease building block composition. Such an input/output specification is given for a building block $f :: \alpha \Rightarrow \gamma$ by a definition of the legal inputs $SPi-f :: \alpha \Rightarrow bool$ and by a definition of the expected relationship between inputs and outputs $SPo-f :: \alpha \Rightarrow \gamma \Rightarrow bool$.

To discuss the verification of composed building blocks, we assume that f is given as the composition of two building blocks $g :: \alpha \Rightarrow \beta$ and $h :: \beta \Rightarrow \gamma$. However, note that our proposed solution also translates to more than just two building blocks g and h and also to building blocks with more than one argument. Moreover, we assume that g and h have been verified w.r.t. their input/output specifications in a previous step, that is, we assume the satisfaction of

$$SPi-g X \implies SPo-g X (g X) \quad \text{and} \quad (6.1)$$

$$SPi-h X \implies SPo-h X (h X). \quad (6.2)$$

That said, the satisfaction of the input/output specification of f is now decoupled from the verification of g and h and can be realized using the following theorem.

Theorem 6.1: «Verification of Sequential Execution»

$$\begin{aligned} & \textcolor{red}{1} \quad SPi-f X \\ \implies & \textcolor{red}{2} \quad SPi-f X \longrightarrow SPi-g X \\ \implies & \textcolor{red}{3} \quad SPi-g X \longrightarrow SPo-g X (g X) \\ \implies & \textcolor{red}{4} \quad SPo-g X (g X) \longrightarrow SPi-h (g X) \\ \implies & \textcolor{red}{5} \quad \forall A B C. SPo-g A B \longrightarrow SPo-h B C \longrightarrow SPo-f A C \\ \implies & \textcolor{red}{6} \quad \forall A. SPi-h A \longrightarrow SPo-h A (h A) \\ \implies & \textcolor{red}{7} \quad SPo-f X (h (g X)) \end{aligned}$$

Explanation: $\textcolor{red}{1}$ states that X is an expected input for the function f and the conclusion $\textcolor{red}{7}$ states that the composition of g with h (that is, the function f) is sound w.r.t. the input/output specification of f . $\textcolor{red}{2}$ states that the input specification of g accepts any input that is valid for f . $\textcolor{red}{3}$ states that the operation g satisfies its input/output specification for the given X (cf. (Eq. 6.1)|p.99). $\textcolor{red}{4}$ states that the value $g X$ obtained by applying g in the first step is an admissible input for the operation h to be executed subsequently. $\textcolor{red}{5}$ states that the composition of the output specifications of g and h results in the output specification of f , a requirement that guides the definition of the output specification of f for applications of this theorem. Finally, $\textcolor{red}{6}$ states that h satisfies its input/output specification (cf. (Eq. 6.2)|p.99), which is in particular relevant for the value $g X$.

While the trivial proof of this theorem requires only a single line in Isabelle, we included it here for its practical relevance for the verification of the concrete controller synthesis algorithm where we applied it 46 times (also counting similar formulations for handling multi-argument building blocks as well).

6.2.2. Verification of Partial Recursive Functions in Isabelle

In this thesis, we make use of various concrete building blocks that obtain fixed points by executing recursive functions of the form:

$$f :: \alpha \Rightarrow \alpha \quad (6.3)$$

$$f \ x = \text{if Guard } x \text{ then } f \ (\text{Call } x) \text{ else } x \quad (6.4)$$

using the two functions

$$\text{Guard} :: \alpha \Rightarrow \text{bool} \quad \text{and} \quad \text{Call} :: \alpha \Rightarrow \alpha. \quad (6.5)$$

Subsequently, we are only concerned with functions f of this kind that terminate for a *relevant* part of the domain elements but *not for all* domain elements because (a) there is sufficient support for functions terminating on all domain elements and (b) we often exhibit such partial functions in our concrete controller synthesis algorithm from chapter 5|p.61.

For example, the concrete building block for enforcing accessibility for CFG entails (see Definition 5.4|p.66) the construction of the greatest set of productions of the given CFG that occur in initial derivations of the CFG. The domain of this operation contains pairs (G, A) where G is of the type of CFG and where A is a set of productions. However, the type does not impose sufficient restrictions on these pairs. Firstly, G is not required to be a valid CFG with a finite set of productions because this restriction is not part of the type but is imposed on top of it by an additional definition of valid CFGs. Secondly, the set A may be infinite and is also not required to be contained in the productions of G .

For the general case of a partial recursive function f , we identify a well-formed subset of the domain by defining a function:

$$\text{WF-Inputs} :: \alpha \Rightarrow \text{bool}. \quad (6.6)$$

This function returns *True* for the domain elements to which f may be applied. Hence, it also describes an invariant to be satisfied throughout the recursive computation. For the example, we require for a pair (G, A) that G is a valid CFG with a finite set of productions and that A is a subset of these productions.

For the example, termination follows from the fact that the size of the set of productions of G that are not already contained in A strictly decreases between any two adjacent recursive calls. That is, if an argument (G, A) is modified into (G, A') for the next recursive call using the function *Call* from above, we verify that $\text{card}(\text{cfg-productions } G - A)$ is strictly greater than $\text{card}(\text{cfg-productions } G - A')$.

For the general case, we state this termination argument by defining a function

$$\text{Measure} :: \alpha \Rightarrow \text{nat}. \quad (6.7)$$

This function maps each domain element to some natural number such that well-formed arguments x_i to which the function f is applied during a computation

have strictly decreasing values *Measure* x_i . As expected, termination is guaranteed for well-formed arguments because the number of further applications of f is bounded by *Measure* x_i for any given well-formed argument x_i .

We introduced the following theorem to verify properties such as termination of f on well-formed domain elements but also for more advanced properties on the actual return values of f . The theorem lists as premises certain minimal requirements that are needed to be satisfied to derive for a given domain element y that the property P is satisfied for it. The theorem also requires for its instantiation concrete definitions for *Call*, *Guard*, *WF-Inputs*, and *Measure* where the first two are given by the definition of f and where the latter two are to be derived such as in our example-based explanations above.

Theorem 6.2: «Partial Termination»

- 1 $\forall x. \text{WF-Inputs } x \longrightarrow \text{Guard } x \longrightarrow \text{WF-Inputs } (\text{Call } x)$
- \implies 2 $\forall x. \text{WF-Inputs } x \longrightarrow \text{Guard } x \longrightarrow \text{Measure } x > \text{Measure } (\text{Call } x)$
- \implies 3 $\forall x. \text{WF-Inputs } x \longrightarrow \neg \text{Guard } x \longrightarrow P \ x$
- \implies 4 $\forall x. \text{WF-Inputs } x \longrightarrow \text{WF-Inputs } (\text{Call } x) \longrightarrow P \ (\text{Call } x) \longrightarrow P \ x$
- \implies 5 $\text{WF-Inputs } y$
- \implies 6 $P \ y$

Explanation: 1 states that all applications of f in recursive calls are made to well-formed inputs, that is, the well-formedness of the initial input is preserved throughout the computation. 2 states that the arguments to recursive calls decrease during any computation w.r.t. the function *Measure*. 3 states that the property P to be verified is satisfied upon termination. 4 states that the satisfaction of property P can be translated backwards to arguments of earlier recursive calls. 5 states that y is a well-formed input. Finally, the conclusion 6 states that y satisfies the given property P .

For the concrete example at hand for enforcing accessibility for CFG, we may state as a property P that the recursive fixed-point computation terminates and that the obtained result is the desired maximal set of productions that occur in initial derivations.

We verified this simple theorem by complete induction on the *Measure*-values and applied it successfully for the verification of nine different concrete building blocks of our concrete controller synthesis algorithm. Due to our experiences with this theorem, we believe that it may be beneficial to use it also in future applications of our Isabelle-based framework and also, independently, in other scenarios with partial recursive functions.

For a comparison, Isabelle introduces for the partial function f from above for the purpose of handling termination only the following rule by default (where the predicate f_dom states that f terminates on the argument).

$$f.\text{domintros}: (\text{Guard } x \implies f_dom \ (\text{Call } x)) \implies f_dom \ x \quad (6.8)$$

However, this rule $f.\text{domintros}$ proved insufficient for our purposes.

6.2.3. Hierarchical Framework of Abstract Theories for Defining Semantics

We introduce a framework in which semantics of a broad range of formalisms that describe discrete event systems can be defined. This framework allows for the *concise* definition and *uniform* handling of semantics, which is of paramount importance in the context of the Isabelle-based verification of our concrete controller synthesis algorithm. In this thesis, we apply this framework to the formalisms of EPDA, Parsers, and CFGs to obtain various semantics for them (confer chapter 2|p.15, subsection 6.2.6|p.112, and subsection 6.2.7|p.114). Also confer to subsection 8.1.8|p.186 for related Isabelle-based frameworks.

— *Abstraction and Concretion in Hierarchical Frameworks*

Abstraction and concretion are important elements in the domain of software engineering and mathematics as can be seen from the following two examples. Firstly, the programming language Java offers support for abstraction and concretion in the sense of abstract Java classes with generic types containing abstract and concrete methods with input/output types ranging also over these generic types. Concretions of such abstract Java classes then determine consistent substitutions of the abstract types and abstract methods and inherit all concrete methods of the abstract class as well. Secondly, algebraic structures are often introduced by making use of abstraction and concretion. Abstract algebraic structures such as rings and complete lattices are defined by listing abstract types, abstract functions ranging over these abstract types, and assumptions to be satisfied by these abstract functions. Also, in such an abstract algebraic setting, based on these functions and assumptions, further notions and properties can be defined and verified, respectively. Concrete algebraic structures are then determined by consistent substitutions of the abstract types and functions such that the assumptions of the abstract algebraic structure are satisfied. With these two examples in mind, we move on with the framework's introduction.

We define our framework of semantics in the form of a hierarchy of parameterized theories where each parameterized abstract theory is formalized in Isabelle using the locale-technology (type classes, which are special locales, are used in the Isabelle distribution for the definition of algebraic structures such as rings and complete lattices mentioned above). A single locale describes a single parameterized abstract theory as follows. The head of a locale is given by a list of abstract type parameters, abstract functions, and assumptions on these abstract functions. Moreover, the head *may* additionally contain a list of locales that are to be extended together with their function parameters to obtain a hierarchy of locales. The body of a locale consists of definitions and theorems where these abstract types and functions can be used. In general, definitions not using any of the abstract types or abstract functions are usually defined before introducing the locale. Finally, a locale is interpreted by providing for each abstract type and each abstract function a concrete function. More specifically, concrete theories are obtained from our given hierarchy of locales by interpretation of all locales contained in a selected subtree that includes the root node of the hierarchy. Locales that are not selected may, for example, contain assumptions that are not satisfied

by the concrete theory to be obtained. Note, many nodes of our hierarchy contain theorems and definition that are inherited upon concretization.

An abstract framework allows for a reduced number of definitions and proofs because common *derived notions* (for example, derivations and languages in our case) and *derived properties* (for example, compositionality of derivations and inclusion of the marked language in the unmarked language in our case) are defined and verified only once in the abstract domain and are inherited to the concrete formalisms automatically. Moreover, *uniform* definitions using the Isabelle built-in locale mechanism for our abstract parameterized theories allow for intuitive interformalism considerations, for example, if instances from one formalism are to be translated into instances of another formalism while preserving or reflecting certain properties (see subsection 6.2.4|p.109).

A *hierarchical* framework is called for because there is an inherent trade-off between abstractness and usefulness as follows. On the one hand, an abstract parameterized theory should be as *abstract* as possible by restricting the amount of knowledge it *extracts* from the concrete semantics through the listed abstract functions and assumptions. On the other hand, an abstract parameterized theory should be as *useful* as possible by containing as many useful definitions and theorems as possible such that, upon instantiation of this theory, a broad useful spectrum of concrete definitions and theorems is available in the concrete setting. However, strong assumptions are required for the derivation of such useful theorems. Given this trade-off, we argue that a single abstract theory is not desirable and, instead, we propose to cope with this trade-off, similarly to abstraction and concretion in Java and the corresponding handling of algebraic structures above, by introducing, based on an abstract core theory, a tree shaped hierarchy of abstract parameterized theories that gradually incorporate different characteristics from the concrete settings.

To sum up, we employ a hierarchical framework for the definition of the semantics of EPDA, Parsers, and CFGs. We construct this framework of locales by gradually capturing aspects of the various semantics of the formalisms of EPDA, Parsers, and CFGs. The additional locales are introduced as direct or indirect extensions of a core locale by listing further function parameters and assumptions thereon. Concrete semantics are then obtained as concretizations of the locales given by a subtree of this hierarchy that includes the core locale.

We continue with the introduction of the abstract core theory, which is the root node of our hierarchical framework, and then introduce our extensions of this core locale, which we group for presentation purposes in three categories. Firstly, we define the marked and unmarked language of a structure in Par. *Abstract Theories for Marked and Unmarked Languages*|p.104. Secondly, we introduce locales to cover certain kinds of components occurring in configurations such as scheduler and history variables resulting in the classification of branching and linear semantics in Par. *Branching and Linear Semantics*|p.105. Thirdly, we introduce locales for the abstract definition of further operational properties such as accessibility, nonblockingness, determinism, and controllability in Par. *Abstract Theories for Operational Properties*|p.108.

→ *The Abstract Core Theory for Semantics*

The core locale of our framework states only the most fundamental requirements. It has the abstract type parameters *structures*, *configurations*, and *labels*. Moreover, it uses the abstract function parameters *WF-structures*, *WF-configurations*, and *WF-labels* for identifying well-formed elements of these three type parameters. The function parameter *step-relation* maps a structure, a source configuration, a used label, and a target-configuration to a boolean value. As a first assumption, we state that the target configurations accessed by application of the step-relation are always well-formed whenever the structure, the source configuration, and the label are well-formed. Finally, the function parameter *initial-configurations* identifies configurations where derivations are to be started. As a second assumption, we state that the initial-configurations are well-formed configurations.

In this core locale, we define, for example, derivations (using the step relation) and initial derivations (using the initial configurations) and verify results on their composition and decomposition.

For *structures* and *labels*, we use in the interpretations for EPDA, Parsers, and CFGs the corresponding record types as discussed in chapter 2|p.15. However, while the configurations and initial configurations are of the same pattern for all semantics of CFGs, we define custom configurations and initial configurations based on custom record types for each semantics of EPDA and Parsers. Moreover, we provide for each semantics of EPDA, Parsers, and CFGs a custom step relation.

→ *Abstract Theories for Marked and Unmarked Languages*

For the abstract definition of the marked and unmarked language of a structure, we extend the core locale from above with the additional abstract type parameter *effects*. This type parameter is interpreted for our concrete formalisms by lists of events. Moreover, we introduce additional abstract function parameters. Firstly, the function *WF-effects* describes for a structure the elements of the type effects that may occur. For EPDA, Parsers, and CFGs these well-formed effects are the words ranging over the event alphabet contained the respective structure. Secondly, the function *marking-condition* is a predicate defining the initial derivations that mark a word. Confer to our explanations in chapter 2|p.15 where we explained the marking condition already for the semantics described there and also see our explanations on marked and unmarked languages in the subsequent paragraph. Finally, the two operations *unmarked-effect* and *marked-effect* define for each initial derivation a set of unmarked effects that are executed in that derivation and a set of marked effects that are marked in that initial derivation. In the locale given by these extensions, we define the marked language and the unmarked language of a given structure by gathering the marked and unmarked effects from the initial derivations, respectively.

The interpretation of these parameters depends for concrete semantics obviously on the respective employed types of configurations. In further locales, we specialize the type of effects to words and consider only finite initial derivations for the definition of the two sets of languages.

—• *Branching and Linear Semantics*

As a major achievement, we introduced locales to distinguish between branching and linear semantics. The usage of both kinds of semantics resulted in great benefits throughout the verification of our concrete controller synthesis algorithm.

In branching semantics, the applicability of steps is *solely* restricted by the structure whereas in linear semantics the applicability of steps is also governed by a schedule. Such a schedule typically is a word of events that are to be executed in that order. It is contained as a component in the configurations and guides the future evolution because steps that conflict with the schedule are disabled. In the linear and branching semantics of Parsers, we also encounter fixed schedulers (see section 2.3|p.22) and unfixed schedulers as components of configurations as a more explicit decomposition of scheduler variables. Linear semantics then have a scheduler variable or an unfixed scheduler variable and branching semantics (of Parsers with a look-ahead of at least one) may also fix events on the fly by storing them in a fixed scheduler variable. Also, various semantics make use of history variables that remember events executed in previous steps of the current derivation.

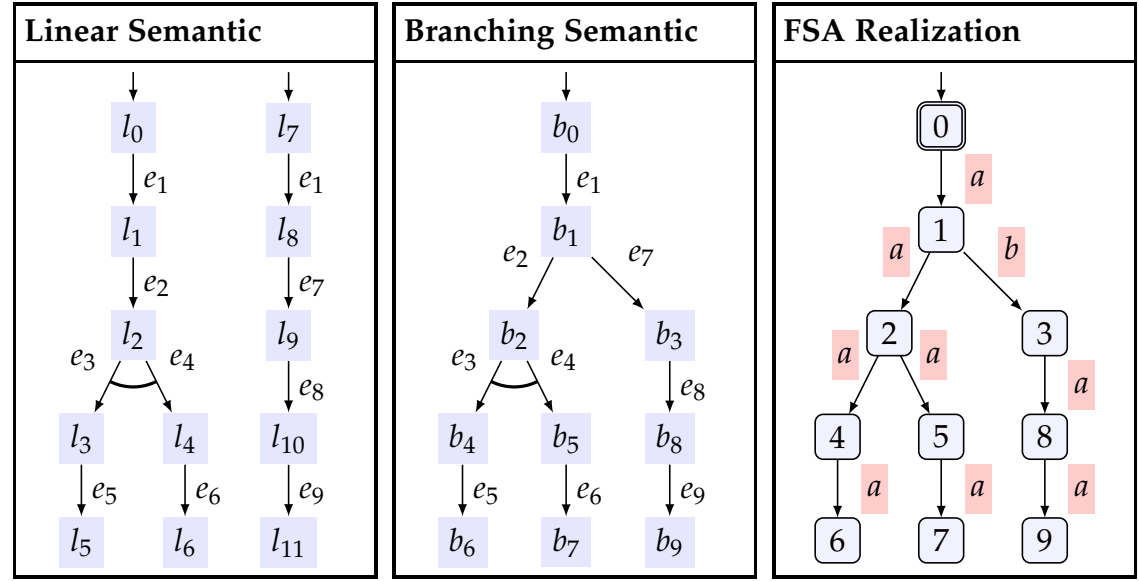
The most common semantics for EPDA and Parsers in the literature are the linear semantics discussed in subsection 6.2.6|p.112 and subsection 6.2.7|p.114. For many other formalisms such as CFG and Petri net it is not common to have a schedule guiding the upcoming behavior.

In this thesis, we introduce linear and branching semantics for EPDA and Parsers and relate them suitably to allow for their interchangeable usage (see subsection 6.2.4|p.109). Since branching and linear semantics have, as explained later on, specific definitions of key notions such as determinism, it is important to ensure that the different definitions coincide among the semantics of a formalism. Moreover, depending on the variables contained in the configurations, it is also possible that multiple definitions of some notion are inherited from different abstract locales upon concretization; in these cases, we also verify the correspondence between these definitions.

As visualized in Figure 6.3|p.106 the two kinds of semantics result in different shapes of accessibility graphs. In the linear semantics of EPDA and Parsers presented in this thesis, there is a unique initial configuration for each word over the event alphabet and for each of these initial configurations there is (up to nondeterminism and maximality) a unique derivation starting in this initial configuration. In contrast to this, the branching semantics presented in this thesis have a unique initial configuration, each derivation starts in this initial configuration, and the accessibility graph is of no degenerate form in general because often multiple distinct effects may be executed depending on the EPDA, Parser, or CFG at hand. If the executed events are recorded in a history variable and each step appends a non-empty word to the history variable, the accessibility graphs are of the form of trees. If no such history variable is used, as for example in Petri net, the accessibility graph may contain loops.

Figure 6.3: «Linear versus Branching Semantics for FSA»

Due to the existence (nonexistence) of a scheduler variable in configurations of linear (branching) semantics, the accessibility graphs form a forest (tree) for linear (branching) semantics. Incompatible steps (here e_2 and e_7 executing the distinct events a and b) occur in different trees in linear but may occur from the same node in branching semantics. Compatible steps (here e_3 and e_4 both executing the event a) violating determinism result in branching of the trees in linear semantics. Note, in both kinds of semantics the same sequences of edges are applicable: $[e_1, e_2, e_3, e_5]$, $[e_1, e_2, e_4, e_6]$, and $[e_1, e_7, e_8, e_9]$.



Semantics where no (unfixed) scheduler variable is contained in the configuration (that is, branching semantics) have the advantage that properties such as operational nonblockingness can be defined easily by stating that for every initial derivation there is some continuing derivation that can be appended to result in a marking derivation. For a semantics where an (unfixed) scheduler variable is contained in the configuration (that is, in a linear semantics) the definition is more complicated because for the (unfixed) schedule remaining at the end of the given initial derivation there may be no such continuing derivation, and, to handle that problem, each configuration of the given initial derivation would have to be adapted such that the remaining (unfixed) schedule is then chosen such that the required continuation exists. Hence, branching semantics are, as demonstrated by the example, more suitable when derivations are to be appended to a given derivation.

Note, a similar problem occurs when a derivation is to be prepended to a given derivation in a semantics where history variables are used in the configurations, which is the case for our branching semantics for EPDA and Parsers. In this case the history variables contained in all configurations of the given derivation may need to be adjusted before the composition.

The definition of the marked and unmarked language also depends on the variables available in the configurations. In a semantics where configurations have no history and no scheduler variables, we define *unmarked-effect* and *marked-effect* by replaying the impact of the labels contained in the derivation on the initial configuration. Such a definition is quite complex and can be simplified when scheduler variables are contained in the configurations. Using these variables, we define *unmarked-effect* and *marked-effect* by considering configurations one by one and by relying on the initial value of the scheduler variable. However, the most simple definition requires history variables in the configurations. Using these variables, we obtain *unmarked-effect* and *marked-effect* directly by considering configurations one by one. We prefer semantics with history variables because the complexity of such central definitions has a big impact on many subsequent proofs and should be, for that reason, as simple as possible.

When having multiple semantics for a given formalism, we require the semantics to coincide in this thesis. For the *marking-condition* this means that initial derivations in different semantics applying the same sequence of labels should coincide w.r.t. their corresponding marking conditions. We argue that this requirement may lead to more complex definitions of the marking condition for branching semantics compared to the equivalent linear semantics. For example, consider the branching semantics *epdaH* and *parserHF* for EPDA and Parsers where we check for marking configurations (see chapter 2|p.15) to determine the satisfaction of the marking condition. For these branching semantics the satisfaction of the marking condition is not preserved by step application as we must check that no events are executed after the detected marking configuration has been accessed. As an example supporting this claim consider the FSA from Figure 6.3|p.106 where we expect that only the empty word is a marked word. Since only the empty initial derivation satisfies the marking condition in the linear semantics (every longer initial derivation in that semantics has a different non-empty initial schedule), precisely the empty initial derivation of the branching derivation must satisfy the marking condition. Consequently, the execution of further events in all non-empty initial derivations of the branching semantics must invalidate the satisfaction of the marking condition.

The fixed scheduler variable, which may occur in configurations of branching and linear semantics, is the part of the scheduler that has been executed in the initial derivation at hand but that has not been removed from the scheduler yet. As steps of the initial derivation depended on this fixed scheduler any modification of it may break the initial derivation. Conversely, the unfixed scheduler variable is the part of the scheduler that has not been observed in the initial derivation up to the configuration at hand. Hence, the unfixed scheduler may be modified or exchanged entirely with the following restriction. Parsers that detected/determined the special processing terminator may not execute further events and, hence, the unfixed schedule contained in configurations of a Parser may only be adapted if the processing terminator has not been fixed in a prior step. We call unfixed schedulers extendable if they may be adapted.

→ *Abstract Theories for Operational Properties*

We define accessibility, nonblockingness, determinism, and controllability by introducing locales as extensions of the locales for linear and branching semantics, which feature scheduler and history variables as introduced before.

For (operational) accessibility, we require in our locales type parameters and function parameters to identify elements, called *destinations*, to be accessible and function parameters for determining which of these elements are contained in a given initial derivation at a given point. Then, operational accessibility means that all elements that are to be accessible occur in some initial derivation.

Recall that operational nonblockingness means that every initial derivation must be extendable to an initial marking derivation (cf. Definition C.2|p.234 for a concrete definition of nonblockingness for EPDA by the use of the *epdaH* semantics) with the additional complication of schedule adaption mentioned in the previous paragraph. We introduce equivalent definitions for nonblockingness based on the availability of the scheduler variables in the configurations of branching as well as linear semantics. In general, the various notions of operational nonblockingness should coincide with the language based nonblockingness from Definition 3.2|p.33. However, Parsers may detect/determine the special processing terminator without executing an event and, to obtain the desired equivalence between both notions, we define *weak* operational nonblockingness by excluding initial derivations where the processing terminator has been detected/determined. Note, steps detecting/determining the processing terminator may bring the Parser from a marking configuration into a deadlock configuration that is not marking. In this case, the language based nonblockingness property is satisfied, but the operational nonblockingness property from above is not satisfied. This example shows that the language based nonblockingness property is not entirely appropriate for formalisms such as Parsers that may detect/determine the processing terminator silently as discussed already in [310].

Determinism for linear semantics means that two steps exiting an accessible configuration must coincide and determinism for branching semantics means that two steps exiting an accessible configuration must coincide if the word w_1 of events executed in the first step is a prefix of the word w_2 of events executed by the second step. We verified that the definitions for the branching and linear semantics (with scheduler or history variables) coincide in all cases. Note, the notion of determinism was already covered in chapter 2|p.15 for EPDA and Parsers for the branching semantics *epdaH* and *parserHF*. In this thesis, we only require operational determinism and do not employ syntactical determinism, which would require for EPDA that two edges exiting the same state are identical if they have compatible *edge-pop* and *edge-event* components. Such a syntactical definition proved more complex in proofs as also inaccessible well-formed configurations need to be considered where required invariants are then not available. Moreover, such syntactical definitions cannot be stated concisely at the abstract level of locales. Again, we provided a more general definition of determinism for branching semantics of Parsers where unfixed schedulers

occurring in configurations may be unextendable. Intuitively, if w_1 is a strict prefix of w_2 above and the special processing terminator is detected/determined in the first step, then both steps do not need to be identical for determinism.

Based on the locale based foundation for branching semantics, we define operational controllability in the context of an abstract parameterized theory involving two abstract semantics. These two semantics are interpreted by the branching semantics *epdaH* because the formalisms of DFA and DPDA used for the plant and the controller in our concrete controller synthesis algorithm are both subformalisms of the formalism of EPDA. The obtained definition is equivalent to Definition C.1|p.233 and the language based controllability definition from Definition 3.1|p.33.

6.2.4. Hierarchical Framework of Abstract Theories for Relating Semantics

In our Isabelle-based formalization, we state theorems and lemmas in which instances and semantics of one, two, or more formalisms are used. Adequate support is called for to allow for the derivation of simple and concise proofs of these properties.

→ *Categorization of Properties to be Verified*

Firstly, we state *intra-semantics intra-formalism properties* using a single semantics of a single formalism. For example, we show for a single EPDA that we obtain again an EPDA with the same marked language when applying a certain operation on it (two EPDA and typically one semantics). This default scenario is covered by our introduction of our hierarchy of locales from the previous subsections.

Secondly, we state *inter-semantics intra-formalism properties* using at least two semantics of a single formalism. For example, we show that the marked language of an EPDA is equally defined using any of our branching and linear EPDA semantics (one EPDA and at least two semantics). Also note, the semantics used in this example may be both linear semantics, both branching semantics, or a combination of the two kinds. For any selection of semantics of a single formalisms, we expect that the operational properties indeed coincide as explained before. However, branching and linear semantics of a common formalism have few similarities when it comes to locale interpretation: since configurations are adapted, every parameter that depends directly or indirectly on configurations is affected. From the locales presented so far, we typically only expect that the interpretations agree on *structure*, *step-labels*, *effects*, and *destinations*. We support this scenario as a special case of the following (last) scenario.

Thirdly, we state *inter-semantics inter-formalism properties* using at least two semantics of at least two formalisms. For example, we convert a Parser into an EPDA with equal marked language (one Parser, one EPDA, and two semantics). In general, we are primarily interested (as for the previous case of inter-semantics intra-formalism properties) in the marked and unmarked languages, nonblockingness, accessibility, determinism, and controllability.

→ *Locale-based Approach to Relating Semantics*

We now discuss our support of these kinds of properties by introducing various parameterized abstract theories, called *translational theories*, by using once more the locale technology of Isabelle. These translational theories are defined as locales that extend more basic locales and that have type parameters, function parameters, and assumptions on these parameters as before. Firstly, recall that we define concrete semantics as interpretations of a selection of a subtree of our locale hierarchy including the core locale. In fact, this selection also defines an abstract semantics in the form of a locale where the leafs of this subtree are the dependencies. The locale dependencies have to be provided with their function parameters and types and, of course, different leafs overlap in their function parameters as they extend more basic locales on a path to the core locale (at least they agree on the types and function parameters of the core locale). Secondly, the definition of a translational theory is a slightly more general case where two such selections, which correspond to two abstract semantics, are listed as dependencies and where the abstract types and function parameters occurring may overlap also between these two lists of locales. Thirdly, as for other locales, we may add further elements to such translational theories such as abstract types, function parameters, and assumptions to require suitable characteristics and to impose the satisfaction of additional constraints on the parameters, which may span over the two semantics simultaneously.

Our translational theories differ in the abstract semantics (that is, in the selections resulting the locale dependencies), in the overlapping between the two semantics (depending on the characteristics of the semantics and formalisms to be related), in the additional elements added to the locale's head, and in the semantical properties translated in the locales (each property may require a different set of knowledge for its translation).

Abstract results obtained within translational theories are derived in their concrete form for two given concrete semantics by interpretation of these locales. For such an interpretation of a translational theory, we have to provide, as before, consistent substitutions for the types and function parameters occurring in the locale's definition and verify the (additional) assumptions.

Note, relationships between two concrete semantics can only be established in this approach when the abstract locales are strong enough in the sense of capturing the characteristics required to derive the envisioned relationship. Definitions only present in the concrete setting but not in the abstract setting cannot be used within a translational locale as introduced here.

→ *Techniques Employed for the Translation of Operational Properties*

For two given abstract instances G_1 and G_2 with their corresponding abstract semantics, we employ weak bisimulations on initial derivations to relate G_1 and G_2 as follows. A weak bisimulations on initial derivations R is a relation containing pairs (d_1, d_2) where d_1 and d_2 are of the type of derivations of the first and second abstract semantics, respectively. Moreover, to make this relation R a weak bisimulations on initial derivations, we require (*well-formedness*) that for

every $(d_1, d_2) \in R$ both, d_1 and d_2 are initial derivation of G_1 and G_2 , respectively, (*completeness*) that for each zero length initial derivation d_1 of G_1 there is some initial derivation d_2 of G_2 such that $(d_1, d_2) \in R$ (and vice versa), and (*soundness*) that for every $(d_1, d_2) \in R$ and every single-step derivation d'_1 that may be appended to d_1 resulting in an initial derivation d''_1 of G_1 there is a derivation d'_2 that may be appended to d_2 resulting in an initial derivation d''_2 of G_2 such that $(d''_1, d''_2) \in R$ (and vice versa).

For special purposes, we make use of more special and simple kinds of (bi)simulations. For example, (*a*) dropping the reverse assumptions on well-formedness and soundness still allows for the verification of the preservation but not for the reflection of properties, (*b*) stating that the derivations d'_1 and d'_2 have both length 1 results in strong (bi)simulations and is suitable when the relationship between G_1 and G_2 is indeed that close, which is of course the case if $G_1 = G_2$ and some inter-semantics intra-formalism property is to be verified, and (*c*) not using initial derivations but well-formed configurations of G_1 and G_2 instead, which is useful in cases where no further invariants are required and are to be established along the way.

Besides (bi)simulation relations, we also employ maps. Firstly, we use bijective maps on configurations and labels allowing for simple proofs. Secondly, we use maps of initial derivations directly for relating linear and branching semantics and for translating (the nonexistence of) infinite initial derivations.

Common to all the presented approaches is a relationship between the initial derivations (or an abstraction thereof in the form of the last configuration for finite derivations) that is compatible with the corresponding step relations of the semantics.

Finally, once the initial derivations of two semantics are connected, we usually have to state some further property-specific assumptions to be able to translate for example the marked language between the two semantics.

6.2.5. Evaluation of our Hierarchical Framework of Semantics

We believe that our hierarchical framework of semantics presented in this section is *extendable* by integration of further locales as needed due to its fine grained structure. Moreover, we believe that it is *interpretable* for many further discrete event formalisms such as Turing machines [346], context-sensitive grammars [79], Petri nets [272], probabilistic automata [277], Büchi automata [65], input/output automata [227], Statecharts [152], and timed automata [12]. Finally, the uncluttered and intuitive proofs that we constructed for the verification of our concrete controller synthesis algorithm make us believe that our framework, which stabilized rather soon during this verification process, is also *applicable* to many further problems on discrete event systems in a natural way as well.

Our framework supports also Büchi automata because we allow for infinite derivations and marking conditions that are defined on entire derivations. However, we do not support formalisms such as hybrid automata [10] because the domain of our derivations is given by the natural numbers.

6.2.6. Additional Semantics for EPDA

In section 2.2|p.18, we introduced the branching semantics $epdaH$ for EPDA in which configurations contain a history variable but no scheduler variable. As explained in this section, we have also defined the standard linear semantics $epdaS$ for EPDA in which configurations contain no history variable but a scheduler variable and have proven the equivalence of $epdaS$ and $epdaH$. In fact, we relied on a further linear semantics $epdaHS$ where configurations contain a scheduler and a history variable. The equivalence of the three semantics has been proven in two steps by two interpretations of translational theories. The first translational theory relates the two linear semantics $epdaS$ and $epdaHS$ and allows to add the history variable to $epdaS$ in the expected way. Then, the second translational theory relates the linear semantics $epdaHS$ and the branching semantics $epdaH$ by allowing to drop the scheduler from the configurations of the $epdaHS$ semantics. The interpretation of the translational theories requires, of course, proofs of the satisfaction of the concretizations of the assumptions of the translational theories. Note, the same translational theories are also used for the equivalence proofs for the semantics of Parsers discussed in the next subsection.

We omit the $epdaHS$ semantics, but we give the key definitions of $epdaS$ corresponding to the definitions given in section 2.2|p.18 for $epdaH$.

Definition 6.1: «(Initial) Configurations of an EPDA in $epdaS$ »

If q is from Q , s is a word over Γ not containing \square , and w is a word over Σ , then configurations of an EPDA are of the following form.

$$\langle epdaS\text{-conf-state}=q, epdaS\text{-conf-scheduler}=w, epdaS\text{-conf-stack}=s @ [\square] \rangle$$

If w is a word over Σ , then the initial configurations are of the form $\langle q_0, w, [\square] \rangle$.

When considered closely, the step relations of $epdaS$ and $epdaH$ appear to be quite similar. Yet, the difference between the two semantics in terms of linear vs. branching semantics is tremendous.

Definition 6.2: «Step Relation $epdaS$ -step-relation»

If q_1 and q_2 are states from Q , a is an event from Σ , w is a word over Σ , s , s_1 , and s_2 are words over Γ not containing \square , then there are the following four different kinds of steps.

- | | |
|--------|--|
| Step 1 | Edge: $\langle q_1, None, s_1, s_2, q_2 \rangle$
Pre: $\langle q_1, w, s_1 @ s @ [\square] \rangle$
Post: $\langle q_2, w, s_2 @ s @ [\square] \rangle$
Kind: execute no event, do not observe \square at end of stack |
| Step 2 | Edge: $\langle q_1, None, s_1 @ [\square], s_2 @ [\square], q_2 \rangle$
Pre: $\langle q_1, w, s_1 @ [\square] \rangle$
Post: $\langle q_2, w, s_2 @ [\square] \rangle$
Kind: execute no event, observe \square at end of stack |

Step 3	Edge: $\langle q_1, \text{Some } a, s_1, s_2, q_2 \rangle$ Pre: $\langle q_1, [a] @ w, s_1 @ s @ [\square] \rangle$ Post: $\langle q_2, w, s_2 @ s @ [\square] \rangle$ Kind: execute event a , do not observe \square at end of stack
Step 4	Edge: $\langle q_1, \text{Some } a, s_1 @ [\square], s_2 @ [\square], q_2 \rangle$ Pre: $\langle q_1, [a] @ w, s_1 @ [\square] \rangle$ Post: $\langle q_2, w, s_2 @ [\square] \rangle$ Kind: execute event a , observe \square at end of stack

In the *epdaS* semantics a given EPDA is deterministic if there is a most one step from any reachable configuration.

Definition 6.3: «Determinism for *epdaS*»

An EPDA is deterministic in *epdaS*, if whenever an initial derivation leads to a configuration c and *epdaS*-step-relation $G \ c \ e_1 \ c_1$ and *epdaS*-step-relation $G \ c \ e_2 \ c_2$ are two applicable steps, then $e_1 = e_2$ and $c_1 = c_2$.

Finally, the marked and unmarked words and languages of an EPDA are obtained by comparing a scheduler in a reachable configuration with the initial scheduler in the initial derivation at hand.

Definition 6.4: «unmarked-language and marked-language for *epdaS*»

The unmarked language of an EPDA is given by all words v over Σ such that some configuration c is reachable by some initial derivation d starting with configuration c_0 such that *epdaS*-conf-scheduler $c_0 = v @ \text{epdaS}$ -conf-scheduler c . For a word to be in the marked language of the EPDA, we additionally require that *epdaS*-conf-state c is marking state from F .

Finally, we present an example in which we demonstrate the similarities between *epdaH* and *epdaS* derivations.

Example 6.2: «Derivations in *epdaS* and *epdaH*»

Corresponding Derivations of the DPDA G from Example 2.1|p.22:

$\rightarrow_{\text{epdaH},G}^{e_1} \langle 1, a, \bullet \square \rangle$	$\rightarrow_{\text{epdaS},G}^{e_1} \langle 1, abb, \bullet \square \rangle$
$\rightarrow_{\text{epdaH},G}^{e_2} \langle 1, aa, \bullet \bullet \square \rangle$	$\rightarrow_{\text{epdaS},G}^{e_2} \langle 1, bb, \bullet \bullet \square \rangle$
$\rightarrow_{\text{epdaH},G}^{e_3} \langle 2, aab, \bullet \square \rangle$	$\rightarrow_{\text{epdaS},G}^{e_3} \langle 2, b, \bullet \square \rangle$
$\rightarrow_{\text{epdaH},G}^{e_4} \langle 2, aabb, \square \rangle$	$\rightarrow_{\text{epdaS},G}^{e_3} \langle 2, , \square \rangle$
$\rightarrow_{\text{epdaH},G}^{e_5} \langle 3, aabb, \square \rangle$	$\rightarrow_{\text{epdaS},G}^{e_4} \langle 3, , \square \rangle$

id	edges of G	id	edges of G
e_1	$\langle 0, \text{Some } a, \square, \bullet \square, 1 \rangle$	e_2	$\langle 1, \text{Some } a, \bullet, \bullet \bullet, 1 \rangle$
e_3	$\langle 1, \text{Some } b, \bullet, -, 2 \rangle$	e_4	$\langle 2, \text{Some } b, \bullet, -, 2 \rangle$
e_5	$\langle 2, \text{None}, \square, \square, 3 \rangle$		

6.2.7. Additional Semantics for Parsers

As for EPDA, we defined the standard linear semantics *parserS* for Parsers as it has advantages over the branching semantics *parserHF* when verifying properties that have simpler definitions in *parserS* such as determinism. We established the equivalence of the semantics *parserS* and *parserHF* by relating them step-wise to the further linear semantics *parserHFS* and *parserFS* as follows using interpretations of abstract translational theories. The linear semantics *parserS*, which is introduced below, has a scheduler variable in its configurations. We add a fixed scheduler variable to the configurations resulting in the *parserFS* semantics. Note, the fixed scheduler can be computed by replaying the steps of a derivation in *parserS*, which results in additional complexity in proofs. Then, we add a history variable (as between the EPDA semantics *epdaS* and *epdaHS*) to the configurations resulting in the semantics *parserHFS*. Finally, we remove the scheduler variable (as between the EPDA semantics *epdaHS* and *epdaH*) from the configurations to obtain the branching semantics *parserHF*.

We omit the intermediate *parserFS* and *parserHFS* semantics, but we introduce the key definitions of the *parserS* semantics that correspond to the definitions given in section 2.3|p.22 for *parserHF*.

Definition 6.5: «(Initial) Configurations of a Parser in *parserS*»

If s is a word over N , q is from N , and v is a word Σ not containing $\$$, then configurations of a Parser are of the following form.

$$\langle \text{parserS-conf-stack} = s @ [q], \text{parserS-conf-scheduler} = v @ [\$] \rangle$$

If w is a word over Σ not containing $\$$, then the initial configurations are of the form $\langle [q_0], w @ [\$] \rangle$.

For the step relation of *parserS*, we only have to consider the two patterns of rules given in Definition 2.11|p.24 resulting in two kinds of steps. Hence, the step relation of *parserS* has fewer kinds of steps compared to the step relation of *parserHF* where the history and fixed scheduler variables resulted in many different kinds of steps.

Definition 6.6: «Step Relation *parserS*-step-relation»

If q_1 and q_2 are from N , s , s_1 , and s_2 are words over N , v_1 , v_2 , and v_3 are words over Σ not containing $\$$, then there are the following kinds of steps.

Step 1	Rule: $\langle s_1 @ [q_1], v_1 @ v_2, s_2 @ [q_2], v_2 \rangle$ Pre: $\langle s @ s_1 @ [q_1], v_1 @ v_2 @ v_3 \rangle$ Post: $\langle s @ s_2 @ [q_2], v_2 @ v_3 \rangle$
Step 2	Rule: $\langle s_1 @ [q_1], v_1 @ v_2 @ [\$], s_2 @ [q_2], v_2 @ [\$] \rangle$ Pre: $\langle s @ s_1 @ [q_1], v_1 @ v_2 @ [\$] \rangle$ Post: $\langle s @ s_2 @ [q_2], v_2 @ [\$] \rangle$

As for the *epdaS* semantics of EPDA, we define a given Parser to be deterministic if there is at most one step applicable to every configuration reachable from any initial configuration (confer to Definition 6.3|p.113).

Finally, the marked language and the unmarked language of a Parser is also obtained similarly to the case for *epdaS*. However, we must incorporate the fixed scheduler of *parserS*, which has to be recursively computed, into our definition. That is, the events executed by a Parser in the *parserS* semantics is given by the events removed from the schedule (obtained by comparison with the initial configuration of the initial derivation at hand) and the fixed scheduler where the possibly trailing \$ element is removed. The recursive computation of the fixed scheduler results in a technically more complex and undesirable definition.

Definition 6.7: «unmarked-language and marked-language for *parserS*»

The unmarked language of a Parser is given by all words $v @ f$ over Σ such that some configuration c is reachable after n steps by some initial derivation d starting with configuration c_0 such that *parserS-conf-scheduler* $c_0 = v @ \text{parserS-conf-scheduler } c$ and such that f is the fixed scheduler obtained from replaying the derivation d up to the index n where the possibly trailing \$ has been removed from f .

For such an unmarked word to be also in the marked language of the Parser, we additionally require that *parserS-conf-stack* c is of the form $s @ [q]$ where q is a marking stack element from F .

Finally, we present an example in which we demonstrate the similarities between two initial derivations using the Parser semantics *parserHF* and *parserS*. Both of these two initial derivations apply the same sequence of rules, execute the same unmarked words, and mark the same word *aabb* by means of the last configuration.

Example 6.3: «Derivations in *parserHF* and *parserS*»

Corresponding Derivations of the Parser G from Example 2.2|p.27:

	$\langle \quad , \quad , \quad 0 \rangle$		$\langle \quad 0, aabb\$ \rangle$
$\xrightarrow{r_1}$	$\text{parserHF}, G \langle \quad , a \quad , \square \bullet 1 \rangle$	$\xrightarrow{r_1}$	$\text{parserS}, G \langle \square \bullet 1, abb\$ \rangle$
$\xrightarrow{r_2}$	$\text{parserHF}, G \langle \quad , aa \quad , \square \bullet \bullet 1 \rangle$	$\xrightarrow{r_2}$	$\text{parserS}, G \langle \square \bullet \bullet 1, bb\$ \rangle$
$\xrightarrow{r_3}$	$\text{parserHF}, G \langle \quad , aab \quad , \square \bullet 2 \rangle$	$\xrightarrow{r_3}$	$\text{parserS}, G \langle \square \bullet 2, b\$ \rangle$
$\xrightarrow{r_4}$	$\text{parserHF}, G \langle \quad , aabb, \quad \square 2 \rangle$	$\xrightarrow{r_4}$	$\text{parserS}, G \langle \quad \square 2, \quad \$ \rangle$
$\xrightarrow{r_5}$	$\text{parserHF}, G \langle \quad , aabb, \quad 3 \rangle$	$\xrightarrow{r_5}$	$\text{parserS}, G \langle \quad 3, \quad \$ \rangle$

In this simple example, the fixed scheduler is permanently empty for both given derivations because the rules of the Parser G that are used in the derivations have empty *rule-scheduler-push* components. This is, however, not the case for the Parsers that we obtain during the application of our concrete controller synthesis algorithm in subsection 5.2.2|p.73.

6.3. ISABELLE-BASED VERIFICATION OF THE TRANSLATION OF DETERMINISTIC PUSHDOWN AUTOMATA INTO LR(1)-CONTEXT-FREE GRAMMARS

We provide an intuitive explanation of our formal proof of a theorem that plays an important role for the overall soundness of our concrete controller synthesis algorithm and parsing theory in general. To the best of our knowledge, no formal or informal proof of this theorem has been presented previously, but it has been conjectured in [189] and [325, Volume II, Proposition 6.41, p. 53].

As discussed briefly in subsection 5.2.1|p.69, we apply in our concrete controller synthesis algorithm the operation $F_{SDPDA \rightarrow CFG, Std}$, which was first presented in [189], to an SDPDA G that does not mark a word in any initial marking derivation twice. The result $F_{SDPDA \rightarrow CFG, Std} G$ of this application is a CFG that is equivalent to G as it generates the same marked language. As the next step, we apply the operation $F_{CFG-Trim}$ to $F_{SDPDA \rightarrow CFG, Std} G$ to remove the least sets of nonterminals and productions to obtain a CFG G' where every nonterminal and every production occurs in at least one initial marking derivation. Note, the sequential application of $F_{SDPDA \rightarrow CFG, Std}$ and $F_{CFG-Trim}$ defines the compound construction $F_{SDPDA \rightarrow LR(1), Std}$, as also explained in section 5.2.1|p.71. Finally, the theorem that is given below (in Isabelle notation) states that G' satisfies the LR(1)-CFG property. The LR(1)-CFG property is introduced subsequently and its satisfaction means in this case that the determinism of the given SDPDA G has been transferred successfully to G' as desired.

Theorem 6.3: « $F_{SDPDA \rightarrow LR(1), Std}$ Translates an SDPDA into an LR(1)-CFG»

$\text{1 valid-sdpda } G$
 $\implies \text{2 } \neg \text{duplicate-marking } G$
 $\implies \text{3 valid-cfg } G'$
 $\implies \text{4 cfg-sub } G' (F_{SDPDA \rightarrow CFG, Std} G)$
 $\implies \text{5 cfg-nonterminals } G' =$
 $\quad \text{cfgLM-accessible-nonterminals } G'$
 $\quad \cap \text{cfgSTD-nonblocking-nonterminals } G'$
 $\implies \text{6 cfg-LR } G' (Suc 0)$

1 states that G is a valid SDPDA, **2** states that G has no initial marking derivation in $epdaS$ where two configurations at distinct positions in the derivation agree on their schedule and contain both a marking state, **3** states that G' is a well formed CFG, **4** states that G' is contained in $F_{SDPDA \rightarrow CFG, Std} G$ in the sense that they agree on the set of events and the initial nonterminal and that the sets of nonterminals and productions of G' are contained in the corresponding sets of $F_{SDPDA \rightarrow CFG, Std} G$, **5** states that every nonterminal A of G' is accessible (that is, there is some initial derivation that reaches a configurations containing A) and eliminable (that is, there is some derivation of G' that starts with the configuration $\langle A \rangle$ and ends with a configuration without nonterminals), and the conclusion **6** states that G' satisfies the LR(1)-CFG property introduced below.

Before continuing with the introduction of the LR(k)-CFG property, there are two observations on the given theorem to be mentioned. Firstly, the theorem states a slightly stronger property than necessary as it does not only consider the unique maximal trim CFG contained in $F_{SDPDA \rightarrow CFG, Std} G$ (observe that G' may vary in the set of productions), which does not exist whenever G has an empty marked language. Secondly, the theorem is also applicable to our optimized construction $F_{SDPDA \rightarrow CFG, Opt}$ because the CFGs obtained by $F_{SDPDA \rightarrow CFG, Std}$ and $F_{SDPDA \rightarrow CFG, Opt}$ are equal after the application of the operation $F_{CFG-Trim}$.

We now provide the definition of the operation $F_{SDPDA \rightarrow CFG, Std}$ occurring in the theorem above (see Example 6.4|p.118 for a first example of an application of this construction).

Definition 6.8: « $F_{SDPDA \rightarrow CFG, Std}$ »

Nonterminals of the CFG: The nonterminals of the resulting CFG are of the two forms $L_{q,X}$ and $L_{q,X,p}$ where q and p are states of the given SDPDA G and where X is a stack-element of the given SDPDA G .

Initial Nonterminal of the CFG: The initial nonterminal $L_{q_0, \square}$ is given by the initial state and the end-of-stack element of the SDPDA G .

Event alphabet of the CFG: The event alphabet is given by the event alphabet of the input SDPDA G .

Productions of the CFG: The productions of the resulting CFG are constructed from the three kinds of edges of the SDPDA G and from its marking states. In three of these four cases the resulting productions (below the horizontal line) also depend on further elements (above the horizontal line) chosen from the given SDPDA G .

<i>Executing Edge</i>	$e = \langle q_1, \text{Some } a, [X], [X], q_2 \rangle \text{ and } q \in \text{epda-states } G$
	$\begin{array}{l} L_{q_1, X, q} \longrightarrow a L_{q_2, X, q} \\ L_{q_1, X} \longrightarrow a L_{q_2, X} \end{array}$
<i>Pushing Edge</i>	$e = \langle q_1, \text{None}, [X_1], [X_2, X_1], q_2 \rangle \text{ and } \{q', q''\} \subseteq \text{epda-states } G$
	$\begin{array}{l} L_{q_1, X_1, q''} \longrightarrow L_{q_2, X_2, q'} L_{q', X_1, q''} \\ L_{q_1, X_1} \longrightarrow L_{q_2, X_2, q'} L_{q', X_2} \\ L_{q_1, X_1} \longrightarrow L_{q_2, X_2} \end{array}$
<i>Popping Edge</i>	$e = \langle q_1, \text{None}, [X], [], q_2 \rangle$
	$L_{q_1, X, q_2} \longrightarrow []$
<i>Marking States</i>	$q \in \text{epda-marking } G \text{ and } X \in \text{epda-gamma } G$
	$L_{q, X} \longrightarrow []$

In derivation trees (see for example block 6|p.120), we also write $q X$ and $q_1 X q_2$ instead of $L_{q, X}$ and L_{q_1, X, q_2} , respectively.

Before presenting our running example, we provide the following small set of notations to enhance the readability of this section.

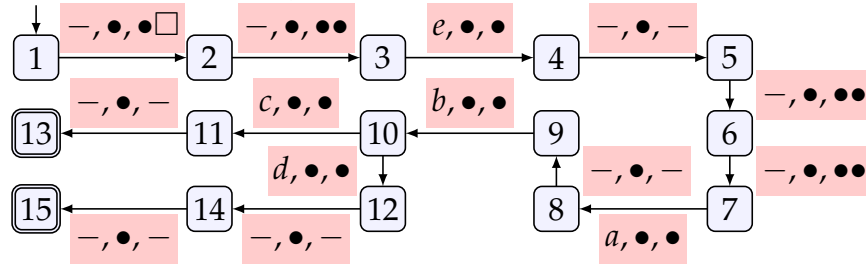
Figure 6.4: «Notations Used in This Section»

Variable Names: G' is the CFG equal to $F_{SDPDA \rightarrow LR(1), Std} G$ using the SDPDA G , c is a configuration of G or G' , n and k are natural numbers, a through e are events, α and σ are words of events, A is a nonterminal of G' , S is the initial nonterminal of G' , δ and ω are words of events and nonterminals, e is an edge of G , ρ is a production of G' , π is a word of either productions of G' or of edges of G , and d is a derivation of G or G' .

In the following running example, which is used throughout the remainder of this section, we provide an SDPDA G with two distinct maximal initial derivations using the *epdaH* semantics and the $LR(1)$ -CFG G' obtained from the SDPDA G with the initial derivations corresponding to the two *epdaH* derivations of G using the *cfgLM* and *cfgRM* semantics as well as using derivation trees.

Example 6.4: «Running Example for section 6.3|p.116 (Part 1/4)»

(1) *Input SDPDA G*: The only two initial derivations of G in the semantics *epdaH* that cannot be extended are given in block 2|p.118.



(2) *epdaH* Derivations $d_{1,h}$ and $d_{2,h}$ of G :

$\rightarrow_{epdaH,G}^{e_1} (1, \quad , \quad \square)$	$\rightarrow_{epdaH,G}^{e_1} (1, \quad , \quad \square)$
$\rightarrow_{epdaH,G}^{e_2} (2, \quad , \quad \bullet \square)$	$\rightarrow_{epdaH,G}^{e_1} (2, \quad , \quad \bullet \square)$
$\rightarrow_{epdaH,G}^{e_3} (3, \quad , \quad \bullet \bullet \square)$	$\rightarrow_{epdaH,G}^{e_2} (3, \quad , \quad \bullet \bullet \square)$
$\rightarrow_{epdaH,G}^{e_4} (4, e \quad , \quad \bullet \bullet \square)$	$\rightarrow_{epdaH,G}^{e_3} (4, e \quad , \quad \bullet \bullet \square)$
$\rightarrow_{epdaH,G}^{e_5} (5, e \quad , \quad \bullet \square)$	$\rightarrow_{epdaH,G}^{e_4} (5, e \quad , \quad \bullet \square)$
$\rightarrow_{epdaH,G}^{e_6} (6, e \quad , \quad \bullet \bullet \square)$	$\rightarrow_{epdaH,G}^{e_5} (6, e \quad , \quad \bullet \bullet \square)$
$\rightarrow_{epdaH,G}^{e_7} (7, e \quad , \quad \bullet \bullet \bullet \square)$	$\rightarrow_{epdaH,G}^{e_6} (7, e \quad , \quad \bullet \bullet \bullet \square)$
$\rightarrow_{epdaH,G}^{e_8} (8, ea \quad , \quad \bullet \bullet \bullet \square)$	$\rightarrow_{epdaH,G}^{e_7} (8, ea \quad , \quad \bullet \bullet \bullet \square)$
$\rightarrow_{epdaH,G}^{e_9} (9, ea \quad , \quad \bullet \bullet \square)$	$\rightarrow_{epdaH,G}^{e_8} (9, ea \quad , \quad \bullet \bullet \square)$
$\rightarrow_{epdaH,G}^{e_{10.2}} (10, eab \quad , \quad \bullet \bullet \square)$	$\rightarrow_{epdaH,G}^{e_9} (10, eab \quad , \quad \bullet \bullet \square)$
$\rightarrow_{epdaH,G}^{e_{12}} (12, eabd, \quad \bullet \square)$	$\rightarrow_{epdaH,G}^{e_{10.1}} (11, eabc, \quad \bullet \bullet \square)$
$\rightarrow_{epdaH,G}^{e_{14}} (15, eabd, \quad \square)$	$\rightarrow_{epdaH,G}^{e_{11}} (13, eabc, \quad \bullet \square)$

(3) Output LR(1)-CFG G' : The only two initial derivations of G' that cannot be extended are given in the subsequent blocks in the two semantics $cfgLM$ and $cfgRM$ and in the form of derivation trees.

id	productions of G'	id	edges of G
ρ_1	$L_{1,\square} \rightarrow L_{2,\bullet,15} L_{15,\square}$	e_1	$(1, None, \square, \bullet\square, 2)$
ρ_2	$L_{15,\square} \rightarrow$	—	—
ρ_3	$L_{2,\bullet,15} \rightarrow L_{3,\bullet,5} L_{5,\bullet,15}$	e_2	$(2, None, \bullet, \bullet\bullet, 3)$
ρ_4	$L_{5,\bullet,15} \rightarrow L_{6,\bullet,14} L_{14,\bullet,15}$	e_5	$(5, None, \bullet, \bullet\bullet, 6)$
ρ_5	$L_{14,\bullet,15} \rightarrow$	e_{14}	$(14, None, \bullet, -, 15)$
ρ_6	$L_{6,\bullet,14} \rightarrow L_{7,\bullet,9} L_{9,\bullet,14}$	e_6	$(6, None, \bullet, \bullet\bullet, 7)$
ρ_7	$L_{9,\bullet,14} \rightarrow b L_{10,\bullet,14}$	e_9	$(9, Some\ b, \bullet, \bullet, 10)$
ρ_8	$L_{10,\bullet,14} \rightarrow d L_{12,\bullet,14}$	$e_{10.2}$	$(10, Some\ d, \bullet, \bullet, 12)$
ρ_9	$L_{12,\bullet,14} \rightarrow$	e_{12}	$(12, None, \bullet, -, 14)$
ρ_{10}	$L_{7,\bullet,9} \rightarrow a L_{8,\bullet,9}$	e_7	$(7, Some\ a, \bullet, \bullet, 8)$
ρ_{11}	$L_{8,\bullet,9} \rightarrow$	e_8	$(8, None, \bullet, -, 9)$
ρ_{12}	$L_{3,\bullet,5} \rightarrow e L_{4,\bullet,5}$	e_3	$(3, Some\ e, \bullet, \bullet, 4)$
ρ_{13}	$L_{4,\bullet,5} \rightarrow$	e_4	$(4, None, \bullet, -, 5)$
ρ_{14}	$L_{1,\square} \rightarrow L_{2,\bullet}$	e_1	$(1, None, \square, \bullet\square, 2)$
ρ_{15}	$L_{2,\bullet} \rightarrow L_{3,\bullet,5} L_{5,\bullet}$	e_2	$(2, None, \bullet, \bullet\bullet, 3)$
ρ_{16}	$L_{5,\bullet} \rightarrow L_{6,\bullet,13} L_{13,\bullet}$	e_5	$(5, None, \bullet, \bullet\bullet, 6)$
ρ_{17}	$L_{13,\bullet} \rightarrow$	—	—
ρ_{18}	$L_{6,\bullet,13} \rightarrow L_{7,\bullet,9} L_{9,\bullet,13}$	e_6	$(6, None, \bullet, \bullet\bullet, 7)$
ρ_{19}	$L_{9,\bullet,13} \rightarrow b L_{10,\bullet,13}$	e_9	$(9, Some\ b, \bullet, \bullet, 10)$
ρ_{20}	$L_{10,\bullet,13} \rightarrow c L_{11,\bullet,13}$	$e_{10.1}$	$(10, Some\ c, \bullet, \bullet, 11)$
ρ_{21}	$L_{11,\bullet,13} \rightarrow$	e_{11}	$(11, None, \bullet, -, 13)$

(4) $cfgLM$ Derivations $d_{1,lm}$ and $d_{2,lm}$ of G' :

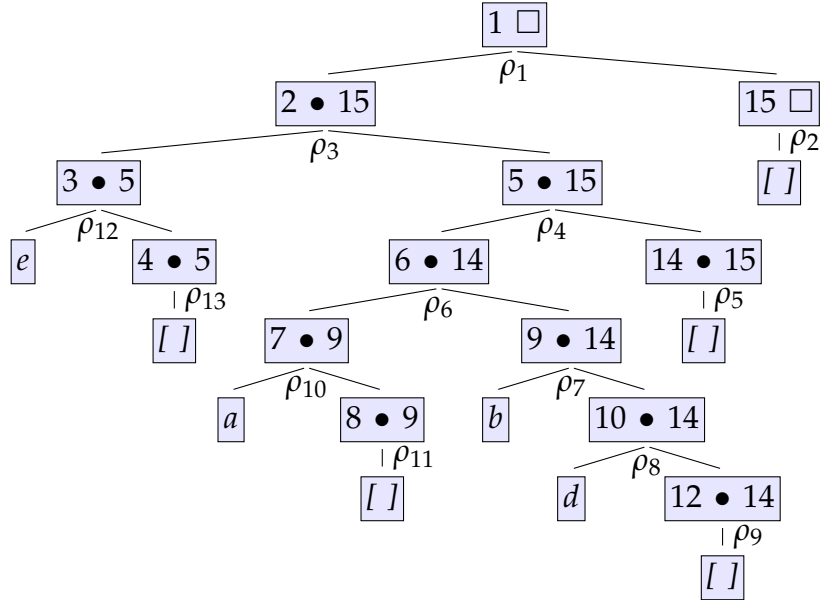
$L_{1,\square}$	$L_{1,\square}$
$\rightarrow_{cfgLM,G'}^{\rho_1} L_{2,\bullet,15} L_{15,\square}$	$\rightarrow_{cfgLM,G'}^{\rho_{14}} L_{2,\bullet}$
$\rightarrow_{cfgLM,G'}^{\rho_3} L_{3,\bullet,5} L_{5,\bullet,15} L_{15,\square}$	$\rightarrow_{cfgLM,G'}^{\rho_{15}} L_{3,\bullet,5} L_{5,\bullet}$
$\rightarrow_{cfgLM,G'}^{\rho_{12}} e L_{4,\bullet,5} L_{5,\bullet,15} L_{15,\square}$	$\rightarrow_{cfgLM,G'}^{\rho_{12}} e L_{4,\bullet,5} L_{5,\bullet}$
$\rightarrow_{cfgLM,G'}^{\rho_{13}} e L_{5,\bullet,15} L_{15,\square}$	$\rightarrow_{cfgLM,G'}^{\rho_{13}} e L_{5,\bullet}$
$\rightarrow_{cfgLM,G'}^{\rho_4} e L_{6,\bullet,14} L_{14,\bullet,15} L_{15,\square}$	$\rightarrow_{cfgLM,G'}^{\rho_{16}} e L_{6,\bullet,13} L_{13,\bullet}$
$\rightarrow_{cfgLM,G'}^{\rho_6} e L_{7,\bullet,9} L_{9,\bullet,14} L_{14,\bullet,15} L_{15,\square}$	$\rightarrow_{cfgLM,G'}^{\rho_{18}} e L_{7,\bullet,9} L_{9,\bullet,13} L_{13,\bullet}$
$\rightarrow_{cfgLM,G'}^{\rho_{10}} e a L_{8,\bullet,9} L_{9,\bullet,14} L_{14,\bullet,15} L_{15,\square}$	$\rightarrow_{cfgLM,G'}^{\rho_{10}} e a L_{8,\bullet,9} L_{9,\bullet,13} L_{13,\bullet}$
$\rightarrow_{cfgLM,G'}^{\rho_{11}} e a L_{9,\bullet,14} L_{14,\bullet,15} L_{15,\square}$	$\rightarrow_{cfgLM,G'}^{\rho_{11}} e a L_{9,\bullet,13} L_{13,\bullet}$
$\rightarrow_{cfgLM,G'}^{\rho_7} e a b L_{10,\bullet,14} L_{14,\bullet,15} L_{15,\square}$	$\rightarrow_{cfgLM,G'}^{\rho_{19}} e a b L_{10,\bullet,13} L_{13,\bullet}$
$\rightarrow_{cfgLM,G'}^{\rho_8} e a b d L_{12,\bullet,14} L_{14,\bullet,15} L_{15,\square}$	$\rightarrow_{cfgLM,G'}^{\rho_{20}} e a b c L_{11,\bullet,13} L_{13,\bullet}$
$\rightarrow_{cfgLM,G'}^{\rho_9} e a b d L_{14,\bullet,15} L_{15,\square}$	$\rightarrow_{cfgLM,G'}^{\rho_{21}} e a b c L_{13,\bullet}$
$\rightarrow_{cfgLM,G'}^{\rho_5} e a b d L_{15,\square}$	$\rightarrow_{cfgLM,G'}^{\rho_{17}} e a b c$
$\rightarrow_{cfgLM,G'}^{\rho_2} e a b d$	

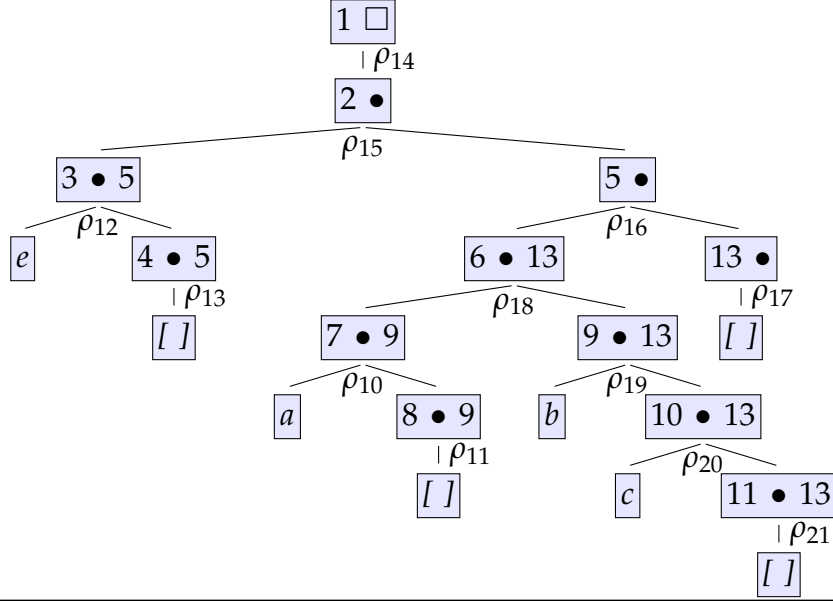
(5) *cfgRM Derivations $d_{1,rm}$ and $d_{2,rm}$ of G'* : Note, the obvious connection between the *cfgLM* and *epdaH* derivations is not available for the *cfgRM* derivations.

$ \begin{aligned} & \rightarrow^{\rho_1}_{cfgRM, G'} L_{1, \square} \\ & \rightarrow^{\rho_2}_{cfgRM, G'} L_{2, \bullet, 15} L_{15, \square} \\ & \rightarrow^{\rho_3}_{cfgRM, G'} L_{2, \bullet, 15} \\ & \rightarrow^{\rho_4}_{cfgRM, G'} L_{3, \bullet, 5} L_{5, \bullet, 15} \\ & \rightarrow^{\rho_5}_{cfgRM, G'} L_{3, \bullet, 5} L_{6, \bullet, 14} L_{14, \bullet, 15} \\ & \rightarrow^{\rho_6}_{cfgRM, G'} L_{3, \bullet, 5} L_{6, \bullet, 14} \\ & \rightarrow^{\rho_7}_{cfgRM, G'} L_{3, \bullet, 5} L_{7, \bullet, 9} L_{9, \bullet, 14} \\ & \rightarrow^{\rho_8}_{cfgRM, G'} L_{3, \bullet, 5} L_{7, \bullet, 9} b L_{10, \bullet, 14} \\ & \rightarrow^{\rho_9}_{cfgRM, G'} L_{3, \bullet, 5} L_{7, \bullet, 9} b d L_{12, \bullet, 14} \\ & \rightarrow^{\rho_{10}}_{cfgRM, G'} L_{3, \bullet, 5} L_{7, \bullet, 9} b d \\ & \rightarrow^{\rho_{11}}_{cfgRM, G'} L_{3, \bullet, 5} a L_{8, \bullet, 9} b d \\ & \rightarrow^{\rho_{12}}_{cfgRM, G'} L_{3, \bullet, 5} a b d \\ & \rightarrow^{\rho_{13}}_{cfgRM, G'} e L_{4, \bullet, 5} a b d \\ & \rightarrow^{\rho_{14}}_{cfgRM, G'} e a b d \end{aligned} $	$ \begin{aligned} & \rightarrow^{\rho_{14}}_{cfgRM, G'} L_{1, \square} \\ & \rightarrow^{\rho_{15}}_{cfgRM, G'} L_{2, \bullet} \\ & \rightarrow^{\rho_{16}}_{cfgRM, G'} L_{3, \bullet, 5} L_{5, \bullet} \\ & \rightarrow^{\rho_{17}}_{cfgRM, G'} L_{3, \bullet, 5} L_{6, \bullet, 13} L_{13, \bullet} \\ & \rightarrow^{\rho_{18}}_{cfgRM, G'} L_{3, \bullet, 5} L_{6, \bullet, 13} \\ & \rightarrow^{\rho_{19}}_{cfgRM, G'} L_{3, \bullet, 5} L_{7, \bullet, 9} L_{9, \bullet, 13} \\ & \rightarrow^{\rho_{20}}_{cfgRM, G'} L_{3, \bullet, 5} L_{7, \bullet, 9} b c L_{10, \bullet, 13} \\ & \rightarrow^{\rho_{21}}_{cfgRM, G'} L_{3, \bullet, 5} L_{7, \bullet, 9} b c \\ & \rightarrow^{\rho_{10}}_{cfgRM, G'} L_{3, \bullet, 5} a L_{8, \bullet, 9} b c \\ & \rightarrow^{\rho_{11}}_{cfgRM, G'} L_{3, \bullet, 5} a b c \\ & \rightarrow^{\rho_{12}}_{cfgRM, G'} e L_{4, \bullet, 5} a b c \\ & \rightarrow^{\rho_{13}}_{cfgRM, G'} e a b c \end{aligned} $
---	---

(6) *Derivation Tree T_1 Corresponding to $d_{1,h}$, $d_{1,lm}$, $d_{1,rm}$* :

We use derivation trees for the visualization of *cfgLM* and *cfgRM* derivations to provide a better understanding of the structure of how productions are applied and to represent *cfgLM* and *cfgRM* derivations in one common formalism.



(7) Derivation Tree T_2 Corresponding to $d_{2,l}, d_{2,lm}, d_{2,rm}$:


We continue with the correctness proof of the construction of a *deterministic* LR(k)-Parser taken from [325, Volume II, Section 6.3, p. 31] by relying on the following definition of an LR(k)-CFG also taken from [325, Volume II, Theorem 6.39, p. 52]. While this definition is given for arbitrary natural numbers k , we are only interested in the case of $k = 1$ for the purpose of our concrete controller synthesis algorithm (see Theorem 6.3|p.116) because the steps of DPDA may only consider the next event to be executed (that is, they have only access to the first event contained in the scheduler in the *epdaS* semantics).

The LR(k)-CFG property describes the preserved determinism property of the SDPDA that has been translated to the obtained CFG. In this definition, two initial right-most derivations must be identical w.r.t. the steps that generate compatible nonterminal-free prefixes. Right-most derivations are used in [189, 325] because the derivations of the LR(k)-Parser correspond more closely to the right-most derivations than to the left-most derivations of the obtained CFG.

Definition 6.9: «cfg-LR»

A CFG G' satisfies *cfg-LR* G' k where k is a natural number if

$$\begin{aligned}
 S &\xrightarrow{\pi_1}_{cfgRM,G'} \delta_1 A_1 \sigma_1 \xrightarrow{\rho_1}_{cfgRM,G'} \delta_1 \omega_1 \sigma_1, \\
 S &\xrightarrow{\pi_2}_{cfgRM,G'} \delta_2 A_2 \sigma_2 \xrightarrow{\rho_2}_{cfgRM,G'} \delta_2 \omega_2 \sigma_2, \\
 \delta_1 \omega_1 \alpha &= \delta_2 \omega_2, \text{ and} \\
 take\ k\ \sigma_1 &= take\ k\ (\alpha \sigma_2)
 \end{aligned}$$

together imply $\delta_1 = \delta_2$, $A_1 = A_2$, and $\omega_1 = \omega_2$.

Note, the CFG G' is not required to be trim by this definition of the LR(k)-CFG property, but we only check this property for such CFG as in Theorem 6.3|p.116. Hence, we may derive a nonterminal-free configuration α' from $\delta_2 \omega_2$ using some

right-most derivation d . The productions π used for the derivation d may also be used to extend the two given right-most derivations to reach a nonterminal-free configuration in each case.

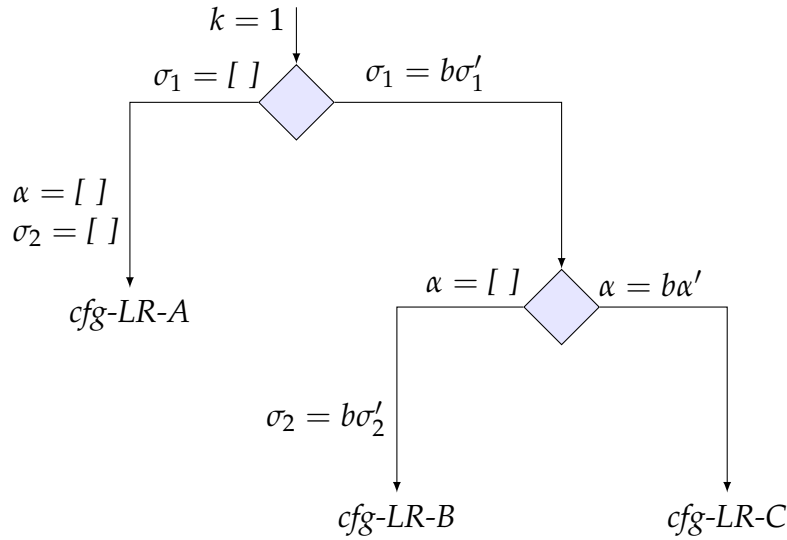
The first complication of this definition is that in the right-most semantics the steps that generate the considered prefix are not solely executed at the beginning of the given initial derivation but are distributed throughout the derivation and, as for the mentioned derivation d , at the end. As a consequence of this problem, we must relate left-most and right-most derivations of the CFG in the proof presented subsequently.

The second complication is that the LR(k)-CFG property is asymmetric and, due to the test for compatibility of the reached configurations, artificially compacted. We approach this problem by splitting the LR(k)-CFG property for the case of $k = 1$ into three more basic properties in the next subsection based on a case distinction on the variables σ_1 , α , and σ_2 . Also, we provide a more intuitive explanation for the three resulting properties below.

6.3.1. Basic Decomposition of the LR(1)-CFG Property

As just pointed out, we perform a case distinction, as given in the following figure, on the variables σ_1 , α , and σ_2 occurring in the definition of the LR(k)-CFG property above. Also, we assume that $k = 1$ in the subsequent presentation because we must verify for a given SDPDA that it satisfies the LR(1)-CFG property. From the case distinction, we obtain three relevant cases resulting in the three subsequent definitions of *cfg-LR-A*, *cfg-LR-B*, and *cfg-LR-C*. Note, for three cases, we immediately obtain two, one, and zero additional conditions on the variables, respectively, which are attached to the arrows in the figure below.

Figure 6.5: «Case Distinction for *cfg-LR-A*, *cfg-LR-B*, and *cfg-LR-C*»



Subsequently, we discuss our verification approach for these three properties.

The *cfg-LR-A* property states that when the CFG reaches identical configurations in two right-most derivations that the previous configurations in the two derivations are also identical. This implies that the two derivations are pointwise identical in the configurations and productions applied.

Definition 6.10: «cfg-LR-A»

A CFG G satisfies *cfg-LR-A* G if

$$\begin{aligned} S &\xrightarrow{\pi_1}_{\text{cfgRM},G'} \delta_1 A_1 \xrightarrow{\rho_1}_{\text{cfgRM},G'} \delta_1 \omega_1, \\ S &\xrightarrow{\pi_2}_{\text{cfgRM},G'} \delta_2 A_2 \xrightarrow{\rho_2}_{\text{cfgRM},G'} \delta_2 \omega_2, \text{ and} \\ \delta_1 \omega_1 &= \delta_2 \omega_2 \end{aligned}$$

together imply $\delta_1 = \delta_2$, $A_1 = A_2$, and $\omega_1 = \omega_2$.

The *cfg-LR-B* property extends the *cfg-LR-A* property because the last configurations are identical up to the possibly different nonterminal-free suffixes σ_1 and σ_2 , respectively. Moreover, the last steps also did not generate σ_1 and σ_2 and, hence, the previous configurations should also be equal up to the trailing σ_1 and σ_2 .

Definition 6.11: «cfg-LR-B»

A CFG G satisfies *cfg-LR-B* G if

$$\begin{aligned} S &\xrightarrow{\pi_1}_{\text{cfgRM},G'} \delta_1 A_1 b \sigma_1 \xrightarrow{\rho_1}_{\text{cfgRM},G'} \delta_1 \omega_1 b \sigma_1, \\ S &\xrightarrow{\pi_2}_{\text{cfgRM},G'} \delta_2 A_2 b \sigma_2 \xrightarrow{\rho_2}_{\text{cfgRM},G'} \delta_2 \omega_2 b \sigma_2, \text{ and} \\ \delta_1 \omega_1 &= \delta_2 \omega_2 \end{aligned}$$

together imply $\delta_1 = \delta_2$, $A_1 = A_2$, and $\omega_1 = \omega_2$.

Finally, the *cfg-LR-C* property states that the existence of the two given right-most derivations is a contradiction. This contradiction is expected because the common nonterminal-free prefix $\delta_1 \omega_1$ is created in two different ways by the two derivations. Also note that the last configurations match the last configurations in the case of *cfg-LR-B* for $\sigma_2 = \alpha_1 \alpha_2$ and this implies there that the corresponding previous configurations correspond to each other as well. But this implication is violated due to the reverse step using ρ_2 in the second derivation in *cfg-LR-C*.

Definition 6.12: «cfg-LR-C»

A CFG G satisfies *cfg-LR-C* G if

$$\begin{aligned} S &\xrightarrow{\pi_1}_{\text{cfgRM},G'} \delta_1 A_1 b \sigma_1 \xrightarrow{\rho_1}_{\text{cfgRM},G'} \delta_1 \omega_1 b \sigma_1, \\ S &\xrightarrow{\pi_2}_{\text{cfgRM},G'} \delta_2 \omega_2 b \alpha_1 A_2 \alpha_2 \xrightarrow{\rho_2}_{\text{cfgRM},G'} \delta_2 \omega_2 b \alpha_1 \alpha_2, \text{ and} \\ \delta_1 \omega_1 &= \delta_2 \omega_2 \end{aligned}$$

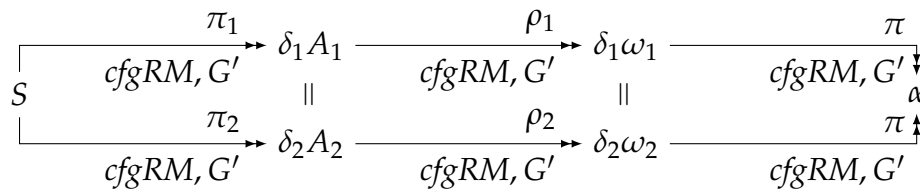
together imply *False*.

6.3.2. Verification of the *cfg-LR-A* Property

The rather trivial proof that the generated CFG satisfies the *cfg-LR-A* property proceeds (at the highest level of abstraction) as follows.

1. There is a derivation d_l in the *cfgLM* semantics from the configuration $\delta_1\omega_1$ to some nonterminal-free configuration α since the generated CFG has only nonblocking nonterminals (according to 5 in Theorem 6.3|p.116).
2. The derivation d_l can be translated into a derivation d_r between the same configurations $\delta_1\omega_1$ and α , because d_l ends in a nonterminal-free configuration and consequently the productions occurring in d_l can be simply reordered to obtain an equivalent and unique right-most derivation.
3. The derivation d_r executing the productions π can be appended to the two given initial derivations d_1 and d_2 in the *cfgRM* semantics leading to the two initial derivations d'_1 and d'_2 because $\delta_1\omega_1 = \delta_2\omega_2$.
4. Note, the generated CFG is nonambiguous with respect to the *cfgLM* semantics because the given SDPDA satisfies $\neg\text{duplicate-marking } G$ (according to 2 in Theorem 6.3|p.116).
5. By formally verifying the relationship between left-most and right-most derivations according to [325, Volume I, Section 4.3, pp. 122–], we also obtained [325, Volume I, Lemma 4.12, p. 129], which also states that a CFG is nonambiguous in *cfgLM* if and only if it is nonambiguous in *cfgRM*. Hence, the generated CFG is also nonambiguous with respect to *cfgRM*.
6. We obtain that d'_1 and d'_2 are equal because the CFG is nonambiguous in *cfgRM* and they are two initial derivations with the same final nonterminal-free configuration α . Moreover, they pointwise agree on all their positions.
7. Also, the two derivations d'_1 and d'_2 are of equal length because they are equal. Assuming that d_1 be of length $n_1 + 1$, that d_2 be of length $n_2 + 1$, and that d be of length n , we obtain that $n_1 + 1 + n = n_2 + 1 + n$, which means that $n_1 = n_2$. Then, d'_1 and d'_2 coincide in the position $n_1 = n_2$ because d'_1 and d'_2 coincide in each of their positions. Then, d_1 and d_2 coincide in the position $n_1 = n_2$ because of the definition of appending derivations. This means that $\delta_1A_1 = \delta_2A_2$. This implies $\delta_1 = \delta_2$ and $A_1 = A_2$ directly and using $\delta_1\omega_1 = \delta_2\omega_2$, we also obtain $\omega_1 = \omega_2$ as required.

Figure 6.6: «Visualization for the Proof of the *cfg-LR-A* Property»



There are two more involved steps in this proof sketch.

Firstly, for the step (4) the translation of the nonambiguity property of the given SDPDA (in the form of \neg *duplicate-marking* G using *epdaH*) by the construction $F_{SDPDA \rightarrow CFG, Std}$ to the resulting CFG (in the form of nonambiguity using *cfgLM*) is established by an involved induction on the length of the derivations.

Secondly, for the steps (2) and (5), we formalized the required relationship between *cfgLM* and *cfgRM* as follows. For every (initial) derivation in *cfgLM* (*cfgRM*) ending in a nonterminal-free configuration (called complete derivation subsequently) there is construction procedure showing that there *exists a unique* (initial) complete derivation in *cfgRM* (*cfgLM*) where the lists of productions used for the two derivations are equal up to a certain reordering. For example, $d_{1,lm}$ and $d_{1,rm}$ from block 4|p.119 and block 5|p.120 correspond to each other and have the following sequences of productions, respectively.

$$\begin{aligned}\pi_{lm}^1 &= \rho_1 \rho_3 \rho_{12} \rho_{13} \rho_4 \rho_6 \rho_{10} \rho_{11} \rho_7 \rho_8 \rho_9 \rho_5 \rho_2 \\ \pi_{rm}^1 &= \rho_1 \rho_2 \rho_3 \rho_4 \rho_5 \rho_6 \rho_7 \rho_8 \rho_9 \rho_{10} \rho_{11} \rho_{12} \rho_{13}\end{aligned}$$

Note, in the semantics *cfgLM* and *cfgRM*, we can represent derivations by the first configuration and the sequence of productions employed in the derivation because the productions are applied in a deterministic way. Moreover, the first configuration is not required for initial derivations such $d_{1,rm}$ and $d_{1,lm}$. However, the translation can be carried out easily when the derivations are visualized in the form of derivations trees as in block 6|p.120 and block 7|p.121. We apply the well-known preorder traversal as follows to obtain the *cfgLM* derivation represented by a derivation tree T in the form of a list of productions.

- *Step (a)*: start with the empty list of productions π and apply step (b) to the root node of T and then return π ,
- *Step (b)*: add the production applied to the current node to π (if any) and then apply step (a) to the subtrees of the current node from left to right and append the resulting lists of productions to π .

We also apply the preorder traversal to obtain the derivation in *cfgRM*, but we consider the subtrees in step (b) in reverse order, that is, from right to left.

Note, we formalized the recursive translation of any initial complete *cfgLM* derivation into the corresponding *cfgRM* derivation by following [325, Volume I, Section 4.3, pp. 122–]. This translation defines the required reordering of the productions based on the structure of the right hand sides of the productions occurring in the derivation. However, this relationship among *cfgLM* and *cfgRM* established for this proof is insufficient for the remaining two parts of our proof as it only covers derivations ending in a nonterminal-free configuration. Subsequently, we combine and extend these proofs to come up with a more general translation between *cfgLM* and *cfgRM* also integrating the construction $F_{SDPDA \rightarrow CFG, Std}$.

6.3.3. Extended Splitting Semantic *cfgEsplrit*

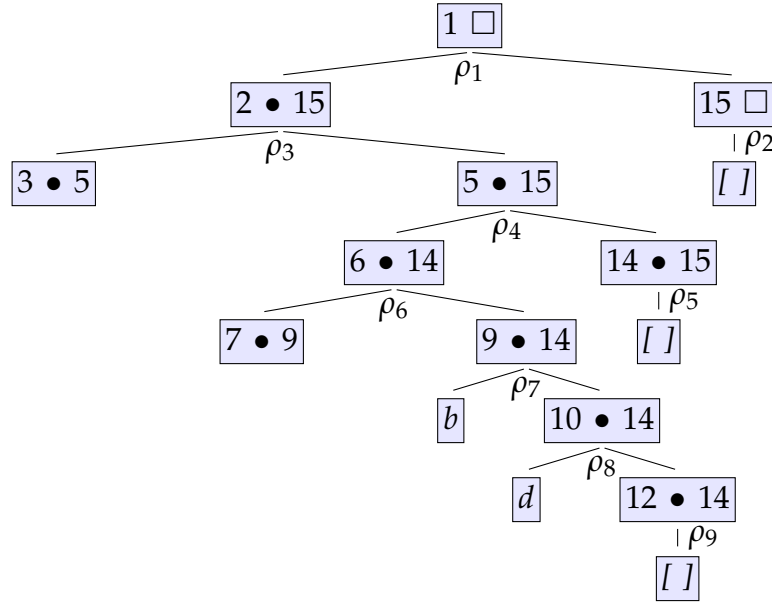
We introduce a further branching semantics *cfgEsplrit* for CFG by applying our framework of semantics introduced in subsection 6.2.3|p.102. With this semantics, we introduce a *partial* piece-wise translation of *incomplete cfgRM* derivations into multiple *cfgLM* derivations. This partial translation thereby extends the recursive translation for complete derivations discussed on the previous page. The configurations in the *cfgEsplrit* semantics then represent the corresponding derivation tree obtained by the right-most derivation up to a certain point by means of pieces of left-most derivations. This additional bookkeeping information that is carried in the configurations of the *cfgEsplrit* semantics then proves useful subsequently for our overall verification task.

We now start with an example of the described decomposition of a *cfgRM* derivation into multiple pieces of *cfgLM* derivations that is implemented by the *cfgEsplrit* semantics.

Example 6.5: «Running Example for section 6.3|p.116 (Part 2/4)»

(1) Incomplete Derivation Tree $T_{1,p}$:

The derivation tree $T_{1,p}$ is a subtree of the derivation tree T_1 from block 6|p.120. It corresponds to the derivation $d_{1,p,rm}$, which is given below as a prefix of the right-most derivation $d_{1,rm}$, but not to any left-most derivation.



Corresponding Right-most Derivation $d_{1,p,rm}$ of G' :

The derivation tree $T_{1,p}$ above corresponds to the incomplete right-most derivation $d_{1,p,rm}$ that is given by the following productions π_R .

$$\pi_R = \rho_1 \rho_2 \rho_3 \rho_4 \rho_5 \rho_6 \rho_7 \rho_8 \rho_9 \quad L_{1,\square} \xrightarrow{\pi_R}_{cfgRM,G'} L_{3,\bullet,5} L_{7,\bullet,9} b d$$

Decomposition of $d_{1,p,rm}$ into Left-Most Derivations of G' :

This derivation $d_{1,p,rm}$ can be decomposed (suitably) into the five subsequent left-most derivations.

$$\begin{array}{ll}
 \pi_{L1} = \rho_1\rho_3 & L_{1,\square} \xrightarrow{\pi_{L1}}_{cfgLM,G'} L_{3,\bullet,5} L_{5,\bullet,15} L_{15,\square} \\
 \pi_{L2} = \rho_4\rho_6 & L_{5,\bullet,15} \xrightarrow{\pi_{L2}}_{cfgLM,G'} L_{7,\bullet,9} L_{9,\bullet,14} L_{14,\bullet,15} \\
 \pi_{L3} = \rho_7 & L_{9,\bullet,14} \xrightarrow{\pi_{L3}}_{cfgLM,G'} b L_{10,\bullet,14} \\
 \pi_{L4} = \rho_8 & L_{10,\bullet,14} \xrightarrow{\pi_{L4}}_{cfgLM,G'} d L_{12,\bullet,14} \\
 \pi_{L5} = \rho_9\rho_5\rho_2 & L_{12,\bullet,14} L_{14,\bullet,15} L_{15,\square} \xrightarrow{\pi_{L5}}_{cfgLM,G'} []
 \end{array}$$

The decomposition given here is constructed by means of the *cfgEsplit* semantics introduced subsequently.

For the formal definition of the *cfgEsplit* semantics, we provide an interpretation of the core locale and the locales for the marked and unmarked languages. Then, for the CFGs such as G' from Theorem 6.3|p.116, we show as a first step that every initial derivation of *cfgRM* can be translated to an initial derivation of *cfgEsplit* preserving the satisfaction of the marking condition, the marked effects, and the unmarked effects.

Well-formed *structures* of *cfgEsplit* are given by the set of all CFG that actually result from the translation using $F_{SDPDA \rightarrow CFG, Std}$.

Well-formed (initial) *configurations* of *cfgEsplit* are lists of elements of type *esplit-item* that satisfy certain well-formedness conditions. A full presentation of the configurations and the step relation of *cfgEsplit* is given in Appendix D|p.237.

Definition 6.13: «(Initial) Configurations of a CFG in *cfgEsplit*»

The *cfgEsplit* configurations of a CFG are lists of elements of type *esplit-item*, which are of the following form.

$$\begin{array}{ll}
 \langle esplit\text{-}item\text{-}elim & : \text{list of nonterminals,} \\
 esplit\text{-}item\text{-}from & : \text{at most one nonterminal,} \\
 esplit\text{-}item\text{-}ignore & : \text{list of nonterminals,} \\
 esplit\text{-}item\text{-}elim\text{-}prods & : \text{list of lists of productions,} \\
 esplit\text{-}item\text{-}prods & : \text{list of productions,} \\
 esplit\text{-}item\text{-}elem & : \text{at most one event or nonterminal,} \\
 esplit\text{-}item\text{-}to & : \text{list of nonterminals} \rangle
 \end{array}$$

Note, the components *esplit-item-from* and *esplit-item-elem* are defined using the type *option* of Isabelle.

The unique initial configuration of *cfgEsplit* for a CFG G' is given as follows.

$$\langle [], Some\ S, [], [], [], Some\ S, [] \rangle$$

Example 6.6: «Running Example for section 6.3|p.116 (Part 3/4)»

cfgEsplit Derivation $d_{1,p,es}$ Corresponding to $T_{1,p}$: The derivation has 9 steps but we only show four configurations in detail.

	<i>esplit-item-elim</i>	<i>esplit-item-from</i>	<i>esplit-item-ignore</i>	<i>esplit-item-elim-prods</i>	<i>esplit-item-prods</i>	<i>esplit-item-elim</i>	<i>esplit-item-to</i>
		$L_{1,\square}$				$L_{1,\square}$	
$\xrightarrow{\rho_1\rho_2\rho_3\rho_4\rho_5\rho_6\rho_7}$ <i>cfgEsplit, G'</i>	$L_{14,\bullet,15}L_{15,\square}$	$L_{1,\square}$ $L_{5,\bullet,15}$ $L_{9,\bullet,14}$ $L_{10,\bullet,14}$	$L_{15,\square}$ $L_{14,\bullet,15}L_{15,\square}$ $L_{14,\bullet,15}L_{15,\square}$	ρ_5, ρ_2	$\rho_1\rho_3$ $\rho_4\rho_6$ ρ_7	$L_{3,\bullet,5}$ $L_{7,\bullet,9}$ b $L_{10,\bullet,14}$	$L_{5,\bullet,15}L_{15,\square}$ $L_{9,\bullet,14}L_{14,\bullet,15}$ $L_{10,\bullet,14}$
$\xrightarrow{\rho_8}$ <i>cfgEsplit, G'</i>	$L_{14,\bullet,15}L_{15,\square}$	$L_{1,\square}$ $L_{5,\bullet,15}$ $L_{9,\bullet,14}$ $L_{10,\bullet,14}$ $L_{12,\bullet,14}$	$L_{15,\square}$ $L_{14,\bullet,15}L_{15,\square}$ $L_{14,\bullet,15}L_{15,\square}$ $L_{14,\bullet,15}L_{15,\square}$	ρ_5, ρ_2	$\rho_1\rho_3$ $\rho_4\rho_6$ ρ_7 ρ_8	$L_{3,\bullet,5}$ $L_{7,\bullet,9}$ b d $L_{12,\bullet,14}$	$L_{5,\bullet,15}L_{15,\square}$ $L_{9,\bullet,14}L_{14,\bullet,15}$ $L_{10,\bullet,14}$ $L_{12,\bullet,14}$
$\xrightarrow{\rho_9}$ <i>cfgEsplit, G'</i>	$L_{12,\bullet,14}L_{14,\bullet,15}L_{15,\square}$	$L_{1,\square}$ $L_{5,\bullet,15}$ $L_{9,\bullet,14}$ $L_{10,\bullet,14}$	$L_{15,\square}$ $L_{14,\bullet,15}L_{15,\square}$ $L_{14,\bullet,15}L_{15,\square}$	ρ_9, ρ_5, ρ_2	$\rho_1\rho_3$ $\rho_4\rho_6$ ρ_7 ρ_8	$L_{3,\bullet,5}$ $L_{7,\bullet,9}$ b d	$L_{5,\bullet,15}L_{15,\square}$ $L_{9,\bullet,14}L_{14,\bullet,15}$ $L_{10,\bullet,14}$ $L_{12,\bullet,14}$

Discussion: A more complete description of the *step relation* of *cfgEsplit* is included in Definition D.2|p.240. Also, see the next page for a short explanation of the two steps using ρ_8 and ρ_9 in the derivation $d_{1,p,es}$ on the left.

The four given configurations contain one, five, six, and five *cfgEsplit* items, respectively. Each of the five items of the last configuration corresponds to one of the five *cfgLM* derivations from Example 6.5|p.126. In general, one item may describe multiple *cfgLM* derivations as explained below. The *cfgRM* configuration that corresponds to a *cfgEsplit* configuration can be obtained by application of the operation *esplit-sig*, which returns the list of all *esplit-item-elem* components of the items contained in the configuration. For example, the last *cfgEsplit* configuration represents the *cfgRM* configuration $L_{3,\bullet,5}L_{7,\bullet,9}bd$, which is also the last configuration of the *cfgRM* derivation $d_{1,p,rm}$.

We now discuss the two kinds of *cfgLM* derivations described by *cfgEsplit* items and the relationship between two adjacent *cfgEsplit* items.

- *Generating cfgLM derivations:* The first four *cfgEsplit* items of the last configuration represent the first four *cfgLM* derivations from Example 6.5|p.126. Each of these derivations starts with the *esplit-item-from* component, applies the productions from the *esplit-item-prods* component, and generates the element in the *esplit-item-elem* component with the additional remainder given in the *esplit-item-to* component.
- *Eliminating cfgLM derivations:* The last *cfgEsplit* item of the last configuration represents the fifth *cfgLM* derivations from Example 6.5|p.126. In general, there is a *cfgLM* derivation that starts with an element from the *esplit-item-elim* component, applies the productions from the corresponding list from the *esplit-item-elim-prods* component (in the item considered, all three lists have length 1), and finally ends up with the empty string. That is, the last *cfgEsplit* item of the last configuration stores the *cfgLM* derivation from Example 6.5|p.126 in three pieces, which can be composed easily.
- *Interline Connection:* If one *cfgEsplit* item I_1 is followed by a *cfgEsplit* item I_2 in a *cfgEsplit* configuration, then the two items are connected as follows. The remainder (*esplit-item-to* I_1) @ (*esplit-item-ignore* I_1) of the item I_1 equals the list (*esplit-item-from* I_2) @ (*esplit-item-ignore* I_2) of elements available for item I_2 . In this statement, we have implicitly converted the optional elements *esplit-item-to* I_1 and *esplit-item-from* I_2 into lists of length at most one. These two components may be empty (that is, *None*) for the last item.

We now consider two *cfgEsplit* steps, which were presented in the example on the previous page, in more detail.

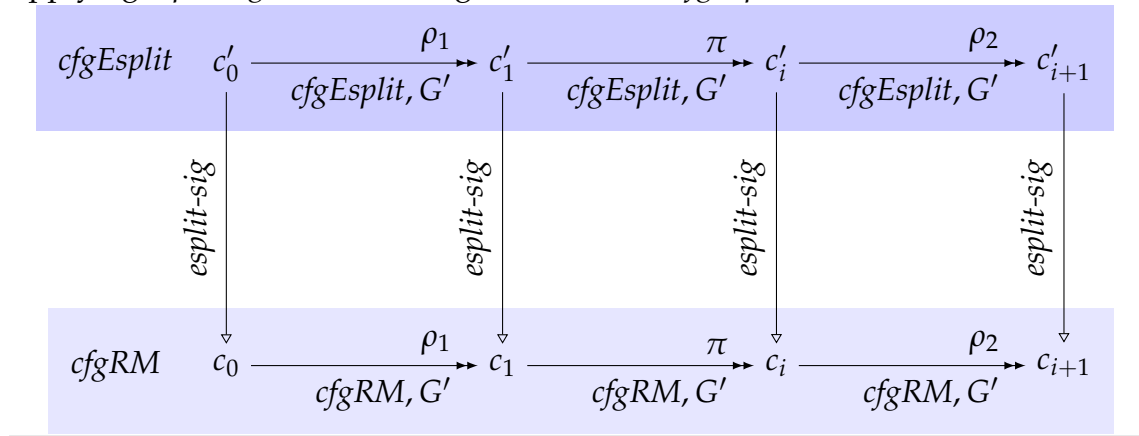
1. The production ρ_8 is used in $d_{1,p,rm}$ (see block 1|p.126) to replace $L_{10,\bullet,14}$ by $dL_{12,\bullet,14}$. In *cfgEsplit* we apply this production to an item I_1 with *esplit-item-from* $I_1 = L_{10,\bullet,14}$ and we obtain two replacement items I_2 and I_3 with *esplit-item-to* $I_2 = d$ and *esplit-item-to* $I_3 = L_{12,\bullet,14}$.
2. The production ρ_9 replaces $L_{12,\bullet,14}$ by $[]$. In *cfgEsplit* we apply this production to an item I_1 with *esplit-item-from* $I_1 = L_{12,\bullet,14}$ and we merge the eliminating derivation into the subsequent item where already two eliminating derivations for $L_{14,\bullet,15}$ and $L_{15,\square}$ are contained.

Well-formed *labels* of *cfgEsplit* are defined as for the other three semantics *cfgSTD*, *cfgRM*, and *cfgLM* introduced before. That is, the labels in the *cfgEsplit* semantics are also given by the productions of the CFG G' at hand.

As a central result, we obtain that the step relations of *cfgEsplit* and *cfgRM* are compatible w.r.t. the operation *esplit-sig* as given in the following figure. Hence, the operation *esplit-sig* can be used to translate initial derivations of *cfgEsplit* into initial derivations of *cfgRM*.

Figure 6.7: «Relationship of *cfgRM* and *cfgEsplit*»

The *cfgRM* derivation (bottom) is obtained from the *cfgEsplit* derivation (top) by applying *esplit-sig* to each configuration of the *cfgEsplit* derivation.



To define the marked and unmarked language of a given CFG G' in *cfgEsplit*, we assume an initial *cfgEsplit* derivation d . This derivation d satisfies the marking condition of *cfgEsplit* if d is finite and ends in a configuration c_{max} where *esplit-sig* c_{max} is nonterminal-free. The unique marked word of such a derivation d is then given by *esplit-sig* c_{max} . Moreover, the unique unmarked word of a configuration c that is contained in the derivation d is the maximal nonterminal-free prefix of *esplit-sig* c . Then, based on these definitions, we state that the two semantics *cfgEsplit* and *cfgRM* are equivalent w.r.t. the described marked and unmarked languages for CFG constructed according to Definition 6.8|p.117.

6.3.4. Partial Splitting Configurations

We now define the type of *cfgPsplit* configurations, which are the result of the abstraction operation *esplit-abs* that is introduced in the next subsection. This abstraction operation is applied to *cfgEsplit* configurations that are obtained from initial *cfgEsplit* derivations of the CFG G' as occurring in Theorem 6.3|p.116. Then, in subsequent subsections, we apply this abstraction operation for the verification of the satisfaction of the *cfg-LR-B* and *cfg-LR-C* properties by the CFG G' .

Similarly to the configurations of *cfgEsplit*, the configurations of *cfgPsplit* are also defined as lists of *cfgPsplit* items. Moreover, the *cfgEsplit* items and the *cfgPsplit* items are quite similar, but two of their components differ not only in their names but also in their types as follows.

- The *psplit-item-elem* component is never empty. That is, the *esplit-item-elem* component is *None* or *Some A* for some nonterminal A of G' whereas the *psplit-item-elem* component is A for some nonterminal A of G' .
- The *psplit-item-prods* component contains CFG productions and also EPDA edges. That is, the *esplit-item-prods* component is a list containing (only) productions ρ of G' whereas the *psplit-item-prods* component is a list containing productions ρ of G' and edges e of G .

The definition of (the type) of *cfgPsplit* configurations is as follows.

Definition 6.14: «cfgPsplit Configurations of a CFG»

The *cfgPsplit* configurations of a CFG are lists of elements of type *psplit-item*, which are of the following form.

<i>psplit-item-elim</i>	: list of nonterminals,
<i>psplit-item-from</i>	: at most one nonterminal,
<i>psplit-item-ignore</i>	: list of nonterminals,
<i>psplit-item-elim-prods</i>	: list of lists of productions,
<i>psplit-item-prods</i>	: list of productions or edges,
<i>psplit-item-elem</i>	: a single event or nonterminal,
<i>psplit-item-to</i>	: list of nonterminals

We do not define a custom *cfgPsplit* semantics along with the *cfgPsplit* configurations, which is sufficient as we only consider *cfgPsplit* configurations that are images of the abstraction operation *esplit-abs*.

We also do not present well-definedness constraints for the *cfgPsplit* configurations because we inherit the well-definedness constraints of the *cfgEsplit* configurations (given by the explanations in the previous subsection and Appendix D|p.237) along with our abstraction operation. That is, for a considered *cfgPsplit* configuration *esplit-abs c*, we can inherit enough properties from the *cfgEsplit* configuration c . Also note that the *cfgLM* derivations that are recorded in the *cfgEsplit* configurations are preserved to the *cfgPsplit* configurations as required for our proof.

6.3.5. Abstraction from Extended to Partial Splitting Configurations

In this subsection, we introduce the abstraction operation *esplit-abs* mentioned before. This operation abstracts a given *cfgEsplit* configurations and results in a *cfgPsplit* configuration.

The abstraction operation *esplit-abs* requires two inputs. The first input is an accessible *cfgEsplit* configuration c and the second input is a decomposition of *esplit-sig* c into two words δ_1 and δ_2 .

In the subsequently presented proof steps, we apply this abstraction operation to the last configuration of the initial *cfgRM* derivations occurring in the *cfg-LR-B* and *cfg-LR-C* properties (see Definition 6.11|p.123 and Definition 6.12|p.123). For these configurations, we use the decompositions where δ_2 is given by $\sigma_1, \sigma_2, \sigma_1$, and $\alpha_1\alpha_2$, respectively.

The fundamental goal of the abstraction operation is to remove all information from \mathcal{I} that relates to the δ_2 suffix. Stated differently, we extract the information from \mathcal{I} that represents the generation of the prefix δ_1 and drop the information that is only responsible for the generation of the suffix δ_2 .

In the subsequent subsections, we apply the operation *esplit-abs* then for well-chosen decompositions for the verification of the satisfaction of the *cfg-LR-B* and *cfg-LR-C* properties by the CFG G' .

Definition 6.15: «Abstraction Operation *esplit-abs*»

We restrict a given *cfgEsplit* configuration that essentially consists of the list \mathcal{I} of *cfgEsplit* items with *esplit-sig* $\mathcal{I} = \delta_1\delta_2$ using *esplit-abs* to obtain a *cfgPsplit* configuration as follows.

- *Step (a)*: We decompose \mathcal{I} into $\mathcal{I}_1\mathcal{I}_2$ such that *esplit-sig* $(\mathcal{I}_1\mathcal{I}) = \delta_1$.
- *Step (b)*: We drop \mathcal{I}_2 and only retain $\mathcal{I}_1\mathcal{I}$.
- *Step (c)*: We identify (*esplit-item-to* \mathcal{I}) @ (*esplit-item-ignore* \mathcal{I}) as the remainder of $\mathcal{I}_1\mathcal{I}$ that is responsible for deriving δ_2 via \mathcal{I}_2 .
- *Step (d)*: We remove this identified sequence from each item of $\mathcal{I}_1\mathcal{I}$ as follows. For the last item \mathcal{I} , we clear the *esplit-item-ignore* component, only retain the first element of the *esplit-item-to* component and, if *esplit-item-from* \mathcal{I} and *esplit-item-to* \mathcal{I} are of the form L_{q_1,X,q_2} , we also drop the target state q_2 resulting in $L_{q_1,X}$. The items from \mathcal{I}_1 are then adapted such that the interline connection (see Example 6.6|p.128) is again satisfied.
- *Step (e)*: We identify the productions ρ for which their right-hand-side is not eliminated by the subsequently applied productions, that is, the last configuration of the derivation contains a nonterminal originating from this right-hand-side.
- *Step (f)*: We replace these productions by the edges e of the SDPDA from which they were constructed according to Definition 6.8|p.117.

We now apply the described abstraction operation *esplit-abs* to *cfgEsplit* configurations in the context of our running example. We also discuss how the application of this abstraction operation can be used to derive that the two initial *cfgRM* derivations considered in the running example do not violate the *cfg-LR-B* and *cfg-LR-C* properties.

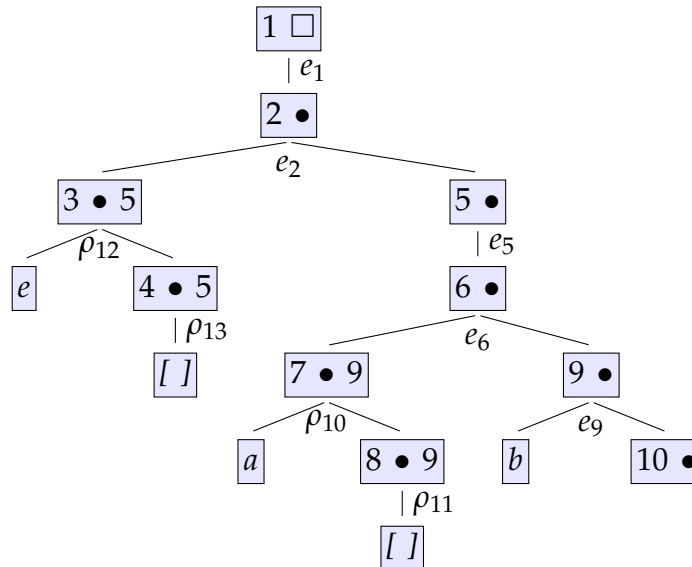
Example 6.1: «Running Example for section 6.3|p.116 (Part 4/4)»

(1) Overview of the Application of *esplit-abs*:

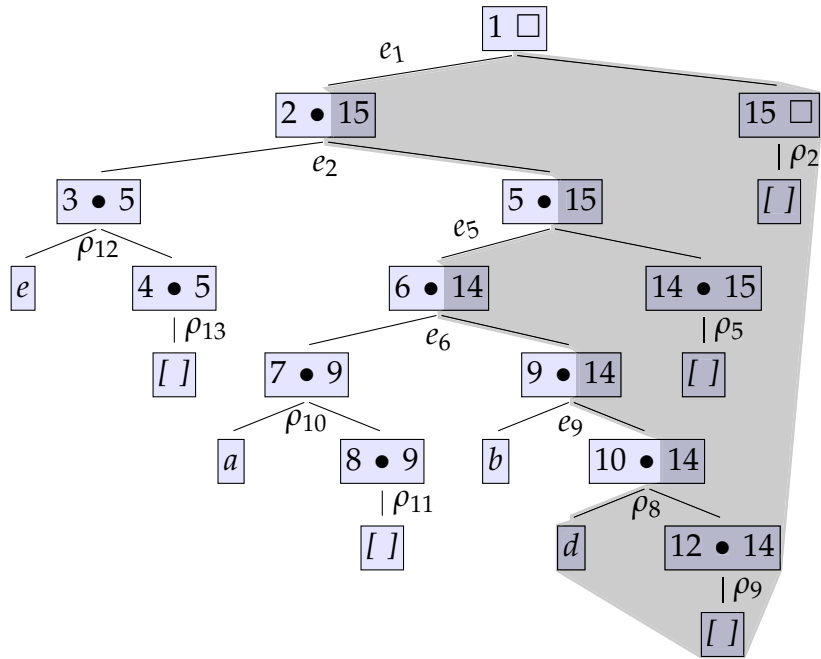
We consider the two initial *cfgRM* derivations $d_{1,rm}$ and $d_{2,rm}$ that are given in block 5|p.120. From these two derivations, we obtain the corresponding two initial *cfgEsplit* derivations $d_{1,es}$ and $d_{2,es}$ according to our description of the *cfgEsplit* semantics given in subsection 6.3.3|p.126. The last configurations of $d_{1,rm}$ and $d_{1,es}$ on the one hand and the last configurations of $d_{2,rm}$ and $d_{2,es}$ on the other hand are also visualized as derivation trees T_1 and T_2 in block 6|p.120 and block 7|p.121, respectively. We now apply *esplit-abs* to the visual representations T_1 and T_2 of the last two configurations of $d_{1,rm}$ and $d_{2,rm}$.

In this example, we have chosen the decomposition of the *esplit-sig* where we drop the information from the derivation trees that is used to generate the events following the event b in the two derivation trees. In block 3|p.134 and block 4|p.134, we present intermediate trees where the part to be removed is highlighted. However, as an important result, we obtain that the application of *esplit-abs* yields the same *cfgPsplit* configuration for both trees T_1 and T_2 . This common configuration is visualized below in block 2|p.133 as an *abstract* derivation tree. Also note that we replaced in the resulting trees the productions by edges that are associated to nodes that have a modified subtree according to the description of *esplit-abs* in Definition 6.15|p.132.

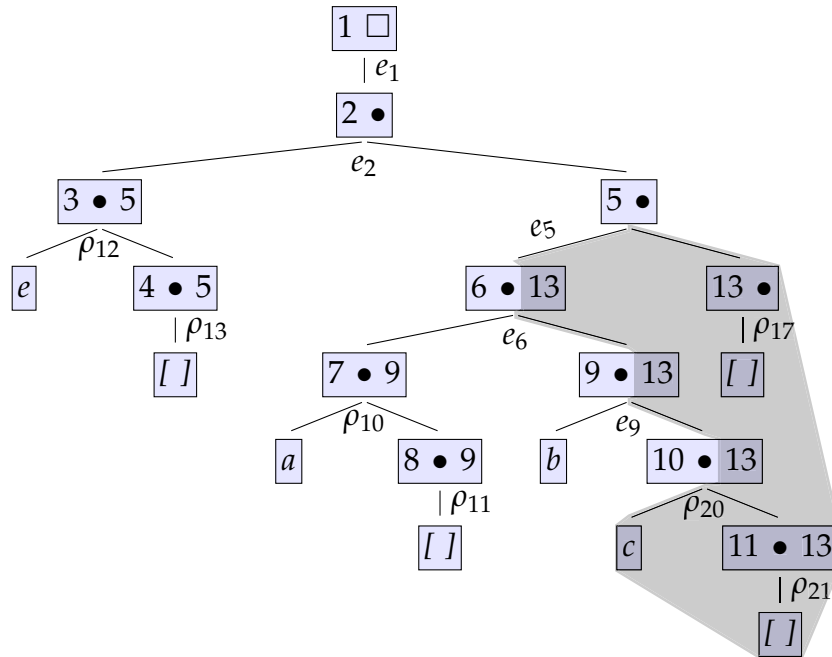
(2) Abstracted Derivation Tree T_{abs} Resulting from Either T_1 or T_2 :



(3) Visualization for Application of *esplit-abs* to T_1 from block 6|p.120:



(4) Visualization for Application of *esplit-abs* to T_2 from block 7|p.121:



(5) Application of *esplit-abs* to *cfgEsplit* Configurations (Part 1/4):

After having applied *esplit-abs* to the visualizations of *cfgEsplit* configurations in the previous blocks, we now explain the application of the abstraction operation to the following four *cfgEsplit* configurations, which are (in the upper part) the last two *cfgEsplit* configurations of $d_{1,es}$ and (in the lower part) the last two *cfgEsplit* configurations of $d_{2,es}$.

$L_{8,\bullet,9}$ $L_{12,\bullet,14}L_{14,\bullet,15}L_{15,\square}$	$L_{1,\square}$ $L_{4,\bullet,5}$ $L_{5,\bullet,15}$ $L_{9,\bullet,14}$ $L_{10,\bullet,14}$	$L_{5,\bullet,15}L_{15,\square}$ $L_{15,\square}$ $L_{14,\bullet,15}L_{15,\square}$ $L_{14,\bullet,15}L_{15,\square}$	ρ_{11} ρ_{9,ρ_5,ρ_2}	$\rho_1\rho_3\rho_{12}$ $\rho_4\rho_6\rho_{10}$ ρ_7 ρ_8	e $L_{4,\bullet,5}$ a b d	$L_{4,\bullet,5}L_{5,\bullet,15}L_{15,\square}$ $L_{8,\bullet,9}L_{9,\bullet,14}L_{14,\bullet,15}$ $L_{10,\bullet,14}$ $L_{12,\bullet,14}$
$L_{4,\bullet,5}$ $L_{8,\bullet,9}$ $L_{12,\bullet,14}L_{14,\bullet,15}L_{15,\square}$	$L_{1,\square}$ $L_{5,\bullet,15}$ $L_{9,\bullet,14}$ $L_{10,\bullet,14}$	$L_{15,\square}$ $L_{14,\bullet,15}L_{15,\square}$ $L_{14,\bullet,15}L_{15,\square}$	ρ_{13} ρ_{11} ρ_{9,ρ_5,ρ_2}	$\rho_1\rho_3\rho_{12}$ $\rho_4\rho_6\rho_{10}$ ρ_7 ρ_8	e a b d	$L_{4,\bullet,5}L_{5,\bullet,15}L_{15,\square}$ $L_{8,\bullet,9}L_{9,\bullet,14}L_{14,\bullet,15}$ $L_{10,\bullet,14}$ $L_{12,\bullet,14}$
$L_{8,\bullet,9}$ $L_{11,\bullet,13}L_{13,\bullet}$	$L_{1,\square}$ $L_{4,\bullet,5}$ $L_{5,\bullet}$ $L_{9,\bullet,13}$ $L_{10,\bullet,13}$	$L_{5,\bullet}$ $L_{13,\bullet}$ $L_{13,\bullet}$	ρ_{11} $\rho_{21}\rho_{17}$	$\rho_{14}\rho_{15}\rho_{12}$ $\rho_{16}\rho_{18}\rho_{10}$ ρ_{19} ρ_{20}	e $L_{4,\bullet,5}$ a b c	$L_{4,\bullet,5}L_{5,\bullet}$ $L_{8,\bullet,9}L_{9,\bullet,13}L_{13,\bullet}$ $L_{10,\bullet,13}$ $L_{11,\bullet,13}$
$L_{4,\bullet,5}$ $L_{8,\bullet,9}$ $L_{11,\bullet,13}L_{13,\bullet}$	$L_{1,\square}$ $L_{5,\bullet}$ $L_{9,\bullet,13}$ $L_{10,\bullet,13}$	$L_{13,\bullet}$ $L_{13,\bullet}$	ρ_{13} ρ_{11} $\rho_{21}\rho_{17}$	$\rho_{14}\rho_{15}\rho_{12}$ $\rho_{16}\rho_{18}\rho_{10}$ ρ_{19} ρ_{20}	e a b c	$L_{4,\bullet,5}L_{5,\bullet}$ $L_{8,\bullet,9}L_{9,\bullet,13}L_{13,\bullet}$ $L_{10,\bullet,13}$ $L_{11,\bullet,13}$

(6) Application of *esplit-abs* to *cfgEsplit* Configurations (Part 2/4):

In this first step (confer to steps (a) and (b) in Definition 6.15|p.132), we drop all *cfgEsplit* items beyond the *cfgEsplit* item that generates the event *b*. That is, we employ the same decomposition as explained before in block 1|p.133.

$L_{8,\bullet,9}$	$L_{1,\square}$			$\rho_1\rho_3\rho_{12}$	e	$L_{4,\bullet,5}L_{5,\bullet,15}L_{15,\square}$
	$L_{4,\bullet,5}$	$L_{5,\bullet,15}L_{15,\square}$			$L_{4,\bullet,5}$	
	$L_{5,\bullet,15}$	$L_{15,\square}$		$\rho_4\rho_6\rho_{10}$	a	$L_{8,\bullet,9}L_{9,\bullet,14}L_{14,\bullet,15}$
	$L_{9,\bullet,14}$	$L_{14,\bullet,15}L_{15,\square}$	ρ_{11}	ρ_7	b	$L_{10,\bullet,14}$
$L_{4,\bullet,5}$ $L_{8,\bullet,9}$	$L_{1,\square}$			$\rho_1\rho_3\rho_{12}$	e	$L_{4,\bullet,5}L_{5,\bullet,15}L_{15,\square}$
	$L_{5,\bullet,15}$	$L_{15,\square}$	ρ_{13}	$\rho_4\rho_6\rho_{10}$	a	$L_{8,\bullet,9}L_{9,\bullet,14}L_{14,\bullet,15}$
	$L_{9,\bullet,14}$	$L_{14,\bullet,15}L_{15,\square}$	ρ_{11}	ρ_7	b	$L_{10,\bullet,14}$
$L_{8,\bullet,9}$	$L_{1,\square}$			$\rho_{14}\rho_{15}\rho_{12}$	e	$L_{4,\bullet,5}L_{5,\bullet}$
	$L_{4,\bullet,5}$	$L_{5,\bullet}$			$L_{4,\bullet,5}$	
	$L_{5,\bullet}$			$\rho_{16}\rho_{18}\rho_{10}$	a	$L_{8,\bullet,9}L_{9,\bullet,13}L_{13,\bullet}$
	$L_{9,\bullet,13}$	$L_{13,\bullet}$	ρ_{11}	ρ_{19}	b	$L_{10,\bullet,13}$
$L_{4,\bullet,5}$ $L_{8,\bullet,9}$	$L_{1,\square}$			$\rho_{14}\rho_{15}\rho_{12}$	e	$L_{4,\bullet,5}L_{5,\bullet}$
	$L_{5,\bullet}$		ρ_{13}	$\rho_{16}\rho_{18}\rho_{10}$	a	$L_{8,\bullet,9}L_{9,\bullet,13}L_{13,\bullet}$
	$L_{9,\bullet,13}$	$L_{13,\bullet}$	ρ_{11}	ρ_{19}	b	$L_{10,\bullet,13}$

 (7) Application of *esplit-abs* to *cfgEsplit* Configurations (Part 3/4):

In this second step (confer to steps (c) and (d) in Definition 6.15|p.132), we consistently remove all but the first nonterminal remaining after generation of the event *b* by starting with the last item in each of the four configurations. Moreover, we consistently drop the target states 14 and 13 from the remaining nonterminals in the two pairs of *cfgEsplit* configuration, respectively, since these states are not relevant for the generation of the event *b* as well. Note, these removed target states are also highlighted in block 3|p.134 and block 4|p.134.

$L_{8,\bullet,9}$	$L_{1,\square}$			$\rho_1\rho_3\rho_{12}$	e	$L_{4,\bullet,5}L_{5,\bullet}$
	$L_{4,\bullet,5}$	$L_{5,\bullet}$			$L_{4,\bullet,5}$	
	$L_{5,\bullet}$			$\rho_4\rho_6\rho_{10}$	a	$L_{8,\bullet,9}L_{9,\bullet}$
	$L_{9,\bullet}$		ρ_{11}	ρ_7	b	$L_{10,\bullet}$
$L_{4,\bullet,5}$ $L_{8,\bullet,9}$	$L_{1,\square}$			$\rho_1\rho_3\rho_{12}$	e	$L_{4,\bullet,5}L_{5,\bullet,15}$
	$L_{5,\bullet}$		ρ_{13}	$\rho_4\rho_6\rho_{10}$	a	$L_{8,\bullet,9}L_{9,\bullet}$
	$L_{9,\bullet}$		ρ_{11}	ρ_7	b	$L_{10,\bullet}$
$L_{8,\bullet,9}$	$L_{1,\square}$			$\rho_{14}\rho_{15}\rho_{12}$	e	$L_{4,\bullet,5}L_{5,\bullet}$
	$L_{4,\bullet,5}$	$L_{5,\bullet}$			$L_{4,\bullet,5}$	$L_{10,\bullet}$
	$L_{5,\bullet}$			$\rho_{16}\rho_{18}\rho_{10}$	a	$L_{8,\bullet,9}L_{9,\bullet}$
	$L_{9,\bullet}$		ρ_{11}	ρ_{19}	b	$L_{10,\bullet}$
$L_{4,\bullet,5}$ $L_{8,\bullet,9}$	$L_{1,\square}$			$\rho_{14}\rho_{15}\rho_{12}$	e	$L_{4,\bullet,5}L_{5,\bullet}$
	$L_{5,\bullet}$		ρ_{13}	$\rho_{16}\rho_{18}\rho_{10}$	a	$L_{8,\bullet,9}L_{9,\bullet}$
	$L_{9,\bullet}$		ρ_{11}	ρ_{19}	b	$L_{10,\bullet}$

(8) Application of *esplit-abs* to *cfgEsplit* Configurations (Part 4/4):

In this last step (confer to steps (e) and (f) in Definition 6.15|p.132), we replace those productions by their corresponding edges that are not entirely relevant for the execution of the event b . As before, we obtain a unique result for each of the two pairs of *cfgEsplit* configurations. That is, the *cfgPsplit* configurations obtained from the last two *cfgEsplit* configurations and the two second but last *cfgEsplit* configurations coincide.

<i>psplit-item-elim</i>	<i>psplit-item-from</i>	<i>psplit-item-ignore</i>	<i>psplit-item-elim-prods</i>	<i>psplit-item-prods</i>	<i>psplit-item-elem</i>	<i>psplit-item-to</i>
$L_{8,\bullet,9}$	$L_{1,\square}$ $L_{4,\bullet,5}$ $L_{5,\bullet}$ $L_{9,\bullet}$	$L_{5,\bullet}$	ρ_{11}	$e_1e_2\rho_{12}$ $e_5e_6\rho_{10}$ ρ_7	e $L_{4,\bullet,5}$ a b	$L_{4,\bullet,5}L_{5,\bullet}$ $L_{8,\bullet,9}L_{9,\bullet}$ $L_{10,\bullet}$
$L_{4,\bullet,5}$ $L_{8,\bullet,9}$	$L_{1,\square}$ $L_{5,\bullet}$ $L_{9,\bullet}$		ρ_{13} ρ_{11}	$e_1e_2\rho_{12}$ $e_5e_6\rho_{10}$ e_9	e a b	$L_{4,\bullet,5}L_{5,\bullet,15}$ $L_{8,\bullet,9}L_{9,\bullet}$ $L_{10,\bullet}$

(9) Identifying the last applied production:

We can now deterministically identify the last productions ρ_{13} and ρ_{12} that were applied to reach the *cfgPsplit* configurations from above, respectively. We identify these productions by inspecting the *cfgPsplit* items I contained in a *cfgPsplit* configuration in the order given as follows.

- If *psplit-item-elim-prods* I is not empty, we select the first production with empty right hand side from the first element of *psplit-item-elim-prods* I .
- If *psplit-item-prods* I contains a production ρ with empty right hand side, we select the first such production ρ .
- If the item I' succeeds I and *psplit-item-elim-prods* I' and *psplit-item-prods* I' are empty, we select the last element of *psplit-item-prods* I .

Above, we identify ρ_{13} and ρ_{12} according to the first and third case, respectively, as desired. Also, for the abstract derivation tree T_{abs} from block 2|p.133 we can visually determine the productions applied in reverse order by choosing the first production visited by the right-to-left preorder traversal also used before for obtaining the *cfgRM* derivation from a derivation tree. Using this procedure, we obtain the sequence of productions $\rho_{13}\rho_{12}\rho_{11}\rho_{10}$ from T_{abs} , which are the productions that are responsible for the generation of the events preceding the event b in the derivations $d_{1,rm}$ and $d_{2,rm}$.

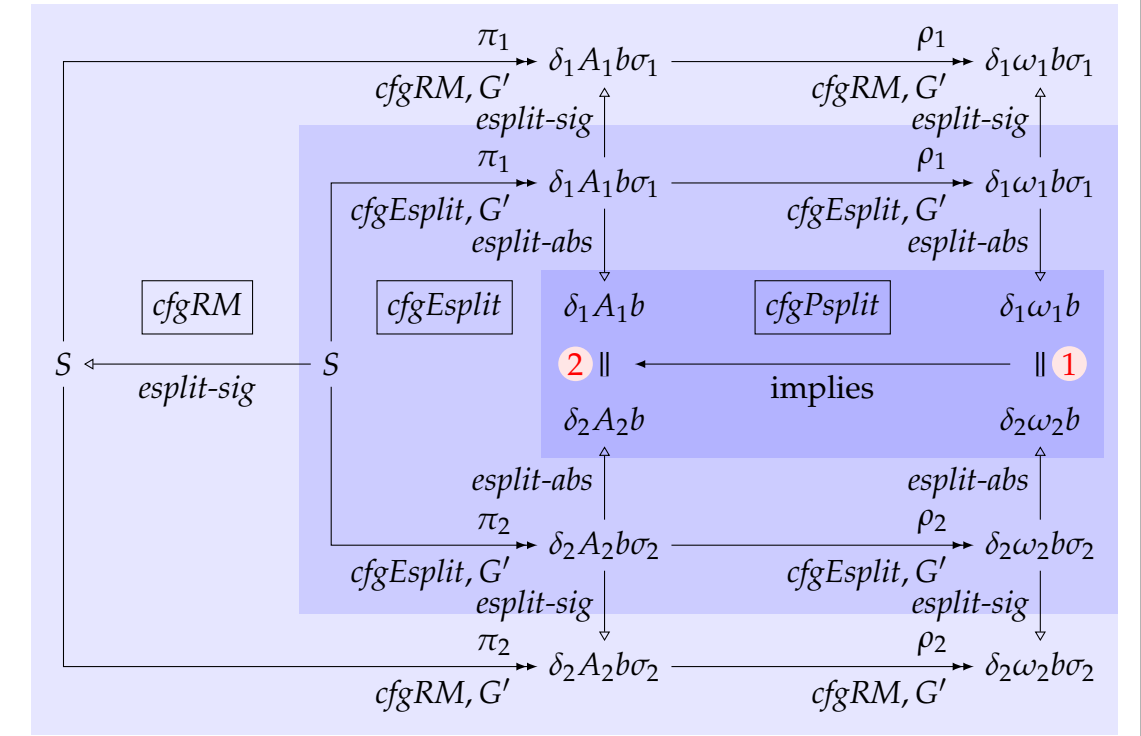
6.3.6. Verification of the *cfg-LR-B* Property

We now present the proof idea for the verification that the CFG G' obtained from $F_{SDPDA \rightarrow LR(1), Std}$ satisfies the *cfg-LR-B* property from Definition 6.11|p.123.

1. We obtain two *cfgEsplit* derivations making use of the two lists of productions $\pi_1\rho_1$ and $\pi_2\rho_2$, respectively, as explained in subsection 6.3.3|p.126.
2. We apply the operation *esplit-abs* from Definition 6.15|p.132 to the last configurations $\delta_1\omega_1b\sigma_1$ and $\delta_2\omega_2b\sigma_2$ of the two *cfgEsplit* derivations.
3. We obtain the result that the two obtained *cfgPsplit* configurations coincide (as pointed out in block 8|p.137).
4. From this single *cfgPsplit* configurations, we determine the unique last production ρ applied (as discussed in block 9|p.137).
5. By uniqueness of the last production, we obtain $\rho = \rho_1 = \rho_2$. Hence, by the step relation of *cfgRM*, we obtain $\rho_1 = \langle A_1, \omega_1 \rangle = \langle A_2, \omega_2 \rangle = \rho_2$. Moreover, using $\delta_1\omega_1 = \delta_2\omega_2$, we obtain $\delta_1 = \delta_2$.

This high-level proof idea is also visualized in the following figure where the equality **1** symbolizes the equality of the two *cfgPsplit* configurations obtained and where the implication arrow leading to the equality **2** symbolizes the step of ensuring that also the two previous configurations coincide in this sense.

Figure 6.8: «Proof Idea for the *cfg-LR-B* Property»



The equality of the two obtained *cfgPsplit* configurations and the uniqueness of the last applied production are more complex properties requiring more involved proofs when considering further details (such as the connection to *deterministic* steps of the given SDPDA in the *epdaH* semantics). However, the central achievement is that the *cfgPsplit* configurations are well-constructed in the sense that (a) they carry enough information to allow for the identification of the last production applied using the presented procedure and (b) they carry no superfluous information to allow for the equality of the *cfgPsplit* configurations obtained from the last *cfgEsplit* configurations.

6.3.7. Verification of the *cfg-LR-C* Property

We now present the proof idea for the verification that the CFG G' obtained from $F_{SDPDA \rightarrow LR(1), Std}$ satisfies the *cfg-LR-C* property from Definition 6.12|p.123. Indeed, the first steps of this proof are similar as in the previous subsection for the *cfg-LR-B* property.

The two given *cfgRM* derivations are translated into *cfgEsplit* derivations, their last configurations are abstracted into *cfgPsplit* configurations, the equality of the two *cfgPsplit* configurations is obtained, and the unique last applied production is obtained from this unique *cfgPsplit* configuration to be ρ_1 . Obviously, this production is the correct production for the first derivation. Then, by soundness of the identification procedure, we have that the production ρ_1 was also used in the last step of the second *cfgEsplit* derivation to reach the *cfgEsplit* configuration $\delta_2\omega_2b\alpha_1\alpha_2$. Moreover, we not only obtain that ρ_1 is the last production applied but also the location where this production was applied. For the first *cfgEsplit* derivation this means (and we just recall this fact here even though we do not make use of it in the proof) that ρ_1 can be applied backwards to the prefix $\delta_1\omega_1b$ of the last configuration $\delta_1\omega_1b\sigma_1$. For the second *cfgEsplit* derivation this also means that ρ_1 can be applied backwards to the prefix $\delta_2\omega_2b$ of the last configuration $\delta_2\omega_2b\alpha_1\alpha_2$. However, the last step of the second *cfgEsplit* derivation could not replace the left-hand side of ρ_1 in the prefix $\delta_2\omega_2b$ of $\delta_2\omega_2b\alpha_1A_2\alpha_2$ because in *cfgEsplit* we must replace the right-most nonterminal, which is A_2 in this case. Hence, we obtain the required contradiction.

6.3.8. Concluding Remarks

We conclude that we verified Theorem 6.3|p.116, which states that the CFG that is obtained by the conversion operation $F_{SDPDA \rightarrow CFG, Std}$ from the given SDPDA satisfies the LR(1)-CFG property. That is, the determinism of the given SDPDA is preserved to the obtained CFG. For the proof, we have decomposed the LR(1)-CFG property into the three properties verified in subsection 6.3.2|p.124, subsection 6.3.6|p.138, and subsection 6.3.7|p.139 using a simple case distinction. While the proof of the *cfg-LR-A* property is straightforward, we had to determine a suitable connection between *epdaH* derivations, *cfgLM* derivations, and *cfgRM*

derivations on the foundation of a problem specific formalization of derivation trees. These derivation trees then allowed to capture the common prefix of two derivations given in the *cfgLM* semantics and, similarly, *epdaH* semantics. As expected, our formalization of derivation trees also results in a close connection with the *cfgRM* derivations used in the LR(1)-CFG property. However, to ease of readability, we have only presented this high-level proof idea and omitted detailed proofs carried out in Isabelle supporting the claims used in our informal discussion.

Chapter 6|p.89
(Isabelle-based Formal Quality Assurance)

We discussed the theorem prover Isabelle as a tool with which we established our trustworthy proofs (available at [303]) for the theorems presented in earlier chapters such as for the abstract and concrete controller synthesis algorithms from Theorem 4.12|p.56 and Theorem 5.1|p.86. Moreover, we introduced further important foundational results and definitions contained in our Isabelle-based framework, which we used heavily throughout the verification of the mentioned results. Finally, we demonstrated our proposed approach to document involved and technical Isabelle proofs by presenting the proof of a central property of our thesis at an informal level making use of a running example. With these three steps, we underlined the importance and usefulness of employing theorem provers such as Isabelle in situations where avoiding errors is of paramount importance.

In the next chapter, we discuss the applicability of our concrete controller synthesis algorithm by presenting three case studies and by evaluating the time and space requirements of the prototype implementation of our concrete controller synthesis algorithm.

7

Application and Prototype-based Evaluation

In the previous chapter, we discussed the use of Isabelle for the verification of our concrete controller synthesis algorithm. In addition to the functional correctness, which has been verified on a clean logical foundation, we focus in this chapter on operability, constructiveness, and efficiency.

Firstly, operability of our algorithm is supported by various usage patterns for the stack that is available in a DPDA specification. Thereby, we cover a broad range of examples of use that can be combined and adapted in concrete applications. For demonstration purposes, we also discuss three concrete examples, which make use of these patterns and which correspond to three distinct general use-cases of our algorithm.

Secondly, constructiveness is demonstrated by implementing the controller synthesis algorithm in the form of our Java-based prototype CoSy for Controller Synthesis available at [304]. On the one hand, this entails the provisioning of Java code for each building block (also for those making use of, for example, choice operators). On the other hand, we substantiate our claim that intermediate values of our concrete controller synthesis algorithm are always finite. This has been verified for intermediate EPDA, Parsers, and CFG at the building block interfaces and, since our prototype succeeds for various examples, we can also derive that temporary intra-building block values are always finite in these cases.

Thirdly, from well-known results, we conclude that the required time and space of applications of our concrete controller synthesis algorithm is exponential in the size of the input DPDA (given by the synchronous product of plant and specification) due to the construction procedure of the LR(1)-Machine. While ongoing optimizations of our algorithm w.r.t. time and space requirements are called for, we show that even our prototype CoSy can be applied successfully with negligible to acceptable costs for the three examples provided where the third example is a benchmark for which the costs can be scaled up arbitrarily.

CONTENTS OF THIS CHAPTER

7.1|p.143 *Patterns for Specifications using Deterministic Pushdown Automata*

The stack-based capabilities of DPDA for recording information during a derivation can be used in various forms to describe desirable behavior in specifications. We discuss the workflow of establishing a DPDA specification as well as basic and advanced usage patterns of the stack.

7.2|p.146 *Application Domains for Discrete Event Controller Synthesis*

We briefly discuss concrete application domains where supervisory control or our concrete controller synthesis algorithm may be applied to determine a controller or to verify a given controller. An in-depth discussion of related work is postponed to section 8.1|p.164.

7.3|p.147 *Applications and Use Cases
of the Concrete Controller Synthesis Algorithm*

Three concrete problem examples from the application domain of manufacturing are provided by means of plants, specifications, and controllers given in the form of automata. These examples are connected to the patterns of specifications from section 7.1|p.143 and to three different use cases of our concrete controller synthesis algorithm.

7.4|p.154 *Prototype Realization of the Concrete Controller Synthesis Algorithm*

A Java-based prototype CoSy is presented, which entirely covers our concrete controller synthesis algorithm and thereby demonstrates its implementability. Moreover, CoSy serves as a reference implementation that can be used in back-to-back testing. Finally, CoSy is a test-bed for the development and analysis of optimizations and adaptations of our synthesis algorithm.

7.5|p.157 *Prototype-based Evaluation*

We apply our prototype CoSy from section 7.4|p.154 to the examples given in section 7.3|p.147, discuss the time and space requirements, and consider the obtained results w.r.t. the corresponding use-cases identified in section 7.3|p.147. Even at this proof-of-concept stage, suitable optimizations and the use of multithreading result in acceptable runtime when the provided memory is sufficient.

7.1. PATTERNS FOR SPECIFICATIONS USING DETERMINISTIC PUSHDOWN AUTOMATA

For an application of our concrete controller synthesis algorithm, we require a DFA plant model (uncontrolled behavior), a DPDA specification (desired controlled behavior), and a partitioning of all events into controllable and uncontrollable events. We now focus on the definition of a DPDA specification and its capabilities for expressing desired properties. Firstly, the definition of large and complex DPDA can be supported by constructing such DPDA by composition of smaller DPDA and DFA. Secondly, the use of the stack-based capabilities of DPDA for expressing desired aspects can be supported by *specification patterns*, which constitute a first step towards a high-level description language for DPDA (see also subsection 8.3.1|p.209).

—• *Composition of DFA and DPDA*

We employ the operation $F_{DPDA-DFA-Product}$ from section 5.1|p.64 to construct the synchronous composition of a DPDA and a DFA (or as a special case of two DFA). Also see subsection 8.3.1|p.209 for future work on how to increase the descriptive expressiveness of this synchronous composition operation.

As customary, see for example [71, Example 2.18, p. 83], a DFA plant can be constructed using this composition operation out of multiple smaller DFA each representing one physical or logical component of the plant or a connection of such components. Ideally, this compositional construction can be applied such that only small and simple DFA are to be defined. A specification can be constructed similarly and, as by the contribution of this thesis, one of the automata used in the course of this compositional construction may be a DPDA. This manual construction by composition is applied later in section 7.3|p.147 and the related topic of compositional controller synthesis is discussed in Par. *Synthesis Considering Horizontal Composition (Modular and Distributed Control)*|p.174. See subsection 8.3.2|p.214 for future work on how to allow that multiple automata facilitate a common stack variable using Visibly Pushdown Automata.

—• *Unbounded Stacks of DPDA*

The *unbounded* stack used by the formalism of DPDA is not available in practical applications. However, the adequacy of the unboundedness assumption depends on the *likelihood* that the stack of the generated controller requires more space than available. We also consider this issue in subsection 8.3.2|p.214. Since a stack-overflow cannot be precluded by construction in all cases, we argue that the use of the stack for specification purposes and, thereby, for controlling purposes should account for this inherent problem. That is, an upper bound should be established or estimated rates for pushing and popping elements to and from the stack may be used to obtain an estimated likelihood of out-of-memory errors. Also, the likelihood of such errors can be reduced by emptying the stack periodically during which all elements are removed explicitly or by restarting the entire controller/closed loop as demonstrated in section 7.3|p.147.

→ *Basic Specification Patterns for DPDA*

Instances of SDPDA only allow for three *simple* operations: execution of an event, pushing an element onto the stack, and popping an element from the stack. Then, DPDA allow for *compound* operations, which can be implemented by multiple simple operations; also, they allow to test for the current top-stack element. Moreover, EDPDA loosen the syntactic restriction on the edges even further and allow thereby for even more complex, compound operations.

Using such stack operations, we can obtain usage patterns such as a queue of a bounded size (see also [143, Section 8.4, pp. 116–]). For such a bounded queue, we implement the insertion of an element by a pushing edge and implement the removal/test of an element by (a) removing all elements X_1, \dots, X_n until reaching the last element of the represented queue, by (b) removing this last element, and by (c) pushing the elements X_n, \dots, X_1 back onto the stack. See also Example 7.1|p.145 for a similar implementation using an EDPDA and Example 7.4|p.151 for an application of this pattern.

The access method to a bounded prefix of the stack, as just explained for bounded queues, can be modified to determine further patterns. Firstly, we may pop elements from the stack until a certain number of elements has been popped or until a certain pattern of stack elements (for example, the end-of-stack marker) is detected. During this process, we may count or remember the elements popped using additional states (for example, we may check whether we removed an odd number of elements or whether we removed at least n elements of a certain kind). Secondly, we may then either modify the found pattern by removing further elements (such as in the bounded queue), by testing for a certain extended pattern (such as in a top operation for a bounded queue), or by pushing new elements onto the stack (as in Example 7.1|p.145). Thirdly, we may recover the stack elements stored in the first step and push them back onto the stack. We may also apply arbitrary operations to such stored data before pushing the result onto the stack such as orderings or reversals. Of course, the steps required for pushing the result onto the stack depends then on the size of the result.

Similarly to hardware components (for example, central and graphics processing units) that offer single-step implementations for complex, compound operations, we suggest the adaptation of our concrete controller synthesis algorithm as well as the development of software and hardware support for compound operations on EDPDA as just described to allow the specification to be stated more easily and to allow the controller to be more concise and efficient. In particular, for the implementation of a bounded queue above, we require a number of additional states to store the values X_n, \dots, X_1 that grows exponentially with the maximal length of the queue and the number of different stack elements that may occur in the queue. This exponential growth, as demonstrated later in Example 7.4|p.151, hampers the applicability of the bounded queue pattern and therefore suggests further developments. Similarly, the use of stack elements with an inner structure can be implemented by a flattening operation to elements without inner structure also resulting in an exponential growth.

→ *Advanced Specification Patterns for DPDA*

Besides the subsequently discussed planning of future steps, the stack may be used to log (a subset of) the executed events for later analysis, which requires sufficient actual memory as discussed before, or to store the current mode of the EDPDA in a fixed top-most part of the stack, which may drastically reduce the number of states of the EDPDA.

Planning is the fundamental purpose of using the stack in a controller. That is, the stack elements contained in the stack can be considered to be a todo-list, a plan, a schedule, a program, or a decision list resolving some nondeterministic choices to be executed in the future (also see [143, Section 3.7, p. 52]). In this view, any modification of the stack is an alteration of the plan. In particular, uncontrollable events received by the controller will result in such adaptations, but also controllable events selected nondeterministically may result in such modifications. A decisive contribution of our work is then that the plan given by the stack is always executable due to nonblockingness (assuming that the controller is marking a word once the schedule has been entirely processed). See Example 1.2|p.5 where an operator attempts to enter a valid schedule by sending events to the controller of the plant. In this case, the controller may detect and reject schedules that may not be enforced on the plant.

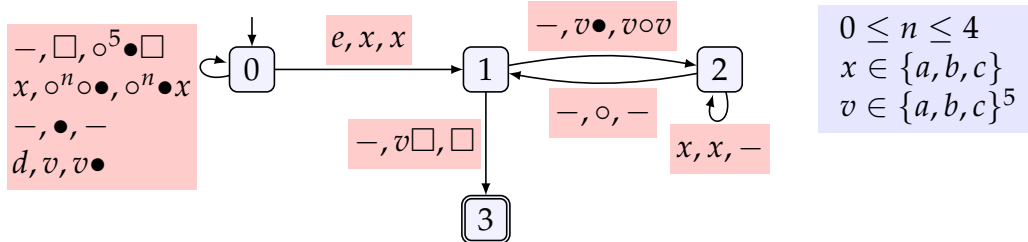
The plan to be processed may be understood to be a nested response to events such as in exception handling (see [143, Chapter 8, pp. 111–]). Also, besides remembering the events as they were received in the stack, we may also only count their number by mapping (possibly not injectively) each event to be recorded to some stack element that is pushed onto the stack upon execution of the event. Building on top of this counting mechanism, we may also store the difference between the numbers of occurrences of two events (or two groups of events where the events from one group are mapped to a common stack-element) by encoding a difference of zero by the empty stack \square , a positive difference by the stack $X^n\square$, and a negative difference by the stack $Y^n\square$ (see Example 7.2|p.147).

In this concluding example, we make use of various patterns described above.

Example 7.1: «EDPDA Specification»

Informal Specification: The words $vd^n ev^n$ are desirable where the word v is a schedule over $\{a, b, c\}$ of length 5 and where d^n is a unary encoding of how often the schedule v is to be executed after the start-up event e .

EDPDA Implementation: Note the usage of abbreviations listed on the right side, which result in numerous edges.



7.2. APPLICATION DOMAINS FOR DISCRETE EVENT CONTROLLER SYNTHESIS

The research in the domain of discrete event control theory has been concerned primarily with the synthesis of controllers for discrete event abstractions of partially physical systems (also called reactive, embedded, or cyber physical systems), which is exemplified in section 7.3|p.147. We now discuss how these algorithms also have the potential to be widely used in application development in the field of computer science at multiple levels of abstraction.

Christopher Griffin mentions already various high-level problems that can be approached using DPDA specifications. Firstly, the synthesis of network protocols is discussed by an example in [143, Chapter 4, pp. 53–]. In this example, the stack is used to store the number of possible remaining retries to be used when connection attempts fail (which are the uncontrollable events). Secondly, detection of security violations (which are the uncontrollable events) and subsequent restoration procedures are considered in [143, Section 6.5, pp. 87–]. In this example, intrusions into a filesystem are detected and logged by use of the stack and affected files are then corrected in the subsequent restoration phase. Thirdly, the handling of exceptions (which are the uncontrollable events) is discussed by an example in [143, Chapter 8, pp. 111–]. In this example, the occurring exceptions (in a robot scenario) are stored on the stack and, as expected, are handled in the nested order resulting from the use of a stack.

Similarly to the basic exception handling example, the adaptation engine proposed in [128] handles a burst of exceptions at once by (a) selecting for each exception a certain pre-given repair action (depending on attributes of faulty components), by (b) optimizing the order of these repair actions (w.r.t. their required duration and the expected reward), and by (c) applying these repair actions in the obtained order. For systems where the response time of the adaptation engine is critical, the adaptation step may consider (in a sliding window mode) only the most recent n exceptions stored in the stack at once. Moreover, if repair actions have more complex dependencies, certain restoration sequences may fail resulting in, for example, deadlocks that can be prevented by using our concrete controller synthesis algorithm.

We argue in section 8.1|p.164 that the field of submodule construction is closely related to controller synthesis (also see [45]). For example, the early work [245] on submodule construction mentions the development of parallel, distributed, and communicating systems as an important broad domain in which submodule construction techniques may prove useful when applying them to instances of suitable size in a hierarchical approach (see section 8.1|p.164 for references on developments on hierarchical controller synthesis).

Deadlocks, livelocks, and unsafe regions of the state space are to be avoided in all these applications and, hence, the main goals of controller synthesis in general and of our concrete controller synthesis in particular may apply very well. However, as mentioned in section 8.3|p.209, our concrete controller synthesis algorithm needs to be applied to concrete problem instances in the future.

7.3. APPLICATIONS AND USE CASES OF THE CONCRETE CONTROLLER SYNTHESIS ALGORITHM

We present and, *at the end of this section*, also discuss and compare three manufacturing examples using the patterns for specifications introduced before.

Example 7.2: «Automated Fabrication Scenario A [302, Section 5.1, pp. 15–]»

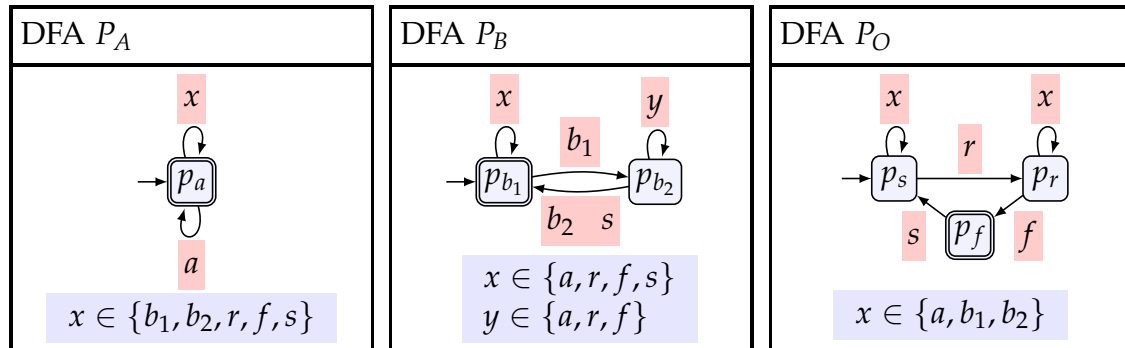
Remark: See section A|p.225 for notes on coauthorship and note that the manual controller synthesis was faulty in [302, 300] as discussed below.

Plant P: The DFA plant P is the synchronous composition of three DFA.

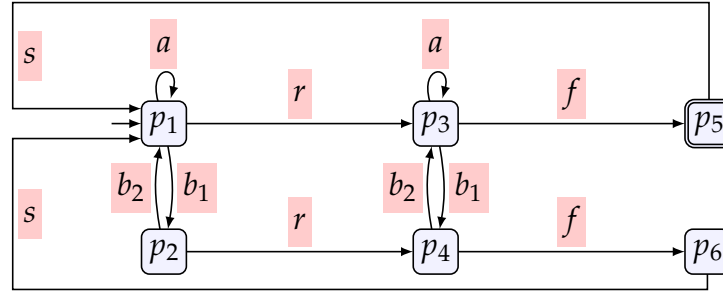
- The DFA P_A represents a machine that is producing items of kind A using event a .
- The DFA P_B represents a machine that is producing two items of kind B on every use where the events b_1 and b_2 represent the production of the first and the second item of kind B , respectively.
- The DFA P_O represents the access by the operator. The operator may trigger the two machines to produce items (events a and b_1), tell the system that the working day is coming to an end (event r), tell the system that the working day has been reached (event f), and tell the system that a new working day has started (event s).

Intuitively, another machine, which is not included in this example, will consume one item of kind A and one item of kind B to produce one item of kind C . Hence, the plant should produce just as many items of both kinds in the long run. But, since the items of kinds A and B are perishable, all such remaining items are discarded once the periodically occurring event f is executed. The desired controller should therefore act upon the mode changing event r to reduce the resulting loss as far as possible.

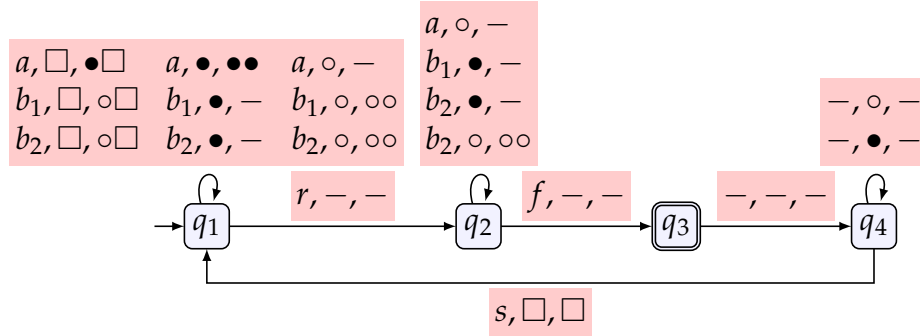
The events r , f , and s are controllable by the operator of the system, that is, they are uncontrollable to the controller to be synthesized. The event b_2 always happens after the event b_1 , and, hence it cannot be prevented by the controller either. In conclusion, we obtain the partitioning of the events Σ into controllable events $\Sigma_c = \{a, b_1\}$ and uncontrollable events $\Sigma_{uc} = \{r, f, s, b_2\}$.



The composition of these three DFA results in the following DFA plant P .



Specification S: In q_1 , the EDPDA specification uses the stack to remember the difference between the numbers of produced items of kind A and kind B . That is, in the state q_1 , a stack $\bullet^n \square$ represents the fact that $n + m$ items of kind A and m items of kind B have been produced and, vice versa, a stack $\circ^n \square$ represents the fact that m items of kind A and $n + m$ items of kind B have been produced. Once the event r is executed and the state q_2 is reached, the specification disables the events a and b_1 unless their execution reduces the depth of the stack. Further executions of a , b_1 , and b_2 then alter the stack as before. Note, if r happened between b_1 and b_2 , the next b_2 may actually increase the depth of the stack. Then, the execution of event f brings the controller to the state q_3 representing the end of the production process. Subsequently, the stack is reset in q_4 before the system may be restarted on the next working day using event s .

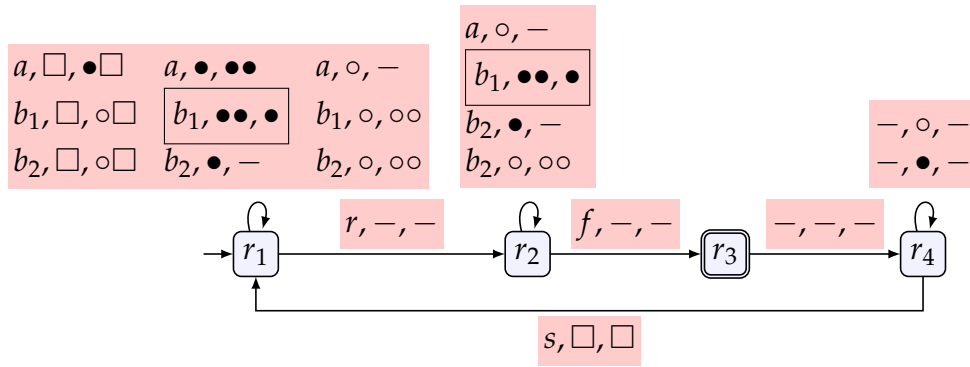


This specification is not a suitable controller because, as an example, the sequence $[a, r, b_1]$ drives P into state p_4 and S into q_2 with stack \square . Then, P can execute the uncontrollable event b_2 in p_4 that is prevented by S in q_2 . Detecting this problem might suffice in this example as it would possibly show a faulty specification. However, the specification should be enforced as given in general.

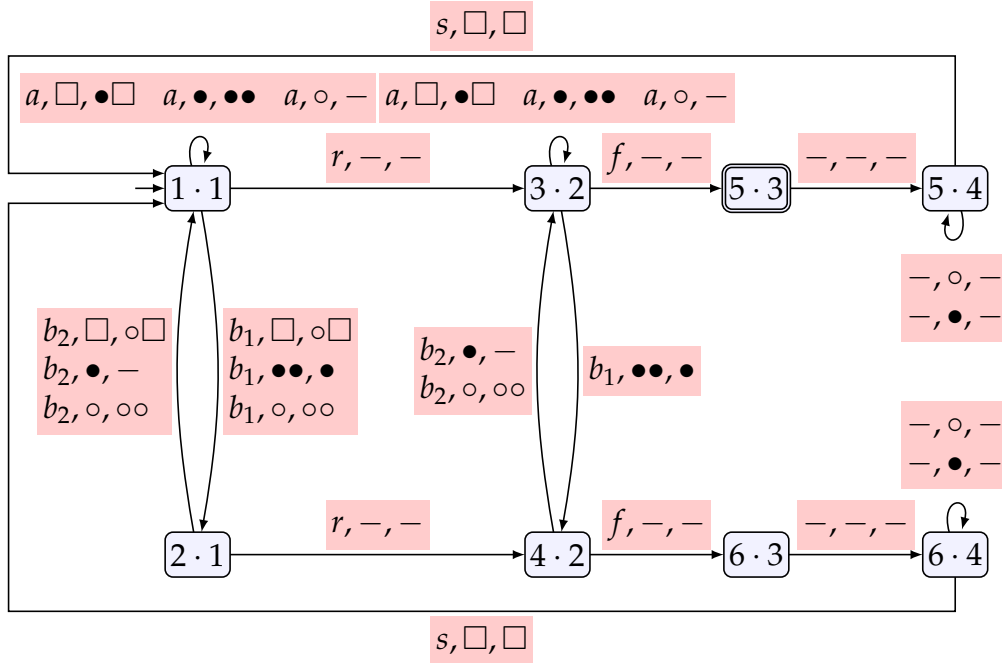
Controller C: In [302, Section 5.2, p. 24], we stated that a controllability problem occurs if and only if an odd number of events a has been executed prior to the execution of the event r . This claim was repeated in [300, Figure 11.5, p. 121]. However, preventing every word with an odd number of events a such as $w = [a, a, a, b_1, r, b_2, f]$ is not minimally restrictive. We believe that the two looping edges $b_1, \bullet, -$ at q_1 and q_2 must be changed into $b_1, \bullet\bullet, \bullet$, which means

that they are only applied if there is a further \bullet on the stack (see the framed labels in the controller below). This guarantees that the subsequently occurring b_2 is also executable.

This described modification of the specification S results in the following EDPDA, which corresponds to the least restrictive satisfactory controller from Definition 3.6|p.35. We applied our prototype implementation from section 7.4|p.154 of the concrete controller synthesis algorithm from chapter 5|p.61 to this example. The EDPDA obtained manually as discussed and automatically using CoSy differ in their structure, but, by use of our prototype, we also ensured that every unmarked/marked word of one of them (by use of an initial derivation up to a certain length) is also an unmarked/marked word of the other EDPDA solution.



Closed Loop CL: The closed loop CL obtained as the synchronous product of the plant DFA P and the controller DPDA C from above is as follows.



Note that b_1 edges are adapted here according to the controller from above.

The following example uses the bounded queue specification pattern.

Example 7.3: «Automated Fabrication Scenario B [143, Section 3.4, pp. 46–]»

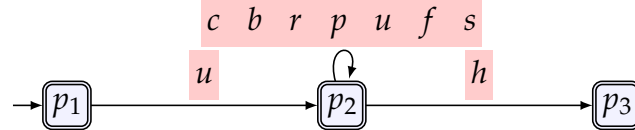
Remark: The marked and unmarked languages of the specification are contained in the corresponding languages of the plant in this example. Moreover, the concrete controller synthesis algorithm returns for this example a controller that is equivalent to the specification. That is, the provided specification is already the least restrictive satisfactory controller

Plant P: The plant produces a single widget in five steps:

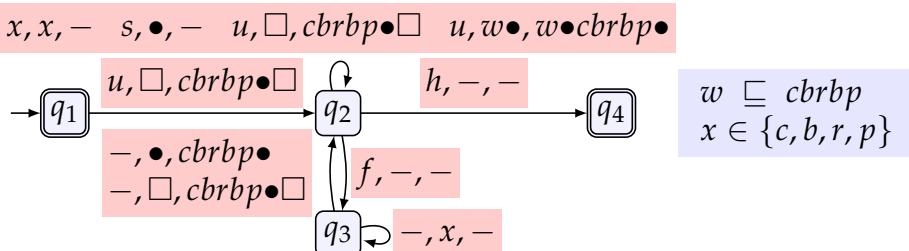
- it cuts the basic shape using event c ,
- it bends the shape using event b ,
- it rotates the product clockwise by 180° using event r ,
- it bends the shape a second time using event b , and
- it punches a round hole in the shape using event p .

The plant issues event s after successful completion of one widget. The plant receives requests from the operator using event u . The operator chooses to halt the machine using event h at any time. Also, an automatic check between two steps may detect faults during the production, which leads to the disposal of the current shape using event f .

The operator controls the events u and h and the automatic check may issue the event f . Hence, we obtain the partitioning of the events Σ into controllable events $\Sigma_c = \{c, b, r, p, s\}$ and uncontrollable events $\Sigma_{uc} = \{u, f, h\}$.



Specification S: The specification uses the stack in this example to insert a copy of the program $cbrbp$ behind the current working copy of it. Also, whenever a failure is detected, the current working copy is substituted by a fresh copy of the program. Hence, the specification requires the production of one widget for every event u .



As stated above, this specification is a satisfactory controller already.

In this last example, we use the specification to model parts of the plants behavior and also use the bounded queue specification pattern.

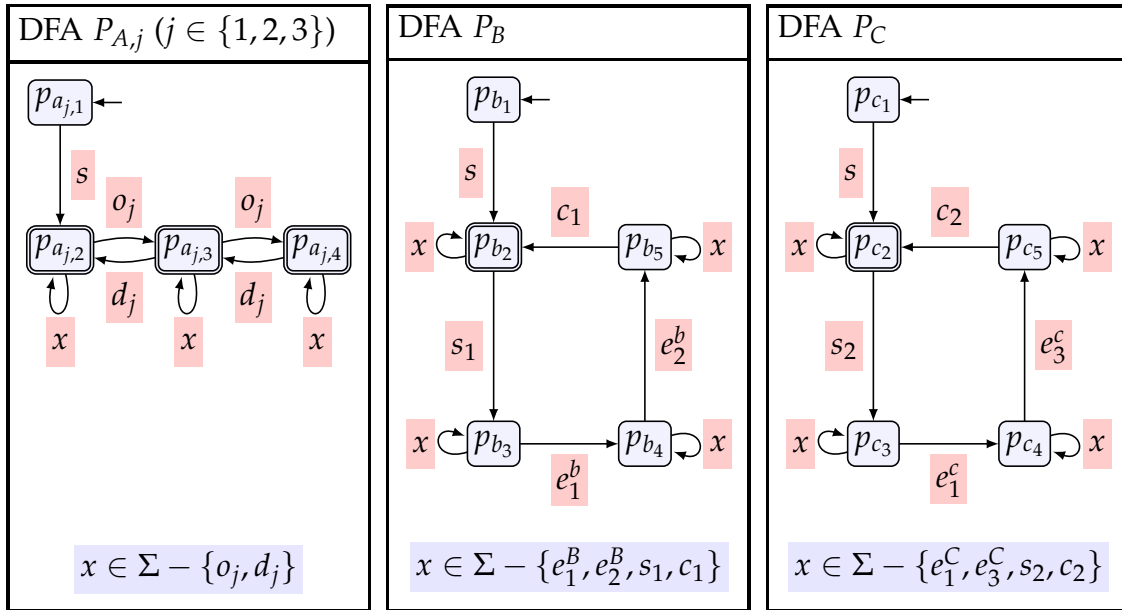
Example 7.4: «Automated Fabrication Scenario C»

Plant P: The DFA plant P is the synchronous composition of five DFA.

- The DFA $P_{A,j}$ represents for each $j \in \{1, 2, 3\}$ a machine that is ordering (concurrently up to two) items of kind j using event o_j . These items are delivered using event d_j into an input box of the following two machines.
- The DFA P_B represents a machine that uses the items e_1 and e_2 to construct an item c_2 . The event s is used as a start-up event of the global, composed machine whereas the event s_1 initiates the construction of c_1 in P_B .
- The DFA P_C is similar to P_B , but it uses and produces other items.

It is essential that the two machines P_B and P_C are only started using s_1 and s_2 , respectively, if the required items will be available to avoid deadlocks of the plant. An operator of the system may send a task using the event t_1 or t_2 to the closed loop to request the production of an item c_1 or c_2 , respectively. We ensure that no requests are lost (assuming no upper bound for pending requests) by using the stack of the EDPDA specification below.

The task events t_1 and t_2 for stating item requests, which are controllable by the operator of the system, and the delivery events d_1 , d_2 , and d_3 are not preventable by the controller. Hence, we obtain the partitioning of the events Σ into controllable events $\Sigma_c = \{s, s_1, s_2, c_1, c_2, e_1^b, e_2^b, e_1^c, e_3^c, o_1, o_2, o_3\}$ and uncontrollable events $\Sigma_{uc} = \{t_1, t_2, d_1, d_2, d_3\}$.

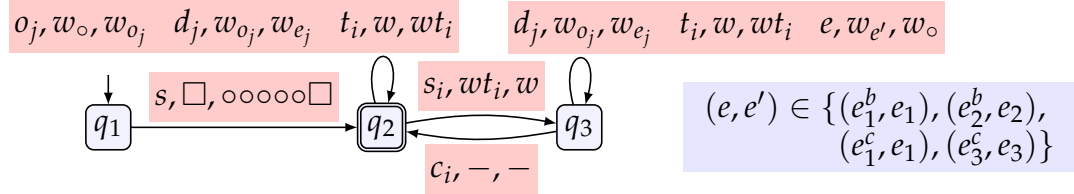


The composition of these five DFA results in the omitted DFA plant P .

Abbreviations used in the Specification S: In the EDPDA specification below, we abbreviate large sets of edges by single edges. In these edges i (for tasks) and j (for items) range over $\{1,2\}$ and $\{1,2,3\}$, respectively. Also, w denotes an ordered word of length 5 over the stack alphabet $\Gamma = \{\square, \circ, t_1, t_2, o_1, o_2, o_3, e_1, e_2, e_3\}$. Finally, in various edges, we replace in elements of such five element prefixes of the stack one element for another: if w_{γ_1} is popped and w_{γ_2} is pushed to the stack, then w_{γ_1} contains γ_1 according to the decomposition $w_{\gamma_1} = v_1 @ [\gamma_1] @ v_2$ (where γ_1 does not occur in v_2) and w_{γ_2} is the ordered version of $v_1 @ [\gamma_2] @ v_2$. Note, the employed ordering of elements reduces the number of edges.

Specification S: Once the state q_2 has been reached in S using the first edge, the stack is partitioned into two components. Firstly, the first five elements of the stack correspond to the input box of the plant, that is, each element is either \circ (an empty compartment), o_j (an empty compartment for which an element e_j has been ordered), or e_j (a compartment filled with an e_j element). Secondly, the stack below this five element prefix is used to store the requests for production, that is, an unbounded stack, which is using elements t_i and which is terminated by the \square symbol. The usage of a stack instead of a queue is reasonable here as usual if the average insertion rate is far below the average removal rate, that is, if the stack is empty at sufficient intervals. The specification uses the event s to establish this invariant structure of the stack.

In q_2 , the controller may actively order further items using the o_j events staying in q_2 or it may decide to start the production of the top-most task reaching q_3 . In q_3 , the controller uses the elements in the input compartment of the plant to produce the item (the selection of items is governed by the plant P above) and, finally, it returns back to q_2 once the production has finished. The controller can passively recognize in q_2 and q_3 the delivery of ordered items and further incoming requests.



The controller must not order a wrong selection of items in q_2 to ensure that the production executed in q_3 can always succeed without blocking. Also, since new tasks are inserted onto the top of the stack (below of the five element prefix) the controller cannot decide on its orders based on the task currently on top of the stack. For example, the EDPDA S is not a suitable controller because, as an example, the sequence $[s, t_1, s_1]$ drives S into state q_3 with stack $○○○○□$ from which the marking state q_2 is not reachable anymore (the items e_1 and e_2 have not been ordered before starting the execution of t_1 using s_1).

Note, if e_1^b, e_1^c, e_2^b , and e_3^c were chosen to be uncontrollable, that is, the machine should not be stalled by the controller in its production, then the described blocking situation is in fact a controllability problem as well.

Controller C: Here, we assume that the current five element prefix of the stack is w . Also, we define two functions $cost$ and $free$ on such prefixes. Firstly, $cost(j, w) = 0$ if w contains e_j or o_j and 1 otherwise. Secondly, $free(w)$ equals the number of occurrences of o in w . Then, the expected least restrictive satisfactory controller C permits

- the event s_1 if and only if $cost(1, w) = cost(2, w) = 0$,
- the event s_2 if and only if $cost(1, w) = cost(3, w) = 0$,
- the event o_1 if and only if $free(w) > cost(2, w) + cost(3, w)$,
- the event o_2 if and only if $free(w) > cost(1, w) + cost(3, w)$, and
- the event o_3 if and only if $free(w) > cost(1, w) + cost(2, w)$.

That is, s_1 and s_2 are only allowed once their requirements are met (or at least ordered) and ordering of an item o_j is only allowed if all other missing items that may be required can still be ordered subsequently.

In the three examples above, we applied our concrete controller synthesis algorithm and some of the DPDA specification capabilities from section 7.1|p.143 in different ways as discussed and summarized subsequently. We used our concrete controller synthesis algorithm for three different use cases in the examples (see also Figure 1.3|p.8).

Use-case 1: The *validity* of a given specification has been analyzed in Example 7.2|p.147. In particular, the provided specification prevented some step that (potentially) should not have been prevented according to the informal description. User assistance is provided by presenting the detected controllability problems to the user. For example, a controllability problem can be represented in the form of a reachable configuration of the closed loop of the current controller candidate and the plant where the controller prevents an uncontrollable event that can be executed by the plant. The user may then inspect the controllability problems and determine whether the controllability problem is the result of a faulty specification. Also, states and edges of the specification that are not accessible in the closed loop indicate invalid specifications or overspecification.

Use-case 2: The *soundness* of a controller (given as a specification) has been analyzed in Example 7.3|p.150. If the specification is given by a domain expert who provides a very detailed description of all parts, then the specification may already turn out to be a satisfactory controller and no controllability or blocking problems are to be removed. Hence, in this use case the EDPDA specification is already a description of the controller that was designed having the plant description in mind. Of course, if a large plant description makes the definition of such a controller-specification a hard task, the next use case is required.

Use-case 3: The *least restrictive satisfactory controller* for a given specification has been computed in Example 7.4|p.151. In this case, the specification provided still requires many decisions to be made to ensure controllability. That is, events are to be prevented in certain situations to reduce local options that eventually lead to controllability problems.

Also, a user may seamlessly combine these use cases in one workflow as follows. Firstly, the user may provide an initial controller candidate or specification in the form of an EDPDA, which contains important specification aspects such as in the examples above. Secondly, the user may apply the concrete controller synthesis algorithm to detect and then inspect controllability problems resulting from this specification. Consequently, the user may refine the specification or correct errors discovered using this methodology. Finally, once no further refinements or corrections of the specification are to be applied, the user may employ the concrete controller synthesis algorithm to generate the least restrictive satisfactory controller contained in the specification obtained so far.

The examples above demonstrated the discussed use cases and made use of various advanced specification patterns discussed in Par. *Advanced Specification Patterns for DPDA*|p.145: the compositional construction of the plant is used in Example 7.2|p.147 and Example 7.4|p.151, the periodic removal of all stack-elements is used in Example 7.2|p.147, the counting of events from two groups is used in Example 7.2|p.147, the stack is used to describe future tasks/programs to be executed in Example 7.3|p.150 and Example 7.4|p.151, and the stack is used to describe a finite memory used by the plant in Example 7.4|p.151.

The three examples given in this section provide an overview on the capabilities of EDPDA for the specification of admissible behavior and for use cases of our concrete controller synthesis algorithm.

7.4. PROTOTYPE REALIZATION OF THE CONCRETE CONTROLLER SYNTHESIS ALGORITHM

We now discuss our implementation of our concrete controller synthesis algorithm in the form of our Java-based prototype CoSy available at [304], some relevant aspects of this implementation, and some conclusions we have been able to draw already from the implementation process. Moreover, in the next section, we provide some results from applying this implementation to the examples from the previous section also resulting in further observations on the concrete controller synthesis algorithm in general and our implementation in particular.

We formalized and verified our concrete controller synthesis algorithm in Isabelle [265] using higher-order logic by defining its building blocks (as presented in chapter 5|p.61) in Isabelle and by verifying Theorem 5.1|p.86. However, the existence of a suitable implementation of our algorithm in the form of a computer program is not immediate because Isabelle with higher-order logic includes the axiom of choice in the sense of the Ernst Zermelo-Abraham Fraenkel set theory.

That is, Isabelle supports the axiom of choice by means of the ϵ -operator of David Hilbert and the closely related ι -operator. Both operators have a similar purpose and can be used to non-constructively select elements satisfying a property (see also section B.1|p.228). Since there is no decision procedure implementing these operators for arbitrary higher-order logic properties P , Isabelle definitions are not constructive per se. This is a key difference compared to the theorem prover Coq [172] where the axiom of choice is not given and all definitions can be converted to Meta Language (ML) code. In comparison to Coq, the support of automatic code generation techniques in Isabelle is, as of now, far from optimal for various reasons. Firstly, the documentation does not adequately describe the process and possible pitfalls, which complicates the process for users not familiar with the actual implementation of the involved procedures. Secondly, the errors produced during code generation are not suitable to guide the user to the actual problem. Thirdly, the usage of code generation requires to paraphrase constructions *significantly* to make use of a limited set of syntax and operations for which implementations are available. Hence, the usage of Isabelle-based code generation techniques is left for future work (see subsection 8.3.1|p.209) when the support in Isabelle has reached an adequate maturity. We have still chosen Isabelle over Coq, despite the lack of code generation support of Isabelle, as the local and global Isabelle community seemed to be more active implying better future availability of tool-support in the form of maintained theorem provers.

Besides the usage of the two choice operators in operations, further problems may preclude executability or code-generation: iterative computations may not terminate in general, intermediate values in computations may require infinite space, and conditionals where the conditions require unavailable decision procedures. The inspection of the definitions of the constructions of our concrete controller synthesis algorithm is a suitable first step towards deciding whether the constructions contain such problems. As a second step, the constructions should be implemented and tested to reveal further unforeseen complications; obviously, infinite intermediate values and nontermination may not be excluded via testing, but choice operators and conditionals decision procedures must be implemented. We believe that, during the implementation process, we determined such decision procedures and that intermediate states are always finite. Hence, we are confident that the concrete controller synthesis algorithm is constructive.

We implemented the concrete controller synthesis algorithm along the formal Isabelle-based definitions in Java using 271 classes (73 classes for building blocks, 35 classes for multithread implementations of building blocks, and 68 classes for datatypes such as EPDA, Parsers, and CFG) with 29 619 lines (of which 10 245 are pairwise distinct) and 89 test classes. Structurally, we implemented the (nested) building blocks as given by the formal definitions to prevent fundamental errors resulting from reorganization of building blocks, to make sure that bugs in the implementation can be located easily by comparing the actual behavior of the implementation with the expected behavior given by the formal definitions, to ease understandability for reviewers of both representations (the algorithms

and their realization in code), to ease maintainability of the code due to the resulting small sizes of methods, and to ease adaptability of the code once formal constructions are adapted in the future to accomodate for optimizations.

While the concrete controller synthesis algorithm has been verified using Isabelle, the Java-based CoSy prototype may contain bugs as mentioned in section 6.1|p.91. For quality assurance, we used code coverage techniques to ensure that our module tests and integration tests (see next section) cover the relevant parts of our implementation. We have also applied our prototype to the three examples from the previous section. The results of these applications are discussed in the next section.

In our CoSy prototype, we implemented additional optimizations that can be enabled *optionally*. For example, in formal definitions, there is no need to ensure that an expression is not evaluated multiple times. Rather, the explicit usage of a cache for such already evaluated expressions would drastically complicate proofs. Hence, we believe that it is reasonable to proceed by: formally verifying the algorithm, paraphrasing the algorithm for optimization and implementation purposes (this optional intermediate step is left for future work (see subsection 8.3.1|p.209) for some but not for all our optimizations), and implementing the algorithm in a programming language. Aside from optimizations that prevent duplicate computations, we have also removed inaccessible elements from EPDA, Parsers, and CFG between building blocks that are applied in succession to simplify subsequent operations. We conclude that even our well-tested optimizations that are implemented in our prototype should be formally verified in Isabelle in the future (see subsection 8.3.1|p.209).

The use of multithreading proved highly useful in CoSy. On the one hand, the fixed-point computation has at the highest level of building blocks only sequentially applied steps in which no parallelism can be used. On the other hand, many lower level building blocks allow for massive parallel execution (for example, when the LR(1)-Machine is constructed for a CFG). The implementation of the prototype allowed us to determine its bottle necks, which are the conversion of SDPDA into LR(1)-CFG and the construction of the LR(1)-Machine from the LR(1)-CFG. To resolve the first bottle neck, we developed the optimization in subsection 8.2.1|p.188 (and used parallelization for the formally verified optimization from section 5.2|p.65 for the SDPDA state space approximation). To resolve the second bottle neck, we used parallelization where each thread in the parallel breadth-first-construction is then concerned with a certain subset of the pending states. The use of parallel execution thereby significantly reduced computation times as indicated by our subsequent tool-based analysis.

Finally, Stefan Jacobi [178] and Ramon Barakat developed a C++ prototype as a plugin of the libFAUDES [32] tool. However, this plugin is not sufficiently efficient (it employs neither optimizations nor multithreading) and not trustworthy (it was not tested thoroughly and is does not implement in all cases the formally verified constructions). The adaptation of this plugin to CoSy remains for the future.

7.5. PROTOTYPE-BASED EVALUATION

In this section, we argue that the worst case complexity of our concrete controller synthesis algorithm w.r.t. time and space is exponential (only considering inputs where the algorithm terminates). Consequently, we have to argue for the usefulness of our algorithm by discussing the required time and space for applications to realistic examples. For this, we apply our Java-based prototype to the concrete examples from section 7.3|p.147 and consider the resulting runtime and the resulting memory consumption.

The worst case complexity of our concrete controller synthesis algorithm is exponential because the operation $F_{LR(k)\text{-Parser}}$ for constructing an LR(1)-Parser from an LR(1)-CFG may result in a LR(1)-Parser of a size that is exponential in the size of the LR(1)-CFG. Stated differently, LR(1)-CFGs are exponentially more succinct compared to deterministic LR(1)-Parsers [325, 127, 126, 157, 347, 217]. In [110, p. 128] Jay Early attributes the following sequence (indexed by n) of LR(o)-CFG to John Reynolds, which result in exponentially larger LR(o)-Parsers.

Example 7.5: «Exponential Worst Case Cost for $F_{LR(k)\text{-Parser}}$ »

Sequence of LR(o)-CFG: For each natural number $n > 0$, we define an LR(o)-CFG with initial nonterminal R using six patterns of productions.

$$\begin{array}{l|l} 1 \leq i \leq n & : R \longrightarrow A_i \\ 1 \leq i \neq j \leq n & : A_i \longrightarrow c_j A_i \\ 1 \leq i \leq n & : A_i \longrightarrow d_i \end{array} \quad \begin{array}{l} 1 \leq i \leq n : A_i \longrightarrow c_i B_i \\ 1 \leq i, j \leq n : B_i \longrightarrow c_j B_i \\ 1 \leq i \leq n : B_i \longrightarrow d_i \end{array}$$

Discussion: For this sequence, there is a constant c such that the LR(o)-Machine and the LR(o)-Parser constructed using the operations $F_{LR(k)\text{-Machine}}$ and $F_{LR(k)\text{-Parser}}$ have more than $2^{c \times n}$ states and rules, respectively. The reason for this is that the LR(o)-Machine differentiates (by having a state for each case) between exponentially many alternative sequences of executable events.

However, in our considered examples that were inspired by actual control problems, we never observed such LR(1)-CFG resulting in an exponential blowup. Also, while the Parser construction above is used in each iteration, the unfolding computed by this conversion is not refined by its applications in later iterations except for the parts that have been adapted by removing some controllability problems. Nevertheless, the computational bottle neck is the construction of the LR(1)-Machine as of now according to our prototype-based analysis even when constructing the LR(1)-Machine using multithreading.

The conversion from SDPDA into the LR(1)-CFG produces a cubic number of productions according to the operation $F_{SDPDA \rightarrow LR(1), Std}$. Also our optimized operations $F_{SDPDA \rightarrow LR(1), Opt}$ (see section 5.2.1|p.71) and $F_{SDPDA \rightarrow LR(1), Rec}$ (see subsection 8.2.1|p.188) still have this worst case complexity. When using the optimized conversion procedure $F_{SDPDA \rightarrow LR(1), Opt}$, the vast amount of computation time is used for computing the employed state space approximation.

We used two machines A and B given in the following table for our subsequent prototype-based evaluation.

Table 7.1: «Machines A and B used for the Prototype-based Evaluation»

Machine A	
Random-access Memory	256 GiB DDR4
Central Processing Unit	$2 \times \text{E5-2643 @ 3.4 GHz} \times 6 \text{ cores} \times 2 \text{ threads}$
Hard Disk	600 GiB on 10 000 RPM RAID 1
Swap Partition	0 GiB
Focus	Central Processing Unit
Machine B	
Random-access Memory	384 GiB DDR3
Central Processing Unit	$2 \times \text{E5-2630 @ 2.3 GHz} \times 6 \text{ cores} \times 2 \text{ threads}$
Hard Disk	1 TiB on 7200 RPM RAID 1
Swap Partition	0 GiB
Focus	Random-access Memory

For the purpose of the evaluation, we conducted several experiments for each of the three examples from section 7.3. For each experiment, we have applied our prototype 10 times using the maximum number of 24 running worker threads and rounded the measured usage of the memory (“Maximum resident set size”), wall time (“Elapsed (wall clock) time”)¹, and cpu (“Percent of CPU this job got”) using the Linux tool `time` [139]. The values subsequently given in tables are the average values over these 10 executions. In these experiments, we used all additional optimizations implemented to reduce memory consumption and computation time. We omit numbers for dry runs measuring the build up and tear down operations as they required at most one second in each run. Also, we make use of the following parameters in all our experiments.

- \mathcal{P}_M designates the name of the machine used.
- \mathcal{P}_{AA} is the maximal length of stack-approximations used to determine inaccessible states and edges in intermediate EPDA. This state space approximation was discussed in Par. *Efficiency of the Conversion Procedure* | p.71 for the purpose of optimizing the conversion from SDPDA into LR(1)-CFG, but we use our more efficient operation $F_{SDPDA \rightarrow LR(1), Rec}$ from subsection 8.2.1 | p.188 in our experiments.
- \mathcal{P}_{MEM} is the maximal (heap) memory size in GiB the experiment may take. We used the Java parameter `-Xmx` to restrict the size of the heap to this value. We used different maximal memory values to determine the optimal set up in which the garbage collector resulted in the minimal runtime overhead.

¹Hence, the just-in-time compilation of hotspot Java methods to native code at runtime (executed by the JVM) does not corrupt our results.

Figure 7.1: «Prototype-based Evaluation for Examples»

We consider Example 7.2|p.147, Example 7.3|p.150, and different variations of Example 7.4|p.151 and report on the set up that resulted in the optimal resource consumption.

Evaluation for Example 7.2|p.147: Both machines achieved similar results and required for $\mathcal{P}_{AA} = 1$ less than 746 MiB of memory and less than 14 s. The resulting controller had 26 439 states, 66 615 edges, and 2322 stack symbols.

Parameters			Performance Results		
\mathcal{P}_M A/B	\mathcal{P}_{AA} nat	\mathcal{P}_{MEM} GiB	memory GiB	wall time min: s	cpu %
A	1	1	1	0:12	433
B	1	1	1	0:14	381

Evaluation for Example 7.3|p.150: Both machines achieved similar results and required for $\mathcal{P}_{AA} = 1$ less than 301 MiB of memory and less than 2 s. The resulting controller had 1180 states, 3079 edges, and 121 stack symbols.

Parameters			Performance Results		
\mathcal{P}_M A/B	\mathcal{P}_{AA} nat	\mathcal{P}_{MEM} GiB	memory GiB	wall time min: s	cpu %
A	1	1	0	0: 1	499
B	1	1	0	0: 2	439

Evaluation for Example 7.4|p.151: This example is designed as a scalable benchmark, which means that we can derive different variations of it using certain parameters and, therefore, we can derive variations where the synthesis requires a relevant amount of resources. The problem from Example 7.4|p.151 is given by variation \mathcal{V}_1 and we obtain two simpler variations \mathcal{V}_2 and \mathcal{V}_3 by using different values for five parameters v_1 – v_5 . The mapping from variations to values of parameters (with their description) is given in the following table.

Parameter	Description	\mathcal{V}_1	\mathcal{V}_2	\mathcal{V}_3
v_1	Number of common elements required by the two machines P_B and P_C	1	1	0
v_2	Number of elements only required by machine P_B	1	1	1
v_3	Number of elements only required by machine P_C	1	1	1
v_4	Length of the container storing elements	5	3	2
v_5	Number of concurrent orders per element	2	1	1

We consider each of the three variations now separately.

Evaluation for Example 7.4|p.151: Variation \mathcal{V}_3 : Both machines required for $\mathcal{P}_{AA} = 1$ less than 4 GiB of memory and less than 5 min where machine A was about 25 % faster than machine B. The resulting controller had 122 080 states, 636 122 edges, and 3429 stack symbols.

Parameters			Performance Results		
\mathcal{P}_M A/B	\mathcal{P}_{AA} nat	\mathcal{P}_{MEM} GiB	memory GiB	wall time min: s	cpu %
A	1	4	4	3:37	616
B	1	4	4	4:51	588

Evaluation for Example 7.4|p.151: Variation \mathcal{V}_2 : Both machines required for $\mathcal{P}_{AA} = 1$ about 200 GiB of memory and less than 16 h where machine A was about 18 % faster than machine B. The resulting controller had 2 139 865 states, 17 316 332 edges, and 21 734 stack symbols.

Parameters			Performance Results		
\mathcal{P}_M A/B	\mathcal{P}_{AA} nat	\mathcal{P}_{MEM} GiB	memory GiB	wall time h:min	cpu %
A	1	230	198	12:31	949
B	1	256	202	15:21	895

Evaluation for Example 7.4|p.151: Variation \mathcal{V}_1 : Both machines do not terminate within 24 h where the computation slows down due to excessive garbage collection when the maximal amount of memory has been allocated.

While the specification patterns used in Example 7.2|p.147 and Example 7.3|p.150 result in negligible costs for synthesis, we conclude from the scalable benchmark obtained along the lines of Example 7.4|p.151 that the usage of a finite prefix of the stack (using the random-access or queue pattern as discussed in Par. *Basic Specification Patterns for DPDA*|p.144) is problematic. In the variations described in the figure above, *exponentially* growing controllers are obtained because the number of possible values of this prefix grows exponentially (in \mathcal{V}_1 at most $5^7 = 78\,125$ different prefixes are considered, due to the length 5 of the prefix and the 7 different values $\{\circ, o_1, o_2, o_3, e_1, e_2, e_3\}$ contained in the prefix). Each of these prefixes is considered separately in the resulting automaton and, moreover, many of these values are not removed by the synthesis procedure. The large controllers then require a huge amount of time and memory to be analyzed. This example demonstrates that controller synthesis can be problematic while manual construction can be supported and manual constructions can be verified along the lines of the use cases presented at the end of section 7.3|p.147.

The size of the resulting controllers also demonstrates that static memory requirements for implementations of the synthesized controllers potentially pose problems in settings where the hardware is equipped with limited memory. Also, the size of the resulting controller was discussed in Par. *Additional Property of Minimality for DPDA*|p.39 and is considered as related and future work in Par. *Further Controller Characteristics*|p.166, Par. *Inputs of the Concrete Controller Synthesis Algorithm*|p.209, and Par. *Operations of the Concrete Controller Synthesis Algorithm*|p.210.

The resource requirements of our prototype CoSy show that further improvements are, albeit sufficient performance for the first two examples, required as future work. The artificial benchmark example can be a guide for further improvements, which may even be specific to the characteristics of the machine on which synthesis is executed in the future, and to reveal problems of the approach, which are often specific to certain kinds of inputs (such as to the random-access usage of a prefix of the stack). Moreover, this benchmark can be used to compare competing tools and approaches in the future. Finally, we point out that our second controller synthesis algorithm from subsection 8.2.2|p.193 has not been evaluated here and may improve performance drastically as it does not rely on the costly repeated conversion between LR(1)-CFG and DPDA controller candidates.

Chapter 7|p.141

(Application and Prototype-based Evaluation)

In this chapter, we discussed the operability of our concrete controller synthesis algorithm by detailing on patterns that can be used when determining DPDA specifications. Also, we introduced our Java-based prototype CoSy available at [304] in which our algorithm is implemented. This prototype demonstrates the overall constructiveness of the algorithm and is also used to evaluate the time and space requirements for three concrete examples provided. Beyond standard controller synthesis, we also discussed two further general use cases that can support the process of determining a suitable controller for a given plant. The first additional use case supports the tool-based support for the validation and manual adaptation of a given specifications and the second additional use case supports the verification of a manually provided controller. This chapter thereby extends the results on the functional correctness established in previous chapters.

In the next chapter, we discuss related, ongoing, and future work.

8

Related, Ongoing, and Future Work

We survey related work focusing on alternative approaches to similar problems, foundations of our contributions, and potential fields for applications of our results. Moreover, we present three approaches that have not yet been formalized in Isabelle but that are promising candidates on extending the contributions of this thesis. Finally, we elaborate, based on open problems indicated throughout the thesis and the presented related work, on various possible future extensions and applications of our results.

CONTENTS OF THIS CHAPTER

8.1|p.164 *Related Work*

We discuss variations and extensions of the supervisory control problem from similar domains, we detail on the faulty controller synthesis algorithm of Christopher Griffin for DFA plants and DPDA specifications, and we compare available tools for controller synthesis and available relevant Isabelle-frameworks.

8.2|p.188 *Ongoing Work*

We present an operation for converting DPDA into LR(1)-CFG that is more efficient, an operation for enforcing controllability for an LR(1)-CFG that leads to a terminating and more efficient controller synthesis algorithm, and approaches for ensuring that a DPDA controller has a worst case execution time in its runtime environment.

8.3|p.209 *Future Work*

We recollect starting points for future work identified throughout the thesis and discuss how the domains surveyed in the related work section on the one hand and our contributions on the other hand can mutually benefit from each other.

8.1. RELATED WORK

Discrete event systems appear in many shapes in the literature and the synthesis of programs/submodules/controllers in these contexts is of general interest as pointed out in chapter 1|p.1. Subsequently, we consider and discuss further plant formalisms used earlier for controller synthesis and the aspects that can be represented in these formalisms, different specification formalisms and the properties and aspects that can be characterized using their instances, related domains of closed-loop systems and further synthesis methodologies for discrete event systems from different fields, some available tool-support for (supervisory) controller synthesis, fundamental results of supervisory controller synthesis for the classical pure DFA based setting as well as extensions to further formalism characteristics, earlier results on supervisory control for formalisms with infinite state spaces such as Petri nets and DPDA, and, finally, preexisting formal Isabelle-based frameworks for the formalisms employed in this thesis.

8.1.1. Plant Characteristics and Formalisms

Plant descriptions using formal models can feature a broad range of characteristics for which we now mention examples (references are pointed out throughout our subsequent presentation in this section). The events executed by the plant may be (non)observable, (non)forcible, or (non)controllable for the controller. Also, it is sometimes assumed that controllable and noncontrollable events are alternately executed to ensure some fairness between the execution of events contained in the two sets. Event execution may require time (for example based on the event), may follow nondeterministic, probabilistic, or stochastic rules, may impose costs, which are to be minimized, or may provide rewards, which are to be maximized. Formalisms may allow for parallel composition to describe multiple processes at once and the synthesis procedure may operate symbolically on this parallel composition or may require its explicit calculation, which typically results in a state space explosion. The resulting composed processes may have basic join and fork synchronization such as in Petri nets [272] or may communicate values such as in the CCS [246]. Also, the employed process interaction model may consider latency as well as other characteristics including for example a model of potential kinds of errors.

While we only employ DFA as plant models in this thesis, other formalisms with different characteristics (as discussed above) such as DFA variations (for example, extended DFA in [6]), Petri nets, Control Flow Nets [115], State Tree Structures [228], timed automata [12, 268], Büchi automata [65], process calculi [37, 28, 27, 242], Moore automata [86], DPDA [129], input/output automata [227], Statecharts [152], hybrid automata [11], and CCS have been used in the context of controller synthesis as discussed later on.

One important problem regarding the plant models is, of course, their construction. However, this plant identification problem is beyond the scope of this thesis and we therefore assume a given model of the (possibly physical) plant.

8.1.2. Control Objectives and Specification Formalisms

We discuss different objectives to be enforced by controllers and their characterization using specification formalisms.

—• *Controllability and Nonblockingness*

Controllability and nonblockingness are the two central properties of supervisory controller synthesis [283, p. 219]. On the one hand, controllability is a safety property as it states that the closed loop never enters a state where the controller prevents an uncontrollable event and, on the other hand, nonblockingness is a liveness property stating that the closed loop can reach a marking state from all reachable states. Equivalent characterizations of controllability including the corresponding supremal controllable languages and their interplay with the notion of nonblockingness have been discussed earlier (see [201, 141]). In our Isabelle-based formalization of the abstract controller synthesis algorithm (see chapter 4|p.45), we also considered such further notions that also lead to suitable fixed point iterators. These results have been presented in [310].

—• *Optimal Control*

In optimal control, the controller has the additional objective to choose amongst the viable controllable events to minimize costs (or, dually, to maximize rewards) that are assigned to states or events (the term “optimal” also refers to other notions in the literature). The notion of optimal control strengthens the nonblockingness property by requiring that the next marking state is indeed reached with minimal cost. That is, an optimal controller forces the plant into a marking state of the closed loop. Hence, optimal control is obviously incompatible with the default least-restrictiveness assumption. In [263] the optimal controller synthesis problem was introduced for the DFA based setting. However, uncontrollable events, which may interrupt calculated optimal plans, are not covered in this definition. Also, for systems with finite state space, a solution considering the uncontrollable events has been presented in [200] where (positive or negative) costs are awarded to states (the cost is incurred only for states that are members of the ultimately obtained state space) and a cost is considered for disabling events (uncontrollable events have infinite disabling costs) and the max-flow min-cut algorithm identifies a part of the state space in which all derivations have optimal costs. Further notions of costs have been introduced for optimal control in [263, 262, 61, 315]. Optimal control in combination with controllability and nonblockingness was considered in [317, 314, 313, 316] and applied for discrete event systems such as communication protocols. Also, in [299] optimal control is considered for scheduling multiple production processes. As these mentioned prior results on optimal discrete event control are only applicable to finite state systems, we believe that future work should also focus on optimal control for DPDA closed loops with infinite state spaces. In [320] optimal control is considered for the case when the closed loop is not in a safe region due to unmodelled intermittent disturbances, which brings us to the following problem domain.

—→ *Robust Control*

For robust control uncertainty is assumed, either in the plant model or by the possibility of malfunctioning communication between plant and controller (potentially even in the form of attacks), and the controller is required to be correct even if the modelled undesirable behavior occurs. That is, the robust controller to be synthesized should then be satisfactory for all possible plant behaviors. For example, in [259] unobservable faults in the plant are considered. Also, in [56] a controller is iteratively constructed for a set S of plants at once (also considering time and forced events) and is thereby robust to changes of the plant as long as the plant stays in S . Uncertainty in the plant behavior is characterized and bounded by approach specific notions in [87, 88, 260, 336, 292, 233]. As stated before, controllers should be robust to the adversarial effects in distribution and on interconnection such as in [369, 355, 91].

—→ *Specification Formalisms*

Various formalisms for the specification of relevant properties such as automata, temporal logics, and ad-hoc properties exist. The DFA specification used in [282] is an automata-based specification of the allowed marked and unmarked behavior. Later, the capabilities of DPDA for specification purposes have then been investigated by Christopher Griffin in [141] (see also subsection 8.1.7/p.180). Temporal logics are often capable of expressing a broad range of properties. In [188] a temporal logic is used to state tasks, goals, and properties in general and in [348] such controller synthesis is combined with verification techniques in an incremental way to reduce synthesis costs. Finally, when the chosen specification formalism is insufficient, further desired properties such as controllability and nonblockingness are stated using ad-hoc formulas as in [282] and this thesis. In [13, 9] the linear temporal logic CaRet is introduced where non-regular properties can be stated by relating *call* and *return* positions in the infinite word under interpretation. As stated in [112] some relevant properties cannot be stated using (regular) temporal logics such as LTL, CTL, or CTL*. Examples of such properties relate different traces (for example, “a certain event occurs at the same time in each trace”) or require a precise connection between events (“every transmission is acknowledged”; “there are not more returns than calls” [196]). Examples of context-free extensions of regular temporal logics are also considered in [26].

—→ *Further Controller Characteristics*

Since a given specification may allow for more than one controller to be satisfactory, further properties of the controller have been identified. Some of them are even of high importance for the effectiveness of the entire approach. Firstly, the size of the controller must not exceed the available memory of the hardware on which it is to be executed. Secondly, the formalism used for the controller must allow for a sufficiently fast computation of the admissible events and of the corresponding successor configuration to be reached by the controller when executing the respective event. As for the step-relation of DFA, it is also straightforward to implement the step-relation of DPDA efficiently. However, DPDA

(and also the DPDA controllers constructed by our concrete controller synthesis algorithm) allow for the execution of an unbounded but finite sequence of silent internal steps. On the one hand, this behavior does not contradict controllability because the sequence is finite for the obtained controllers, but, on the other hand, the application of the steps of this sequence may require too much time. Hence, a limitation of the worst case execution time (WCET) during controller synthesis, an a posteriori analysis of the actual WCET, or optimizations of the step-relation computations are called for (see subsection 8.2.3|p.207 for our initial results on enforcing/analyzing a WCET). We also expect that techniques from synchronous programming (see [8, 97, 240] for already considered connections between synchronous programming and discrete event controller synthesis) may prove suitable when tackling this particular problem in the future.

8.1.3. Methods for Controller Synthesis and Theories for Closed-loops

In this thesis, we followed the supervisory controller synthesis approach initially developed in [282] for discrete event systems. We now consider research domains where related synthesis methods are employed or where closed-loops play a central role.

—• Program Synthesis

Program synthesis is a broad field, which can be understood to subsume controller synthesis. For example, in [238], input- and output-conditions are stated in first order logic and the corresponding program is synthesized deductively by deriving a program from a realizability proof. Also, program synthesis in [358] is based on counterexample guided abstraction refinement (CEGAR) [84], which was introduced to reduce the search space during model checking of ACTL* properties. Indeed, the symbolic (i.e., static) supervisory controller synthesis methodology already applies similar ideas where the current controller candidate, which is an abstract model of the controller to be synthesized, is refined iteratively by determining violations of nonblockingness and controllability and by refining the abstract model accordingly. However, the initial abstract model used in [84] is quite different as it is derived from a program and violations determined by our algorithm are never spurious. The comparison would be more close if our initial controller candidate would mark Σ^* because counterexamples could be then spurious when the relevant words would not be executable by the plant (also, the specification satisfaction would have to be checked for violations in the iterations). While we demonstrated by our concrete controller synthesis algorithms that the removal of violations is realizable, the iteratively computed controllers may grow undesirably. Hence, we point out that the development of automatic adaptations of prior controller candidates such as the initial specification (which is typically the smallest controller candidate) to determined violations of nonblockingness or controllability may help to reduce the size of the ultimately obtained controller. In [153] the semi-automatic synthesis of Statecharts from scenario-based require-

ments (given as live sequence charts) is explored along the ideas of game theory from [64, 24] (also see next paragraph). As another approach to state control problems for hybrid systems, initial work on hybrid control programs and the corresponding controller synthesis was introduced in [324]. Choosing a suitable level of abstractions is required in all cases to allow for effective and efficient synthesis: for example, in [155] a program and a usage policy for low-level security-critical primitives is used to synthesize a program that correctly uses these primitives satisfying the policy at lower levels of the protocol stack. Also, in [176, 173], Petri nets are employed for the synthesis of concurrent programs (see also subsection 8.1.6|p.176).

—• *Game Theory and Reactive Synthesis*

Game theory (see [64] for initial work) takes the perspective of, in our terms, a controller that plays a game against the plant where the plant attempts to execute uncontrollable events to drive the closed loop into an undesirable configuration (viz. violation of specification, nonblockingness, or controllability) and where the controller tries to prevent such behavior. This perspective has been taken in the initial work in [274, 1, 105] where reactive programs are synthesized by considering possibly infinite games between an environment (plant) and a program (controller) where a winning strategy can be represented by an infinite tree. Also see the thesis [290] for a thorough introduction to reactive synthesis for synchronous, asynchronous, and distributed systems. In [351], fairness conditions are considered in the general workflow where temporal formula specifications are translated into Büchi automata that accept at least one infinite tree if and only if the specification is satisfiable (actually, the more restricted notion of realizability has been used) by a finite program, which can also be obtained from such an infinite tree. In fact, the mentioned approaches on infinite behaviors as well as their connection to game theory can be seen as a foundation for later considerations on controller synthesis on infinite words (see just below). A strong formal connection between the two research domains of classical supervisory control (see in the paragraphs below) and reactive synthesis (as described just above) is given in [111]. For models of non-regular infinite behaviors, the *verification* has been considered in [54, 53, 52, 51, 50] with various additional systems' characteristics such as for hybrid or parallel systems. Moreover, when restricting to regular infinite behaviors, *supervisory control* has also been considered in [342, 202, 343, 344, 339, 341, 340, 249, 106]. For example, game theory has been applied in [31] for reactive synthesis when using a suitable subset of LTL to define a subset of LTL games and in [29] for decentralized control. Also, in [232] game theory has been used to determine a resource manager for continuous time applications. The perspective of game theory has also proven useful in domains related to controller synthesis such as system verification [352], controller optimization [367], and security protocols [154, 285]. In [196, pp. 441–] game theory is considered for the fixed-point logic with chop (FLC) [252] where additionally stacks are used in the otherwise regular games, that is, the games are then played on configu-

rations of pushdown processes. More precisely, a visibly pushdown system is employed (see subsection 8.3.2|p.214 for our considerations on visibly pushdown systems for controller synthesis). Examples of conversions from non-regular (deterministic) context free languages to FLC are given in [196, Subsubsection 3.2.2, pp. 434–] and the model-checking problem is considered (similarly, analysis and model-checking of (timed) PDA is also considered in [92, 113]). Since the used logic FLC extends the quite powerful modal- μ calculus [332] it would be interesting whether the game-theoretic characterization also translates (possibly with suitable restrictions) to a synthesis procedure in the form of a winning strategy.

— Adaptive systems

Adaptive systems [186, 215, 216, 77] serve as an umbrella notion for domains where feedback loops are used at some level of abstraction such as in control theory. Adaptive systems handle various kinds of uncertainty by incorporating heuristics that are inspired by human made solutions into the actual running system such that the system is able to adapt to changing conditions by executing the encoded heuristics. This integration of domain knowledge and expertise is supposed to make the systems resilient to such expected situations. Thereby, the broad theory of adaptive systems focuses on quite dynamic system structures compared to control theory (see [122] for a similar introduction to the multi-controller scenario for highly reconfigurable systems): also, in this thesis, the life-span of the controller ends whenever the plant model has to be modified. However, in adaptive systems, the desired closed-loop properties are typically not formally identified and analyzed [224]. Many relevant uncertainties and aspects of systems (validity of abstractions, temporal drift among components, incomplete knowledge about the system at start-up, unknown future inputs and parameter selections, usage of automatic learning, decentralization in the sense of composed systems), of goals (elicitation of requirements, accuracy of goal specifications, and goal changes at runtime), of the context (sufficiency of the observability of the execution context and noise as well as perturbations in sensing), and of human involvement (dependability on assumptions on human behavior and unpredictable changes of the plant) that have been considered in [215, pp. 33–36] provide hints on future problems to be solved in more restricted settings such as control theory as well. Vice versa, the integration of results of control theory into adaptive systems is an ongoing key challenge as discussed in more detail in [224, 293] and [215, p. 25]. However, adaptive systems theory subsumes control theory in the sense that it focuses on a broader range of software systems where cyber-physical elements are *not necessarily* assumed to coexist with software components. In particular, [215, pp. 358–363] detail on the connection of supervisory discrete event controller synthesis and adaptive systems theory. For example, control theory has been used in [357] for workflow scheduling and in [187] for sound software adaptation. Also, Petri nets and control theory have been applied in [173, 176, 174, 225] for the synthesis and analysis of computing systems or to avoid

deadlocks as in [360, 359, 25, 107]. Moreover, supervisory controller synthesis has been used in [123, 124] to restrict the behavior of a program when exceptions reveal unsafe regions of behavior that are not known in advance. At the next level, self-healing for adaptive systems is considered for example in [128] where the healing mechanism is determined using more complex algorithms. A more application oriented example where controller synthesis is used is given in [254] where controllers are synthesized to solve an adaptive cruise control problem for moving vehicles using two different approaches (using the state space of the linear system and using a finite-state abstraction). Also, hierarchical control has been considered for adaptive systems in [194, 98, 89]. Further applications of control theory in the domain of self-adaptive software are given (see [224]) in [159] for the controlling of threading level, memory allocation, and buffer pool sharing in commercial products, such as IBM Lotus Notes and IBM DB2, in [205] for computer power control, in [2] for thread cluster management, in [120] for admission control and video compression, and in [103] for performance and denial of service attack mitigation. Examples of discrete control techniques that are employed in self-adaptive systems are also [57, 89, 334] using planning techniques based on artificial intelligence and [44, 90] using game theory (see also above), and reactive synthesis [224]. Lastly, we state that efficiency is a key challenge for adaptive systems because they typically evaluate decision procedures at runtime: for adaptive systems and in discrete event control theory, the rate of events should be sufficiently low compared to the WCET of the decision procedures used by the controller (again, see subsection 8.2.3|p.207 for our initial results on enforcing/analyzing a WCET).

→ *Submodule Construction by Equation Solving via Quotienting*

The supervisory control problem has been stated similarly in terms of the submodule construction problem (see also [45]) in which one submodule P and a specification S are given and a second most general submodule C is to be constructed such that the combination of P and C conforms to S . The formalisms used for P , S , and C as well as the operations for the combination of submodules and the conformance relation differ among instantiations of this problem. For example, the concrete and the abstract supervisory control problems from Definition 3.11|p.39 and Definition 3.7|p.35, respectively, are instantiations of this submodule construction problem and also the supervisory control problem for DFA plants from [282] is an instantiation. But, for the correspondence to the supervisory control, the specification and the satisfaction of the specification must be defined carefully to encompass the properties of nonblockingness, controllability, and least restrictiveness.

The quotienting methodology refers to the fact that the solution C can be obtained from solving an equation of the form $P \times C = S$ by dividing C on both sides, that is, C is then obtained to be S/C where, of course, the relations \times (composition), $=$ (satisfaction), and $/$ (quotienting) are to be interpreted for the concrete setting. This means that the part of S that is already provided by C is

removed and the remainder constitutes the desired submodule C. However, when considering the supervisory controller synthesis, we can see that the quotient idea does not simplify the synthesis tasks on its own.

Early work [245] on submodule construction focused on parallel, distributed, and communicating systems as an important broad domain in which this techniques may prove useful when applying them to instances of suitable size in a hierarchical approach. In fact, the approach has been applied along these lines in [245] for labelled transition systems, in [150] for prefix closed FSA, in [270, 368, 296] for FSA communication through message queues, in [275, 108, 109, 94, 93, 311] using input/output automata (also using input and output events similar to uncontrollable and controllable events), and in [183] to synchronous FSA as in classical supervisory control theory from [281]. Further work is also considered with more complex satisfaction relations such as in [339] where liveness properties are covered, in [48] where safety and liveness properties are considered using a modal logic, in [211] where parametrized bisimulations are used, in [18] where the modal μ -Calculus is used, in [83] where a so called synchronization skeleton is synthesized from a temporal logic specification given in CTL, in [239] where CSP [162] processes are synthesized from propositional temporal logic, in [58, 213, 17, 210, 235, 20, 109] where continuous time properties are covered, in [47] where first order logic is used to recover previous quotienting results of labelled transition systems with interleaved and synchronous rendezvous, in [73] where hybrid systems are considered, in [319, 261, 276, 19] where the CCS calculus with weak bisimulation is used, and in [212] where the Hennessy-Millner-Logic with recursion is employed with its custom satisfaction relation. Moreover, examples of application domains of these techniques are in [185, 337, 74] the design of protocol converters, in [271] the testing of a module in a context, in [45, 46] the analysis of relational databases, in [321] the construction of network protocols such as the alternating bit protocol where deadlocks and unused transitions are removed similarly to nonblockingness in supervisory control, and in [284] the application to a part of the X.21 protocol.

A problem of quotienting is that the size of the formula can increase drastically. To mitigate this problem, a minimization procedure has been introduced in [210] that is to be applied to the resulting formulas. Moreover, note that the referenced works above consider various different forms of decomposition, continuous time, specification logics, model formalisms, synchronization mechanisms, synthesis procedures (such as tableau based or automata based), but none of them consider stack-based capabilities as available in the DPDA used in this thesis.

8.1.4. Tools Implementing Discrete Event Controller Synthesis

Over the years, many prototypes for controller synthesis have been implemented, but not all of these tools are available, maintained, and supported up to this day. A list of tools for the controller synthesis and analysis based on FSA and Petri nets is given at [326] and we now consider some of them in more detail.

Basic functionality for controller synthesis is provided by the following tools. TCT [118, 117] and STSlib [229] are based on language models and support the supervisory controller synthesis of [281]. Desco [114] supports, besides state automata, also Petri nets for supervisor synthesis. UMDES [72] is a library for the study of discrete event systems modeled by DFA. It supports many operations of supervisory control theory and failure diagnosis of discrete event systems following [71]. Supremica [6] also supports supervisory control for DFA and has built-in support for code generation.

More current tools with a broader scope of functionality are as follows. libFAUDES [32] supports controller synthesis and has also various plugins to support, amongst others, hierarchical control, modular and decentralized control, failure diagnosis, and controller synthesis for ω -languages. Also, as stated before, an earlier version of the algorithm presented here has been implemented as a plugin (called *pushdown*) in libFAUDES. CIF3 [35, 34] comes along with its custom automata-based modelling formalism for discrete, timed, and hybrid systems and supports specification, supervisory controller synthesis, simulation based validation, verification, continuous time testing, and code generation. Visual-State [335] is a commercial product that includes the quotienting techniques from [48]. Sigali [241] is also used in the BZR compiler [99, 55, 100] of the synchronous dataflow language Heptagon and also supports supervisor synthesis based on implicit labelled transition systems. ReaX [42, 41, 40] has support for symbolic controller synthesis for infinite state systems given by transition systems with variables (of type Boolean, integer, or real). HCMC [209, 73] implements a quotienting approach for hybrid systems (see above).

As for example pointed out in [70], tools used by the industry such as by Siemens and Rockwell are typically not released to the public and, moreover, adequate hardware support for controllers is known to increase computation speed significantly, but we are not aware of such off the shelf solutions. However, hardware support for Petri net controllers was reported in [323, 22, 214, 322]. Note that none of the tools presented here provide functionality comparable to our prototype CoSy (see section 7.4|p.154).

8.1.5. Supervisory Control for DFA Plants and DFA Specifications

Supervisory controller synthesis for discrete event systems has been established in [282] by considering the case of DFA plants and DFA specifications. Various further works have then been published introducing and considering (combinations of) certain aspects such as for example observability of events and diagnosability of plants, discrete and continuous time, horizontal composition (modular and distributed control), vertical composition (hierarchical control), failures and robustness (see above in Par. *Robust Control*|p.166), and forced events and optimal control (see above in Par. *Optimal Control*|p.165). We now discuss these examples in more detail and also consider cases where slight variations of DFA are used to represent plants, specifications, and controllers.

—• *Synthesis Considering Observability and Diagnosability*

Observability refers to the possibility that events or the current state are (not) observable by the controller (see also [71, pp. 178–] for an introduction). Nonobservability is used to represent for example nonexistence of certain sensors or failures of certain sensors and, hence, this topic is closely related to Par. *Robust Control*|p.166. There are various characterizations of nonobservability: for example, in [222, 256], where the set of all events is partitioned into observable and nonobservable events and in [297] where events may become nonobservable at runtime. Diagnosability then refers to the capability of the controller to determine whether the plant is in a certain set of states when not all events are observable.

We only consider a small number of results that followed the initial work in [279] where the state of the plant is partially observable and where events are fully observable. In [82], this work is extended to a decentralized scenario (see below) where only some events are observable to the controller. States are not observable and events are partially observable in a similar result in [222]. Later, in [220], online and offline diagnosis is discussed where the model is adapted for the offline case. As mentioned before, other aspects are often considered simultaneously such as in [223] where time and observability are considered at once and, besides an adapted notion of observability, also another more restrictive notion of normality is introduced and compared. Also, pure fault-monitoring in the context of partial observability is discussed in [164] with the obvious relationship to state of the art runtime monitoring (runtime verification) approaches such as in [286]. In [78], deviations of the nominal plant behavior are detected under partial event observability and, based on an analysis, the controller is adapted to accommodate for the detected deviations based on their quantitative classification. In [257], safe diagnosability is introduced to state the fact that the degree of observability is sufficient to prevent malfunctioning in the sense of a certain event occurrence. Moreover, an overview of this fault detection and isolation research is provided in [208, 257]. A comparison of different tools for diagnosability has been presented in [67] and observability and diagnosability were also considered using different formalisms in [43, 268, 106, 68].

In this thesis, we have not considered partial observability.

—• *Synthesis Considering Discrete and Continuous Time*

The aspect of discrete time has been introduced to the setting of discrete event supervisory controller synthesis in [219, 60]. In [60], an additional integer counter variable is increased by one in each step of the plant and reset upon each state change of the plant. However, no controller synthesis procedure is defined for this scenario and, similarly to DPDA, the state space is infinite even with this extension because there is no upper bound on the integer counter. In [219], the impact of horizontal composition (see below) is considered where the correctness of the controller depends on event delays (and the corresponding round-trip-times) that are included in the model. The passing of time is modelled explicitly in [59, 58, 223] where an additional *tick* event is used that has a lower priority

compared to so called forcible events. The employed specification formalism can be compiled into a DFA and can express deadlines by limiting the number of tick events. An optimal controller is obtained by determining the minimal number of tick events which permits controller construction. Further work in [363] refined the approach from [223] to ensure that the closed loop is an instance of the used formalism as well, which is necessary for the proposed hierarchical composition (see also below). In [56], the combination of robust control (see Par. *Robust Control*|p.166) and time is considered. Delay-robustness is investigated in [369] in a decentralized context similar to [59]. Modular controller synthesis (see below) for timed discrete event systems is considered in [298]. For example, the tool CIF3 [35, 34] supports supervisory controller synthesis where continuous time models such as hybrid automata are abstracted to discrete event systems.

Controller synthesis using the quotienting approach (see Par. *Submodule Construction by Equation Solving via Quotienting*|p.170) has been developed in [213, 210, 20] for (one-clock) timed automata. Moreover, in [235], a continuous time controller is synthesized using an approach from game theory (see Par. *Game Theory and Reactive Synthesis*|p.168).

We consider aspects of time along the basic ideas given in [223] in subsection 8.2.3|p.207 to enforce/analyze a WCET.

→ *Synthesis Considering Horizontal Composition (Modular and Distributed Control)*

While the classical supervisory controller synthesis for discrete event systems is already confronted with the parallel (synchronous) execution of two modules, the plant and the controller, it turned out that the usage of multiple plant instances or of multiple controller instances at once results in further complications (see also [71, Section 3.8, pp. 199–]). The goal of horizontal composition is to avoid the state explosion problem resulting from the execution of the synchronous composition of multiple (for example DFA based) plant instances. In this approach, controllers are then synthesized for each plant instance separately, which drastically reduces the resource requirements. However, this divide and conquer approach produces complications. Firstly, there is no obvious way to divide a given huge monolithic representation of a plant into smaller plant instances and, consequently, decomposition is typically not applied, but instead separate components of the (physical) plant are modelled independently in plant models (which results in a granularity of plant models that cannot be adjusted automatically). Secondly, specifications are required for the synthesis of the controllers for the different plant models, but while a monolithic specification that is used for all plant models may be more expressive, it does not fit nicely to the modular definition of the plant models from the previous point (see also [71, Section 3.6, pp. 174–] for the usage of modular specifications). Thirdly, once the controllers are synthesized, the grand challenge of this approach is to derive (using the weakest possible sufficient conditions), to check (using additional test procedures), or to ensure (using additional synthesis algorithms) that the chosen form of controller composition yields a controller that is equivalent to a controller that would have been

synthesized for the synchronous product of the plant instances (the global plant instance). In particular, it turns out that nonblockingness is a property that can be lost easily in this composition.

In [365], this approach has been introduced for two supervisors and a *nonconflicting property* ensures the preservation of controllability by the composition of the controllers. This nonconflicting property is a rather strong sufficient condition stating that one supervisor is more restrictive than the other. The desired least restrictiveness is considered in [221] where the parallel composition of least restrictive controllers is also least restrictive w.r.t. the global plant instance. Also the additional aspect of time is considered in [219, 369] where the round-trip-times of events between the components and simultaneity of events is considered with respect to robustness and validity of the model itself. For the setting of DFA plants and controllers, an additional controller is described in [364] that should foresee upcoming conflicts between the individual closed loops resulting in blocking or even deadlock and that then guides the individual controllers accordingly. However, due to lack of further analysis, it is not immediate whether the problem is effectively and efficiently solved for a reasonable class of problems (for example, sufficient conditions could be too restrictive or additional manual and/or computational effort for constructing the additional controller may turn out to be limitations of this approach). Further reports in [371, 116, 192, 298] elaborate on relevant results and extend the discussed approach from above.

The additional impact of an adversary in decentralized control is considered in [258]. In [119], modular control is considered for the formalism of message sequence charts for communicating processes. Similarly, state tree structures (similar to statecharts) are considered in [228, 76, 69, 181] as a framework of horizontal and vertical composition with efficient algorithms based on binary decision diagrams (BDDs).

The impact of using multiple plant instances is often already considered in the quotienting approach (see Par. *Submodule Construction by Equation Solving via Quotienting*|p.170) such as in [213, 210, 20], but robustness as in [219] poses a further challenge not yet considered explicitly.

See also subsection 8.3.2|p.214 for our considerations on horizontal controller composition when using stack-based formalisms.

—• *Synthesis Considering Vertical Composition (Hierarchical Control)*

The previous discussion of horizontal composition already revealed connections to vertical composition to some extent since the additional controller (see [364]) operates at a more abstract level in some concrete approaches. More precisely, the approaches applied for hierarchical control typically rely on an abstraction on the sets of events, that is, certain low-level events are omitted by the abstraction function and are thereby not visible to the upper level. For example, only events representing the start and end of a certain task could be sent to an upper layer in a hierarchical control architecture. However, as for horizontal control (and when both approaches are applied in combination) the paramount challenge is to

ensure that the overall closed loop has the desired properties of controllability and nonblockingness. Note that abstractions are typically derived manually resulting in an additional effort and, moreover, the abstractions limit the capabilities of controller at more abstract layers regarding their capabilities to prevent/control the plant, that is, the degree of freedom is reduced by this approach. The potential benefits of hierarchical control are immediate when understanding the employed concept of abstraction/concretion as a variation of the divide and conquer principle.

The ideas above have been presented in the initial work in [370] and have been extended later on in [362, 363] also focussing on ensuring overall nonblockingness by providing sufficient conditions on the vertical event mappings. Note that the sufficient condition (formulated as an observer and an observation equivalence) is then similar to the notion of behavioral equivalence of for example CCS. The used sufficient conditions were relaxed in [116]. Also, the mentioned approach on state tree structures (see previous paragraph) also focuses heavily on the integration of horizontal and vertical composition [228, 76, 69, 181]. Hierarchical control is also applied in many other settings (as stated before) such as in [247, 96, 248, 278] where hybrid systems are abstracted to discrete event systems.

Up to now, we have not yet considered hierarchical control when considering stack-based formalisms as the horizontal controller composition (see subsection 8.3.2|p.214) seems to be a prerequisite for this development.

8.1.6. Supervisory Control Using Petri Nets

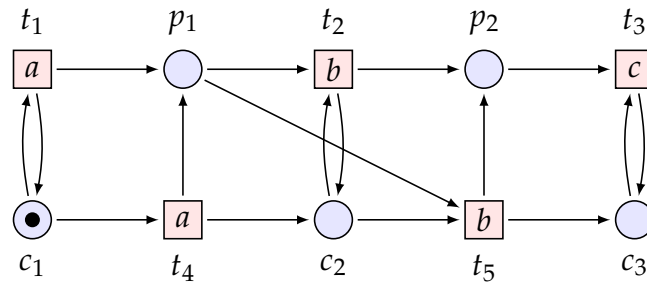
We now consider related work on the control of discrete event systems where Petri nets [272] are employed. The use of Petri nets leads, similarly to the use of DPDA specifications as discussed in the next subsection, to countable but infinite state spaces. Note, as discussed already above, there are also other formalisms such as hybrid automata that result in state spaces of uncountable size, but supervisory controller synthesis is then typically based on an abstraction function to obtain a discrete event system with finite state space instead.

Petri nets are advantageous compared to DFA (as formalisms for controllers, specifications, and closed loops) as they represent a horizontal composition of a varying number of DFA processes. In this interpretation, each token in a Petri net marking can be considered to be the current state of one DFA in such a composition and, by use of synchronization steps based on join and fork, the contained DFA can interact and, moreover, since the number of tokens can increase and decrease, this formalism permits the termination and creation of DFA processes. Also, a Petri net can be understood to be a method to avoid the state space explosion problem that results from the explicit calculation of the synchronous product of multiple DFA because the Petri net can be seen as the disjoint union of multiple DFA rather than the cartesian product (synchronization is then added by shared places/transitions and duplicate structures can be omitted, which also decreases the size of the model). However, the capability of increasing the number of tokens beyond any bound results in a countable

but possibly infinite state space and, at the same time, an increased expressive power. The (un)marked languages of Petri nets are a reasonable choice for the definition of the expressive power also for Petri nets. Note, infinite words of Petri nets are also considered in [267]. The (un)marked words of Petri nets are defined according to [135, 30, 149, 148, 269, 179] as follows: transitions of the Petri net are mapped using a labelling σ to events from Σ (that is, all transitions are observable), a sequence of fired transitions is then mapped to a sequence of events by applying σ to each transition of the sequence, and this sequence of events is then a word of the Petri net. Moreover, a word of a Petri net is a marked word if the final marking of the firing sequence is an element of a set F of marking markings. For example, languages such as $\{ a^n b^n c^n \cdot n \in \mathbf{N} \}$ can be represented by Petri nets (see the following example) but not by DFA or DPDA.

Example 8.1: «A Petri net Using Advanced Counting Capabilities»

Discussion: The marked language of the Petri net below (with the given initial marking) is $\{ a^n b^n c^n \cdot n \in \mathbf{N} \}$ where only $(0,0,1,0,0)$ and $(0,0,0,0,1)$ are marking markings for the ordering $(p_1, p_2, c_1, c_2, c_3)$ of places. It ensures the correct relative number of events by creating an additional token on the special place p_1 for every event a , by moving one token from p_1 to the special place p_2 for each event b , and by removing one token from p_2 for every event c . Note that the Petri net contains nondeterminism (because we used a labelling function σ that is not injective) in the sense that it decides nondeterministically whether it created enough events a (t_1 vs. t_4) and b (t_2 vs. t_5). This is an undesirable situation for the event b because it results in a blocking problem when t_5 is used when the current marking has more than one token on p_1 (consider the unmarked word $aabc$). In this case, the remaining tokens on p_1 cannot be removed in subsequent steps preventing that a marking marking can ever be reached. Essentially, in a Petri net, it is not possible to prevent a transition (such as t_5) from firing if a certain number of tokens are on a certain place (such as p_1). Also see the discussion of inhibitor arcs later on.



Also, languages such as $\{ w c w^{-1} \cdot w \in \{a, b\}^* \}$ can be represented by DPDA but not by DFA or Petri nets. Hence, Petri nets and DPDA are (by their standard definition) incomparable regarding their expressive power w.r.t. their (marked) languages. An important property of Petri nets, which is used in the subsequently presented works, is that incidence matrix analysis can be used to analyze the reachable state space in general and also for satisfaction of invariants.

A general classification of Petri net control is given in [167] as follows. In the *controlled behavior approach*, the closed loop is given as a Petri net and a controller is then to be obtained (in some different formalism) from this closed loop model. See [180, 333, 372, 373] for examples of this approach. In the *logic controller approach*, the controller is given as a Petri net and the closed loop behavior must be analyzed for suitability (for example, regarding nonblockingness and controllability). See [374, 350, 63] for examples of this approach. In the *control theoretic approach*, the supervisory controller synthesis approach is used in accordance with [282]. We now focus on this last approach and provide a short overview of relevant contributions, but it is important to mention that, firstly, the notions of nonblockingness, controllability, and least restrictiveness are not always considered and, secondly, controllers and specifications are not given by Petri nets in each case. In particular, the controller may be given by some formula that enables or disables the firing of transitions based on the current marking and the specification may define forbidden markings using for example so called *generalized mutual exclusion constraints* (which are weaker than Petri net).

Petri nets have been considered in the context of control theory in [95, 253, 198, 170, 140], but the controller synthesis problem was considered later in [197, 166, 199, 165] with heavy restrictions on the classes of Petri nets. Then, in [135], the supervisory controller synthesis for Petri nets has been investigated considering controllability along the lines of [282]. In [135], Petri nets are models for the closed loops and, at the same time, for the controller to be synthesized because the closed loop is usually entirely contained in the plant. That is, the control is added in the form of additional places and arcs to the Petri net plant model, but the use of inhibitor arcs such as in [329] results in undecidability of even fundamental properties. Based on the definitions of unmarked and marked language from above, the standard notions of nonblockingness and controllability are recovered in [135] (along the lines of [282] and section 3.1[p.33]) to state the expected supervisor synthesis problem. However, only a subclass of Petri nets generating regular languages was considered. In [130], this work was extended, but a central problem is that the controller to be synthesized may not be representable by a Petri net or that the closed loop is then not representable by a Petri net (see [130, Section 4.3, p.24]). In [133, 131], the previous result was extended by identifying a subclass of Petri nets where accessibility and nonblockingness may be enforced. Moreover, necessary and sufficient conditions for the existence of a controller in the form of a Petri net are determined when plants and specifications are given by Petri nets. In [327, 204], it was shown that controllability is decidable for the setting of Petri nets because reachability of problematic markings is decidable (however, only prefix closed languages are considered where nonblockingness does not need to be enforced). In [136, 132], an alternative (incomparable) semantics for Petri nets (in terms of marked and unmarked languages) is proposed (a marking M is marking if and only if it contains a marking M' , that is, if $M(p) \geq M'(p)$ for each place p ; the different M' thereby describe different patterns of minimal numbers of tokens

that are required on certain places) that leads to Petri nets that are closed under parallel composition, but, still, the controller to be synthesized is no Petri net of a known Petri net class (this is due to the fact that the generated languages are not closed under arbitrary union as required to obtain the desired supremal element). Due to these results, other control formalisms besides Petri nets were used subsequently. In [134], standard integer linear programming is used for a class of Petri nets to decide whether a Petri net controller is a satisfactory solution to the supervisor control problem by deciding controllability and nonblockingness. The survey [167] suitably collects earlier results of the corresponding authors and also mentions results regarding controllability for timed [151] and colored [184] Petri nets. In [33], supervisory controller synthesis is picked up again and, since a unique least-restrictive satisfactory controller may not exist, some maximal element is computed in the form of a Petri net. However, while alternative maximal controllers cannot be combined or compared in general, it is an open problem which of these maximal controllers is to be chosen (if one of them is to be chosen at all).

Decidability of the reachability of forbidden markings is an important problem (in [281], forbidden words are used as an alternative approach) as it is the first step towards its prevention. The solvability of this problem depends on the class of Petri nets and on the formalisms for specifying such forbidden markings. Results on enforcing the above mentioned generalized mutual exclusion constraints based on place invariants are discussed in [175] and conjunctions of disjunctions of such constraints are used in [231] to specify desired behavior. Moreover, in [230], a subclass of Petri nets (with conflicts and synchronization) and a subclass of generalized mutual exclusion constraints is considered for controller synthesis. The additional aspect of observability (see Par. *Synthesis Considering Observability and Diagnosability*|p.173) is considered in [291] where the forbidden marking problem is solved (for some class of Petri nets) by synthesis of a controller that is least restrictive (but without preventing for example deadlocks).

Applications of controller synthesis (as well as controller synthesis results) for Petri nets are summarized in [167, 175, 174, 138] such as [176, 173] where supervisory control based on Petri nets is considered for the synthesis of concurrent programs, [137] where controller synthesis has been applied (using certain specifications) to a railway network system where the controller and the plant are given as Petri nets, and [184] where colored Petri nets are used in a manufacturing real-time scenario. Also, the survey [174] describes a connection between the research domain of *schedulers* (with the schedulability problem) and controllers (with the supervisory control problem), but the used specifications and schedulers are of different formalisms and future work is required to connect scheduling theory and control theory. Finally, we mention [250] where shared resource systems are modelled by a restricted kind of timed Petri nets called timed event graphs (where the resources are modelled as tokens on a certain place in the Petri net), where the additional aspects of time and optimal control (see Par. *Optimal Control*|p.165) are incorporated, and where a controller synthesis procedure is provided.

Scenarios with restricted subclasses of Petri nets have been considered as well. In [203], controllers are synthesized for Petri net plants and DFA specifications. In [234], controllers are synthesized for colored (but bounded) Petri net plants and so called *consistent permutation symmetry specifications*. In [177], a Petri net controller that is preventing deadlocks is synthesized for Petri net plants. In [156], horizontal and vertical (de)composition is considered from a different perspective, but no controller synthesis algorithm is provided. In [104], the entire finite state space is computed and a control policy is derived to prevent that the Petri net reaches a forbidden marking (also, deadlocks are prevented but nonblockingness is not enforced).

In conclusion, the languages of Petri nets (which allow for more advanced counting capabilities) are incomparable to those of DPDA (which allow for the recording of orderings) and the supervisory controller synthesis (including the central aspects of nonblockingness, controllability, and least restrictiveness) for unrestricted Petri net plants and suitable specifications (for example, based on generalized mutual exclusion constraints) is a challenging problem that is yet to be solved in full generality.

8.1.7. Supervisory Control for DFA Plants and DPDA Specifications

In the context of closed loops, DPDA have been used for example in [354] to detect attacks by recording and monitoring the current call stack of a program and in [295] where a timed DPDA model for a railroad crossing example is verified using constraint logic programming over reals. Moreover, the possibility of more compact representations for regular languages using DPDA has been considered in [125].

Closed loops given by PDA and VPA with partial observability are analyzed in [251, 190, 191] regarding the decidability of the problems of opacity (decide from the observable events whether a certain secret state has been reached) and diagnosability (decide whether a failure state had been reached) also using finite state approximations of them.

For the supervisory controller synthesis, the use of deterministic context-free specification languages (that is, the languages generated by DPDA) has been considered in [283, Section 4] where it was shown that the employed fixed-point computation of the controller may not terminate or may result in a deterministic context-free language in such cases. In [328], the decidability of controllability has been considered for non-DFA formalisms, but no definite results for DPDA were obtained. In [160], the decidability of enforcing controllability for fully and partially observable systems has been considered and verified for the case of DPDA, but no construction procedure is given as in our work in [309, 301]. However, the authors stated that it is “difficult” to extend the decidability proof to the problem of enforcing nonblockingness. An undecidability result for controllability has been presented in [243] for deterministic context-free languages to be used as plant *and* specification languages.

Besides these preliminary considerations, only Christopher Griffin worked on a solution to the supervisor synthesis problem for DFA plants and DPDA specifications in [141, 143, 142, 144, 145, 146] also considering further aspects such as optimal control. However, the synthesis procedure he suggests produces controllers that are not least restrictive as will be discussed in more detail below. Our contributions on supervisory controller synthesis for DFA plants and DPDA specifications have been published in [309, 307, 310, 301, 302].

We now discuss the approach to supervisory controller synthesis of Christopher Griffin as presented in [143, Section 5.2, p. 65]. In his algorithm *B* [143, p. 67], he assumes that all states of the given DPDA controller candidate and the given DFA plant are marking. This implies that the marked languages of these automata are prefix closed and that the automata satisfy the nonblockingness property. The nonblockingness property is preserved by the algorithm and he does not consider the problem of enforcing nonblockingness for DPDA. However, in contradiction to [143, Theorem 5.2.4], his algorithm *B* results in a controller candidate that is not least restrictive. In fact, he determines a substructure of the initial controller candidate, but while this procedure is valid for DFA controllers, it is not valid for DPDA controllers due to the additional stack variable. In particular, an edge of the controller candidate may be used in an initial marking derivation such that a controllability problem cannot be prevented subsequently while it is used in an initial marking derivation where no controllability problem arises. Similarly and also incorrectly, in [143, Section 7, p. 90] Griffin attempts to obtain such a substructure by restricting controllable events to minimize costs for optimal control. Since his analysis of controllability also reveals the pairs of states and viable top-stacks where the current controller candidate violates the controllability property that are obtained in our algorithm, we conclude that there are relevant similarities between his algorithm *B* and our approach. Subsequently, we compare our approach (see section A|p.225 for a remark on coauthorship) from section 5.3|p.82 and the approach of Griffin for identifying controllability problems and then explain the different handling of such detected problems in our approach and the one of Griffin.

—• *Comparison of the Identification of Controllability Problems*

In both approaches, a DPDA controller candidate C_0 and a DFA plant P are the used inputs. The two algorithms then detect controllability problems as follows.

- We determine all pairs (q, X) of a state q of C_0 and a stack element of C_0 such that there is an initial derivation d_1 of the plant P executing events w reaching some configuration c_1 with state p such that there is an initial derivation d_2 of the controller candidate C_0 executing the same events w reaching some configuration c_2 with state q and top-stack X such that the plant P can execute an uncontrollable event $u \in \Sigma_{uc}$ from c_1 but the controller candidate C_0 cannot execute the same event u from c_2 (possibly after applying internal steps).

- Griffin determines edges e of C_0 that execute a controllable event x such that there is an initial derivation d_1 of the plant P executing events $w \in \Sigma^*$ and the event x reaching some configuration c_1 such that there is an initial derivation d_2 of the controller candidate C_0 executing the same events w and x reaching some configuration c_2 by using the edge e in the last step such that there is a derivation d'_1 of the plant P executing events $v \in \Sigma_{uc}^*$ reaching some configuration c'_1 with state p such that there is a derivation d'_2 of the controller candidate C_0 executing the same events v reaching some configuration c'_2 such that the plant P can execute an uncontrollable event $u \in \Sigma_{uc}$ from c'_1 but the controller candidate C_0 cannot execute the same event u from c'_2 (possibly after applying internal steps).

That is, while we detect the actual controllability problem (the lack of a corresponding executing edge in state c_2), Griffin detects the latest edge that executed a controllable event before reaching this configuration. We now consider the steps of the algorithm B of Griffin and describe how he determines these edges.

- *Step 1:* Griffin constructs the complement DPDA C_1 of the DPDA C_0 . This operation follows the construction from [163, Theorem 10.1] and entails a construction that enforces a unique marking behavior as in our approach (see $F_{DPDA-Enforce-Unique-Marking-Late}$ from section 5.3|p.84). The construction of C_1 creates for each state additional edges such that every configuration can be exited by executing any event from Σ (unless an empty step is already possible for the current top stack). The added edges have a common target that is a fresh state and this state is then the unique marking state of C_1 . Hence, the DPDA C_1 marks all words for which the DPDA C_0 gets stuck in some configuration due to a missing edge for the next event to be executed.

This step is applied because controllability problems are now overapproximated by the initial derivations of C_1 that apply an edge that has the additional fresh state as a target state and that executes an uncontrollable event u when P can execute the event u “in the same situation”.

- *Step 2:* Griffin constructs from P and C_1 the product DPDA C_2 by using $F_{DPDA-DFA-Product}$ from section 5.1|p.64. The resulting DPDA C_2 contains states of the form (q, p) where q is a state of the DPDA C_1 and p is a state of the DFA P .

This step is applied because controllability problems are now identified by the initial derivations of C_2 that apply an edge that has the additional fresh state as a target state and that executes an uncontrollable event u . That is, by the product construction the derivations of C_1 and P are suitably connected and only initial derivations are retained where the two automata can proceed together executing the same events. Note, we apply the same product construction in our algorithm to achieve the same simplification.

- *Step 3:* Griffin obtains the DPDA C_3 by removing states and edges from C_2 that cannot occur in any initial marking derivation, but, as mentioned above, he only refers to the general decidability known from the literature for this step. Also, Griffin does not consider the case where all states and edges including the initial state are removed from C_2 .

This step is applied because controllability problems are now identified by the derivations of C_3 that apply an edge e that executes a controllable event x , that then execute a list of uncontrollable events v , and that then apply an edge that has the additional fresh state as a target state and that executes an uncontrollable event u . Note, the invalid assumption that there is always a prior controllable event executed can be fixed by assuming that plant and controller candidate jointly execute a controllable start-up event. In our algorithm, we (roughly speaking) consider the simplified case where the derivation only contains the last step, that is, the edge e executing x is not assumed and v is considered to be the empty list. This is justified by our different handling of controllability problems discussed below. Also, differently from Griffin, we remove inaccessible states and edges using the operation $F_{DPDA-Enforce-Accessible, Opt}$ from subsection 5.2.4|p.80 to ensure that we detect the case where no controllability problems are removed.

- *Step 4:* Griffin constructs a so called predicting machine for C_3 along the lines of [163, Section 10.3]. The obtained predicting machine contains representations of the derivations that end in a configuration with the marking state as described in the previous step in its stack values.

This step is applied because controllability problems are now identified by edges e of the predicting machine that execute a controllable event x where the subsequent top-stack describes such a derivation. In our approach, we do not consider longer sequences of uncontrollable events and determine the relevant top-stack X by dividing each state according to the possible top-stacks using $F_{DPDA-Observe-Top-Stack}$ from section 5.3|p.83 at an earlier step and by removing those created states that cannot be reached as in the previous step. Together with unique late marking and only checking stable states, we then determine the pair (q, X) mentioned in the first item of the previous list.

We conclude that our approach proceeds along the lines of the algorithm of Griffin but is more compact as it avoids the predicting machine that considers longer sequences of uncontrollable events.

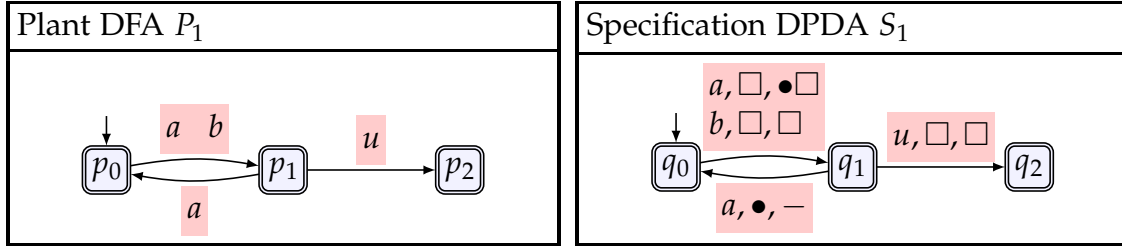
—• *Comparison of the Removal of Controllability Problems*

The procedure of Griffin properly identifies controllability problems, but the removal of all edges identified in the previous paragraph is not least restrictive. Thereby, he correctly prevents initial derivations that use such an edge but produce a stack where the subsequent execution of uncontrollable events leads

to a controllability problem, but he falsely prevents initial derivations that use such an edge but never actually reach a controllability problem later on. The following two examples demonstrate the difference between our algorithm and the algorithm of Griffin for enforcing controllability. In the first example, both algorithms produce the desired solution of the concrete supervisory control problem, but the second example demonstrates that the algorithm of Griffin does not result in the least restrictive controller in all cases.

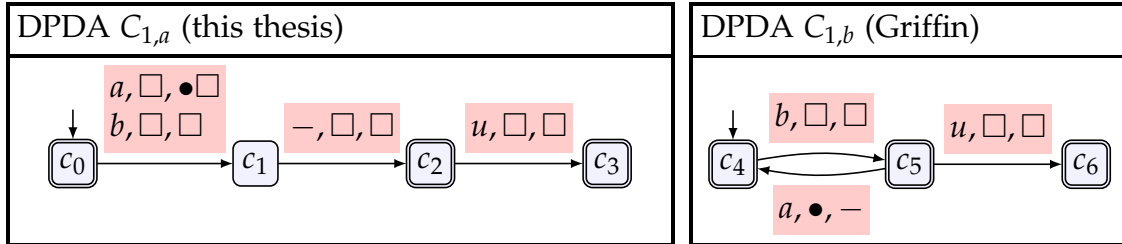
Example 8.2: «Comparison of Algorithms for Enforcing Controllability (1/2)»

Inputs: The plant and specification are given as follows where $\Sigma_{uc} = \{u\}$.



Analysis: Observe that (q_1, \bullet) is a controllability problem because q_1 is accessible with top-stack \bullet but an edge executing u for this state and top-stack is missing in S_1 . Griffin determines that the edge leading to q_1 that executes the event a is the last edge executing a controllable event that is applied whenever this controllability problem is reached in an initial derivation of S_1 . Hence, a satisfactory controller must prevent the event a altogether to resolve this controllability problem. Moreover, a least restrictive controller must allow b and bu to be marked words of the closed loop. Both algorithms succeed in the identification of this controllability problem and result in the following two controllers.

Outputs: The resulting controllers using the two algorithms.



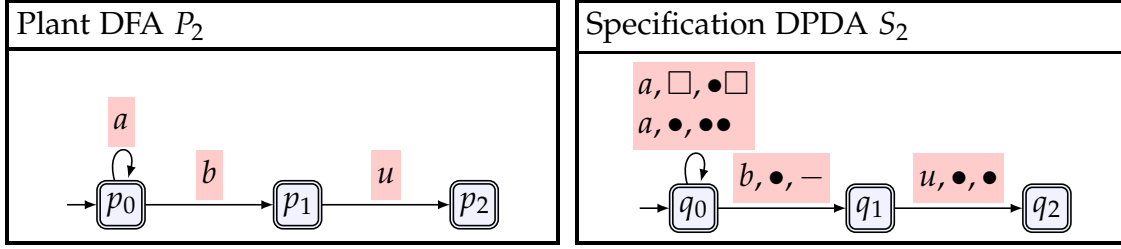
Discussion for the DPDA $C_{1,a}$: After reducing controllability to nonblockingness, nonblockingness is violated in state c_1 for the top-stack \bullet . Using the operation $F_{DPDA-Enforce-Nonblocking, Opt}$, we remove the edge executing a subsequently in our algorithm resolving this problem.

Discussion for the DPDA $C_{1,b}$: After removing noncontrollability, the edge executing a is inaccessible, which can be resolved as well. Note, that the edge removed from S_1 to obtain $C_{1,b}$ is only contained in initial marking derivations of S_1 where a controllability problem is reached.

The following second example demonstrates how we successfully reduce controllability to nonblockingness while Griffin generates a controller that is not least restrictive. The example also demonstrates nicely that the property of least restrictiveness is a crucial requirement of the abstract and concrete supervisory control problem to ensure that an obtained solution is meaningful because the controller of Griffin falsely cuts of a huge part of the desired solution. The steps of the procedure of Griffin with the intermediate results have been included in our presentation in [302].

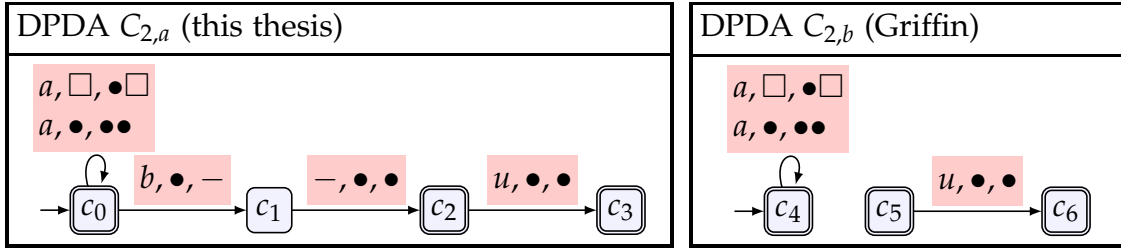
Example 8.3: «Comparison of Algorithms for Enforcing Controllability (2/2)»

Inputs: The plant and specification are given as follows where $\Sigma_{uc} = \{u\}$.



Analysis: Observe that (q_1, \square) is a controllability problem because q_1 is accessible with top-stack \square but an edge executing u for this state and top-stack is missing in S_2 . Griffin determines that the edge leading to q_1 is the last edge executing a controllable event that is applied whenever this controllability problem is reached in an initial derivation of S_2 . Intuitively, the controller must ensure that at least two a occur before the first b occurs to avoid this controllability problem.

Outputs: The resulting controllers using the two algorithms.



Discussion for the DPDA $C_{2,a}$: After reducing controllability to nonblockingness, nonblockingness is violated in state c_1 for the top-stack \square . Using the operation $F_{DPDA-Enforce-Nonblocking, Opt}$, we ensure that at least two events a are executed before the first b is executed resolving this problem.

Discussion for the DPDA $C_{2,b}$: After removing noncontrollability, we obtain a controller candidate that is not least-restrictive because it also prevents marked words such as $aabu$. Note, that the edge removed from S_2 to obtain $C_{2,b}$ is *not* only contained in initial marking derivations of S_2 where a controllability problem is reached. For example, it is important for the marked word $aabu$ that should not have been removed.

We conclude that our algorithm for reducing controllability to nonblockingness (see section A|p.225 for a remark on coauthorship) as presented in section 5.3|p.82, proceeds along the lines of the algorithm of Griffin for the detection of controllability problems but deviates in their handling by reducing this problem to our novel algorithm enforcing nonblockingness. Such a reduction is also the standard strategy in the simpler setting of (not necessarily prefix closed) regular languages and their DFA realizations.

8.1.8. Isabelle Frameworks for EPDA, CFGs, and Parsers

While a broad spectrum of results has already been formalized in Isabelle, we are not aware of alternative Isabelle-based frameworks allowing for the uniform interpretation of the formalisms and semantics we require. Furthermore, existing Isabelle frameworks are often hard to extend, to refactor, or to generalize to make them applicable to a certain task. However, we believe that our framework (see chapter 6|p.89) is, due to the locale hierarchy, easily applicable to various other important formalisms that define discrete event transitions systems. For example, the formalisms of Petri nets, Turing machines, and Büchi automata can be formalized along the lines of the interpretations for the formalisms of EPDA, CFGs, and Parsers presented in chapter 2|p.15 in our framework. Also, due to our fine-grained locale hierarchy, we can presumably include many semantical notions that may become relevant in the future in our framework as well. Furthermore, the application to the controller synthesis algorithm shows that the meta-theoretic definitions and theorems introduced are an acceptable foundation for further formalizations of concrete algorithms. In particular, the transformational results among different semantics and the theorems and definitions on derivations have proven to be a suitable foundation that stabilized thoroughly throughout the verification process of our synthesis algorithm.

The archive of formal proofs [266], where many Isabelle-based formalizations are listed, contains various automata related contributions. These frameworks cannot be extended easily for our purposes where various semantics of various formalisms involving the various relevant semantical properties are to be covered. Furthermore, in most cases, these frameworks have not been available when we started with our formalization. Finally, none of the frameworks suitably covers PDA or Parsers of some kind.

The following frameworks are concerned with regular languages, FSA (in different shapes), CFGs (in Noam Chomsky normal form), and some other more complex automata involving hierarchy or time. In particular, [195] and [366] are concerned with regular expressions (not involving automata). [49] is an executable formalization of the Cocke-Younger-Kasami-algorithm deciding whether a word is contained in the marked language of context free grammar in Noam Chomsky normal form (a formulation of grammars along with their semantics is provided by domain specific definitions). [38] introduces FSA on bit strings and some operations on them based on a certain locale for automata and provides an

executable decision procedure for Presburger arithmetic. [255] introduces FSA using functions as single- and multi-step-behavior and a translation of regular expression into such FSA. In [158], Statecharts are formalized as an example of hierarchical automata. The formalization is also based on sequential automata, which are FSA without marking behavior. [345] defines regular and context free languages based on a coinductive tree-like datatype. This tree-like datatype has some similarities with the branching semantics introduced in Par. *Branching and Linear Semantics* [p.105] that are interpreted for EPDA and Parsers in chapter 2 [p.15]. [23] also covers systems with finite and infinite traces and is applied to for example regular languages. [264] also covers FSA and closure operations on them. [361] introduces the formalism of timed automata with its semantics. [62] introduces DFA, FSA, and Büchi automata by means of a general framework. Many of these frameworks are either not applied to concrete algorithms (thus making an evaluation of the framework with respect to the applicability to concrete algorithms impossible) or are targeted at a single formalism (thus lacking a reasonable amount of generality).

Various process calculi are further formalisms that define discrete event systems of some kind that have been formalized in Isabelle. A formalization of the π -calculus has been provided in [287, 289, 288] and also the spi-calculus has been formalized in [182]. More recently, formalizations of CCS, the π -calculus, and the spi-calculus are given in [36, 266]. In all cases, binders occurring in the terms are a major problem to be handled suitably. However, process calculi are not primarily concerned with the semantical properties relevant in this thesis such as languages and determinism and, hence, the formalizations are fundamentally different.

We conclude that our framework, shortly presented in chapter 6 [p.89], provides a sensible enhancement to the world of Isabelle-based frameworks.

8.2. ONGOING WORK

In the context of our concrete controller synthesis algorithm, we discuss an optimization of the construction of the $LR(1)$ -CFG from an SDPDA and an operation for reducing controllability to nonblockingness that is applied to $LR(1)$ -CFG. These alternative and additional constructions have been implemented and tested in our prototype, but the indispensable formal treatment in Isabelle is missing as of now. We discuss these constructions by examples to demonstrate their value. Moreover, we discuss an approach for enforcing and analyzing worst case execution times (WCET) in the context of DPDA controllers, which is to be applied to a range of concrete application scenarios to underline its usefulness in the future.

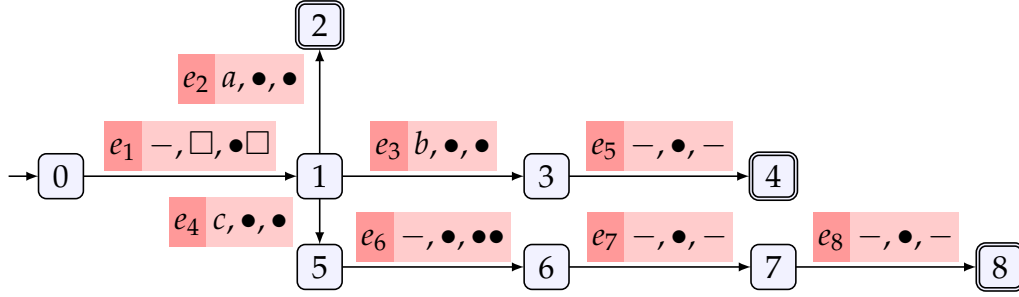
8.2.1. Recursive Construction of the $LR(1)$ -CFG

We introduce the operation $F_{SDPDA \rightarrow LR(1), Rec}$ that translates, similarly to the two operations $F_{SDPDA \rightarrow LR(1), Std}$ and $F_{SDPDA \rightarrow LR(1), Opt}$ from section 5.2.1|p.71, a given SDPDA into the language-equivalent, accessible, and nonblocking $LR(1)$ -CFG. The three operations construct a language-equivalent CFG that is then restricted to enforce nonblockingness and accessibility as required. In comparison to the two other operations, the operation $F_{SDPDA \rightarrow LR(1), Rec}$ generates a smaller CFG that is still language-equivalent and nonblocking and the construction of the CFG and the enforcement of accessibility require (as also supported by our prototype-based evaluation) much less memory and time. We implemented $F_{SDPDA \rightarrow LR(1), Rec}$ in our prototype and used back-to-back testing with the other two operations, but a formal verification is left for future work.

$F_{SDPDA \rightarrow LR(1), Rec}$ constructs a finite directed (possibly cyclic) dependency graph (with a root node) by symbolically applying productions that are possible according to the construction rules given in Definition 6.8|p.117. The nodes of the dependency graph contain left and right hand sides of potential productions where three kinds of nodes also symbolically represent multiple such left and right hand sides at once. We use a blue color for the *successful* nodes of the dependency graph which are *believed* to be relevant for the subsequent generation of productions. The dependency graph is constructed iteratively and we maintain a list of pending nodes, which only contain left hand sides of potential productions, that need to be treated subsequently by our construction procedure. Given such a node containing a left hand side of the form $L_{q,X}$ or $L_{q,X,q'}$, we take action for all edges of the given SDPDA that have q as a source state; also in the case of $L_{q,X}$, we also take action when q is a marking state. We apply three kinds of actions: adding further nodes with the required edges, updating existing nodes (changing colors or extending the symbolic representation), or reusing existing nodes also adding required edges. Reusing existing nodes then leads to possibly cyclic dependency graphs (or as in the example below in a graph that is not a tree).

Example 8.4: «Recursive Construction of the LR(1)-CFG»

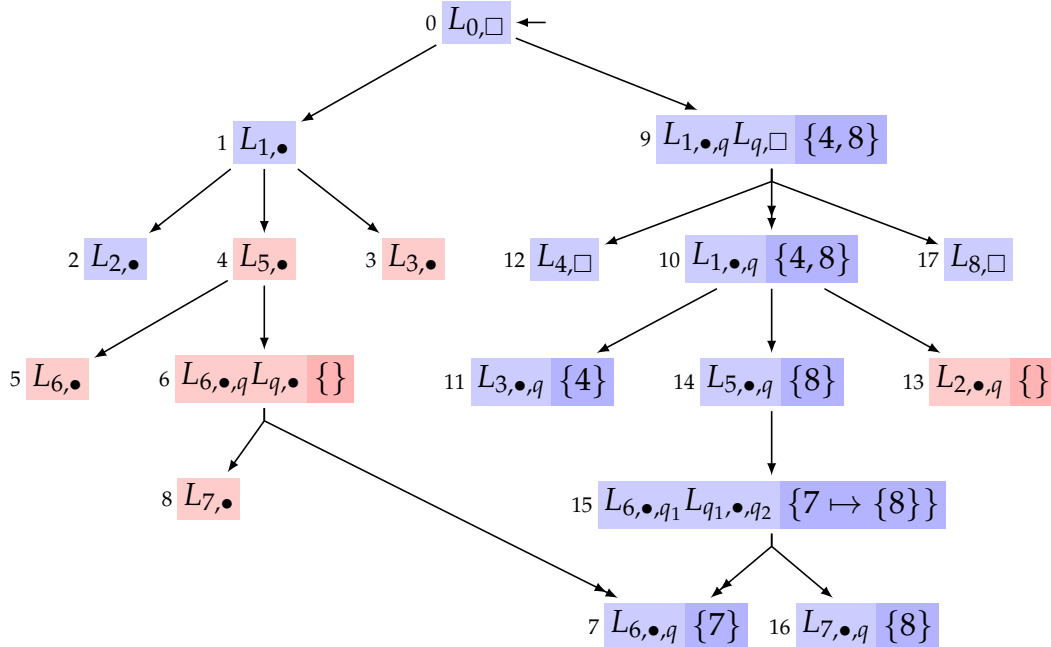
(1) *Input SDPDA C*: Note the identifiers e_1 through e_8 for the edges of the SDPDA.



(2) *Expected Resulting CFG*: The productions in each of the three columns corresponds to the productions that are used in that order in one of the three different initial marking derivations of the expected LR(1)-CFG.

$$\begin{array}{lll}
 \rho_1 : L_{0,\square} \longrightarrow L_{1,\bullet} & \rho_4 : L_{0,\square} \longrightarrow L_{1,\bullet,4} L_{4,\square} & \rho_8 : L_{0,\square} \longrightarrow L_{1,\bullet,8} L_{8,\square} \\
 \rho_2 : L_{1,\bullet} \longrightarrow a L_{2,\bullet} & \rho_5 : L_{1,\bullet,4} \longrightarrow b L_{3,\bullet,4} & \rho_9 : L_{1,\bullet,8} \longrightarrow c L_{5,\bullet,8} \\
 \rho_3 : L_{2,\bullet} \longrightarrow & \rho_6 : L_{3,\bullet,4} \longrightarrow & \rho_{10} : L_{5,\bullet,8} \longrightarrow L_{6,\bullet,7} L_{7,\bullet,8} \\
 & \rho_7 : L_{4,\square} \longrightarrow & \rho_{11} : L_{6,\bullet,7} \longrightarrow \\
 & & \rho_{12} : L_{7,\bullet,8} \longrightarrow \\
 & & \rho_{13} : L_{8,\square} \longrightarrow
 \end{array}$$

(3) *Resulting CFG Dependency Graph for C*: Note the identifiers 0 through 17 for the nodes of the graph that are given to their left. Also note that node 0 is the root node of the graph. Also, for example the hyperedge between nodes 15, 7, and 16 states that the content of node 7 is the *first* nonterminal of the right hand side given in node 15 and that node 16 is an instance of the *second*.



(4) *Incremental Construction*: We provide all steps for the construction of the dependency graph following our general explanation before.

Considered node	Cause for action	Potential production	Kind of action	Result of action
pending nodes: 0				
0	Pushing e_1	$L_{0,\square} \longrightarrow L_{1,\bullet}$	add	1 $L_{1,\bullet}$
		$L_{0,\square} \longrightarrow L_{1,\bullet,q} L_{q,\square}$	add	9 $L_{1,\bullet,q} L_{q,\square} \{\}$
			add	10 $L_{1,\bullet,q} \{\}$
pending nodes: 1, 10				
1	Executing e_2	$L_{1,\bullet} \longrightarrow a L_{2,\bullet}$	add	2 $L_{2,\bullet}$
	Executing e_3	$L_{1,\bullet} \longrightarrow b L_{3,\bullet}$	add	3 $L_{3,\bullet}$
	Executing e_4	$L_{1,\bullet} \longrightarrow c L_{5,\bullet}$	add	4 $L_{5,\bullet}$
pending nodes: 2, 3, 4, 10				
2	Marking	$L_{2,\bullet} \longrightarrow$	upd	2 $L_{2,\bullet}$
			upd	1 $L_{1,\bullet}$
			upd	0 $L_{0,\square}$
pending nodes: 3, 4, 10				
3	<i>no cause</i>			
pending nodes: 4, 10				
4	Pushing e_6	$L_{5,\bullet} \longrightarrow L_{6,\bullet}$	add	5 $L_{6,\bullet}$
		$L_{5,\bullet} \longrightarrow L_{6,\bullet,q} L_{q,\bullet}$	add	6 $L_{6,\bullet,q} L_{q,\bullet} \{\}$
			add	7 $L_{6,\bullet,q} \{\}$
pending nodes: 5, 7, 10				
5	<i>no cause</i>			
pending nodes: 7, 10				
7	Popping e_7	$L_{6,\bullet,7} \longrightarrow$	upd	7 $L_{6,\bullet,q} \{7\}$
			add	8 $L_{7,\bullet}$
pending nodes: 8, 10				
8	<i>no cause</i>			
pending nodes: 10				
<i>continued</i>				

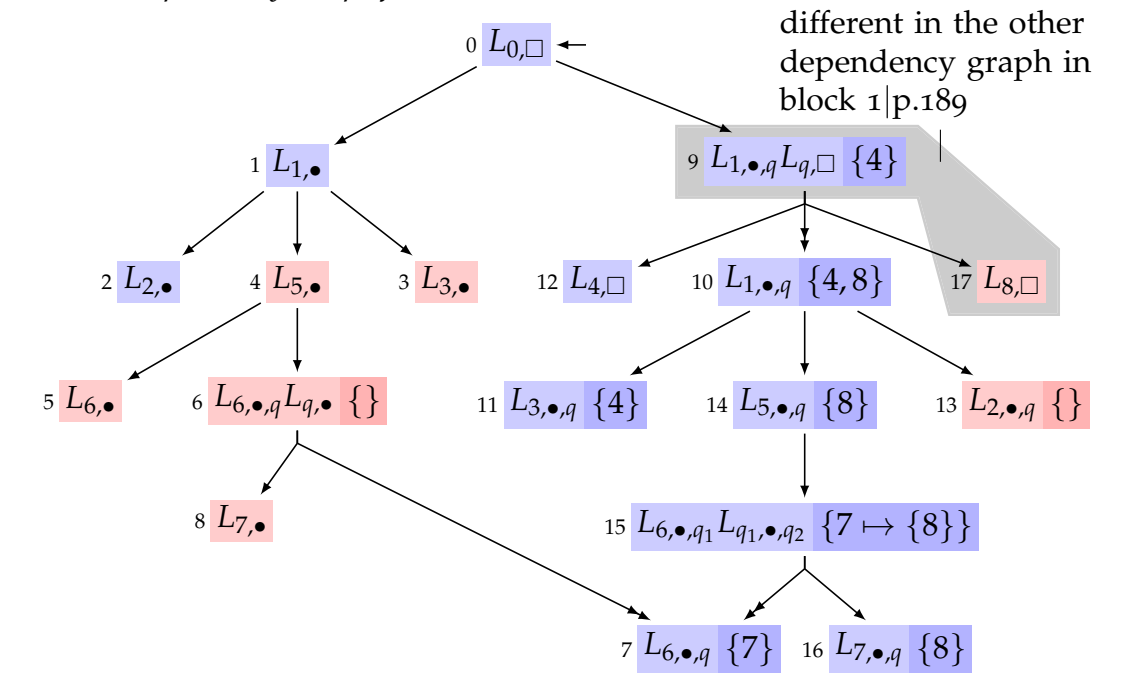
continued				
pending nodes: 10				
10	Executing e_2	$L_{1,\bullet,q} \rightarrow a L_{2,\bullet,q}$	add	$^{13} L_{2,\bullet,q} \{\}$
	Executing e_3	$L_{1,\bullet,q} \rightarrow b L_{3,\bullet,q}$	add	$^{11} L_{3,\bullet,q} \{\}$
	Executing e_4	$L_{1,\bullet,q} \rightarrow c L_{5,\bullet,q}$	add	$^{14} L_{5,\bullet,q} \{\}$
pending nodes: 13, 11, 14				
13	no cause			
pending nodes: 11, 14				
11	Popping e_5	$L_{3,\bullet,4} \rightarrow$	upd	$^{11} L_{3,\bullet,q} \{4\}$
			upd	$^{10} L_{1,\bullet,q} \{4\}$
			add	$^{12} L_{4,\square}$
pending nodes: 12, 14				
12	Marking	$L_{4,\square} \rightarrow$	upd	$^{12} L_{4,\square}$
			upd	$^9 L_{1,\bullet,q} L_{q,\square} \{4\}$
pending nodes: 14				
14	Pushing e_6	$L_{5,\bullet,q_2} \rightarrow L_{6,\bullet,q_1} L_{q_1,\bullet,q_2}$	add	$^{15} L_{6,\bullet,q_1} L_{q_1,\bullet,q_2} \{\}$
			use	$^7 L_{6,\bullet,q} \{7\}$
			add	$^{16} L_{7,\bullet,q} \{\}$
pending nodes: 16				
16	Popping e_8	$L_{7,\bullet,8} \rightarrow$	upd	$^{16} L_{7,\bullet,q} \{8\}$
			upd	$^{15} L_{6,\bullet,q_1} L_{q_1,\bullet,q_2} \{7 \mapsto \{8\}\}$
			upd	$^{14} L_{5,\bullet,q} \{8\}$
			upd	$^{10} L_{1,\bullet,q} \{4,8\}$
			add	$^{17} L_{8,\square}$
pending nodes: 17				
17	Marking	$L_{8,\square} \rightarrow$	upd	$^{17} L_{8,\square}$
			upd	$^9 L_{1,\bullet,q} L_{q,\square} \{4,8\}$
pending nodes: none				

We use two kinds of symbolic representations in the dependency graphs. Firstly, the set that is used in nodes 6, 9, 10, 11, 14, 13, 7, and 16 determines possible replacements for the placeholder q used in these nodes (every state in the set is a possible replacement). Secondly, the map that is used in node 15 determines possible replacements for the two placeholders q_1 and q_2 used in that node (if the map contains the assignment $q_1 \mapsto S$ and $q_2 \in S$, then (q_1, q_2) is a possible replacement).

We obtain the productions given in block 2|p.189 from the resulting dependency graph given in block 3|p.189 by considering all non-hyperedges connecting two blue nodes and by considering all blue nodes with a nonterminal $L_{q,X}$ where q is a marking state of the SDPDA. We can also maintain the relationship between productions of the CFG and the edges of the SDPDA that is required for the operation $F_{DPDA-Enforce-Accessible, Opt}$. In this example, all productions are accessible and, hence, the resulting CFG is already the desired LR(1)-CFG.

(5) *Adapted Input SDPDA C'* : Let the SDPDA C' be defined as the SDPDA from block 1|p.189 where the state 8 is no longer a marking state. For this adapted SDPDA C' , we obtain (see the following dependency graph for C' where only nodes 9 and 17 are changed compared to the dependency graph for C) the same productions from above except that ρ_8 and ρ_{13} are not created because the final step in the construction procedure above cannot be applied because 8 is no longer a marking state. Hence, the resulting CFG would contain the inaccessible productions $\rho_9, \rho_{10}, \rho_{11}$, and ρ_{12} . The reason for this is that when we traverse the dependency graph to obtain the productions (after it has been created entirely), we already obtained the inaccessible productions when realizing in node 9 that node 17 is not successful. Hence, the nodes 14, 15, 7, and 16 are falsely assumed to be successful. We believe that an a posteriori removal of inaccessible productions (see Definition 5.4|p.66 for a description of this procedure) may be more efficient compared to preventing that inaccessible productions are obtained in the first place.

(6) CFG Dependency Graph for C' :



A further optimization of this translation seems not necessary as our prototype-based evaluation suggests that the construction of the LR(1)-Machine is the bottleneck of our concrete controller synthesis algorithm as of now.

8.2.2. Reduce Controllability to Nonblockingness for LR(1)-CFG

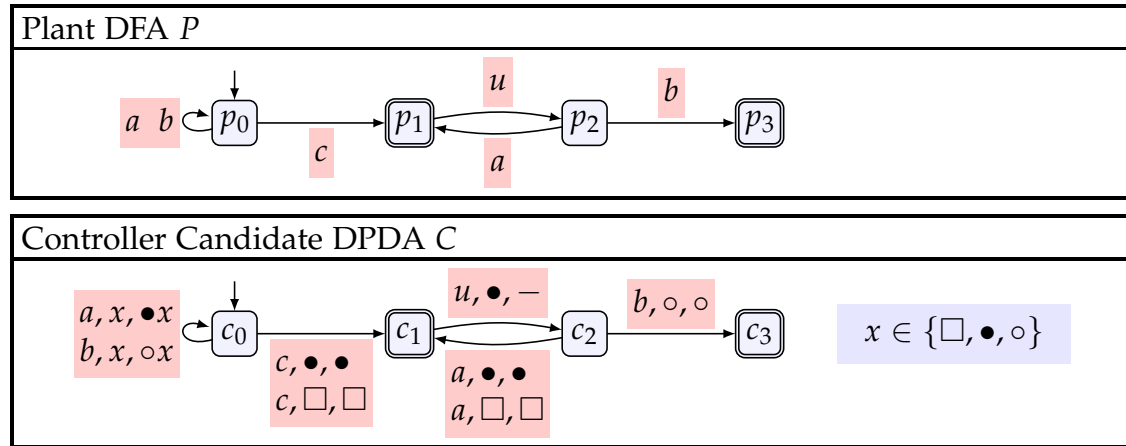
The concrete controller synthesis algorithm presented in chapter 5|p.61 has two drawbacks. Firstly, it does not terminate for some unrealistic specifications that attempt to enforce a precise repetition of uncontrollable events (see section 5.5|p.87 for an example and see subsection 8.3.1|p.209 for another approach to solve this issue within the given algorithm). Secondly, it iteratively converts the controller candidate between DPDA and LR(1)-CFG, which requires the vast amount of time and memory resources (see subsection 8.3.1|p.209 and section 8.3|p.209 in general for approaches to alleviate this issue).

These drawbacks suggest the desirability of a more efficient algorithm, which is investigated subsequently. Firstly, we consider the case of *enforcing* controllability for a given DPDA (i.e., not relying on a reduction to nonblockingness), which is similar compared to enforcing nonblockingness for DPDA because it also entails *non-local* modifications of the DPDA. Secondly, also for the domain of DPDA, we argue that nonblockingness cannot be reduced to controllability to demonstrate (together with the reduction of controllability to nonblockingness used in our synthesis algorithm) that nonblockingness is more complex than controllability (up to termination of the algorithm and when defining the reduction in terms of local modifications of the DPDA). From these two points, we derive our claim that solving both problems in the domain of DPDA is difficult. Then, since enforcing nonblockingness is simple for the CFG obtained from translation of the DPDA, we propose the investigation of enforcing controllability in the context of the obtained LR(1)-CFG. That is, thirdly, we propose to instantiate our abstract controller synthesis algorithm from section 4.3|p.53 to the setting of CFG.

In the following example, we consider the problem of enforcing controllability for a given DPDA and describe the steps of analysis and modification that are relevant for this procedure to conclude that it is a complex problem when it is not reduced to another problem (that is, enforcing nonblockingness in our concrete controller synthesis algorithm).

Example 8.5: «Enforcing Controllability for DPDA»

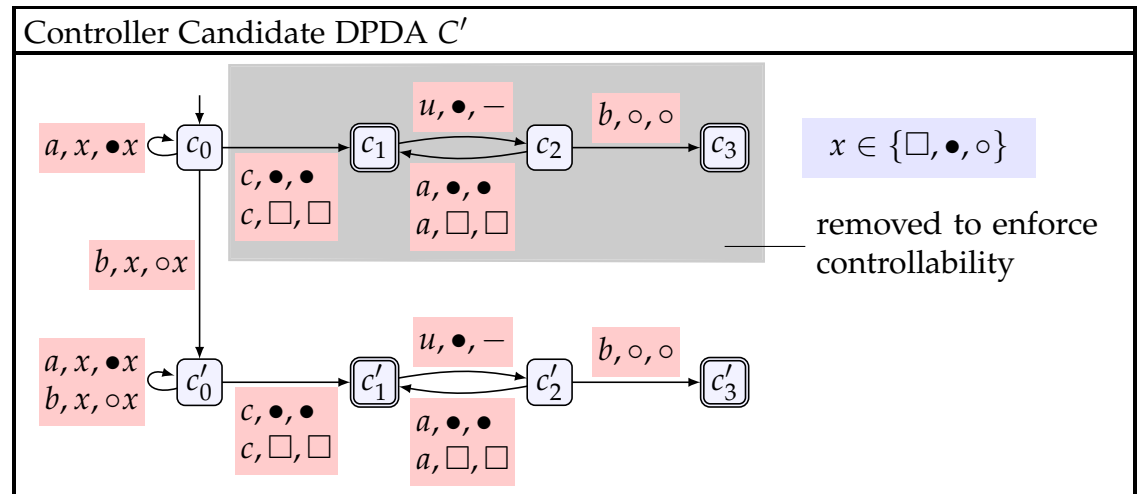
Inputs: The plant and controller candidate are given as follows where $\Sigma_{uc} = \{u\}$.



Analysis of Controllability Problems: Note that only the state c_1 of the DPDA C may have controllability problems because only the corresponding state p_1 in the plant DFA P has an exiting edge that executes an uncontrollable event. Further analysis shows that the state c_1 can be reached with stacks represented by the regular expression $(\bullet(\bullet + \circ)^*\square) + \square$. While a controllability problem occurs in state c_1 only for the stack \square it is also *unavoidable* to reach this controllability problem whenever the state c_1 is reached with a stack in $\bullet^*\square$ (the admissible stacks for state c_1 are given by the regular expression $\bullet^*\circ(\bullet + \circ)^*\square$) because the controller must always permit the event a to reach the marking state c_1 but it prevents eventually the uncontrollable event u .

Detection/Localization of Unavoidable Controllability Problems: From the example above, we can conclude that it is insufficient to consider only the top-stack to determine controllability problems. However, considering a bounded prefix of the stack to detect stacks that unavoidably lead to a controllability problem is also insufficient because \bullet^k is a prefix of the stack $\bullet^k\square$ unavoidably leading to noncontrollability but also of the admissible stack $\bullet^k\circ\square$ definitely not leading to a controllability problem. This observation also implies that the controller to be synthesized cannot avoid controllability problems by inspecting a prefix of the current stack at runtime.

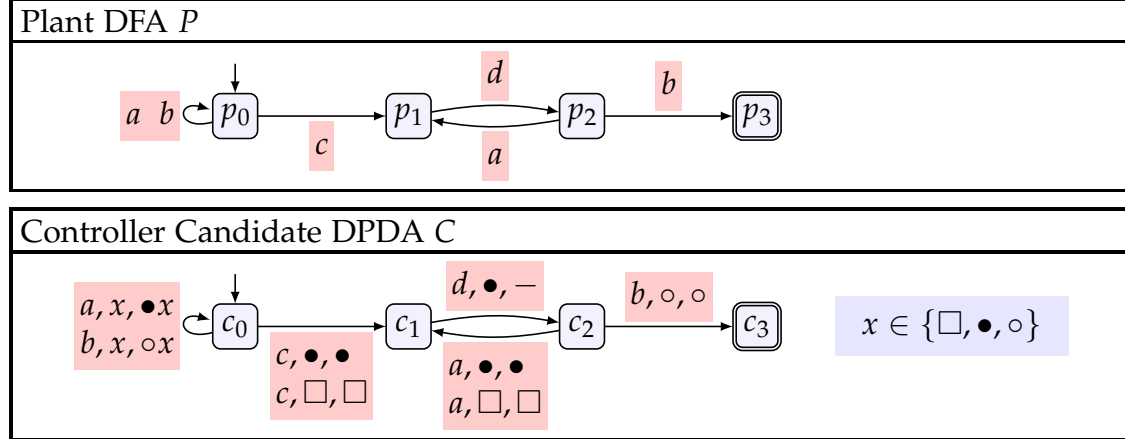
Resolving Controllability Problems: We now assume that the stacks that unavoidably lead to controllability problems have been detected by some procedure, that is, for the current example, we assume that the regular expression $\bullet^*\square$ denoting such problematic stacks for c_1 is given. Manually, we obtain the disambiguated DPDA C' below in which c_1 is only reached with the problematic stacks $\bullet^*\square$ (and that can be removed for that reason) and where its counterpart c'_1 is reached with the remaining admissible stacks (given above). However, this disambiguation requires a non-local modification, which is similarly applied for enforcing nonblockingness for a given DPDA in Example 5.1|p.67. We claim that also the problem of suitably modifying the DPDA is hard.



We believe that nonblockingness cannot be reduced to controllability by means of local modifications because nonblockingness is defined in terms of reachability by unboundedly long derivations whereas controllability is only concerned with the immediately upcoming events. However, we demonstrate in the following example how the removal of deadlocks, which is a subproblem of the removal of nonblockingness, can be reduced to controllability.

Example 8.6: «Reducing Deadlocks to Controllability for DPDA»

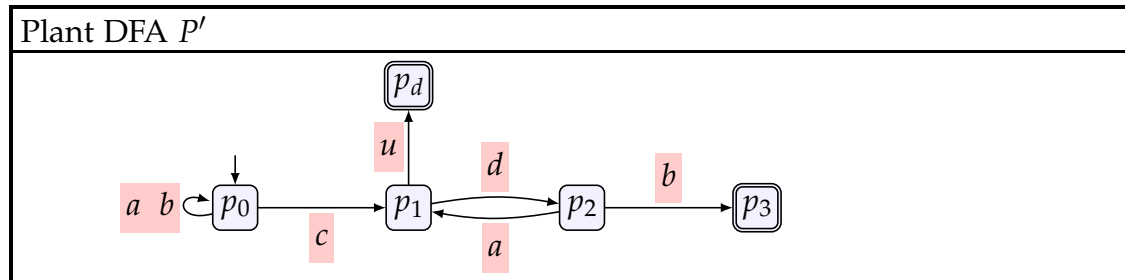
Inputs: The plant and controller candidate are given as follows where $\Sigma_{uc} = \{\}$.

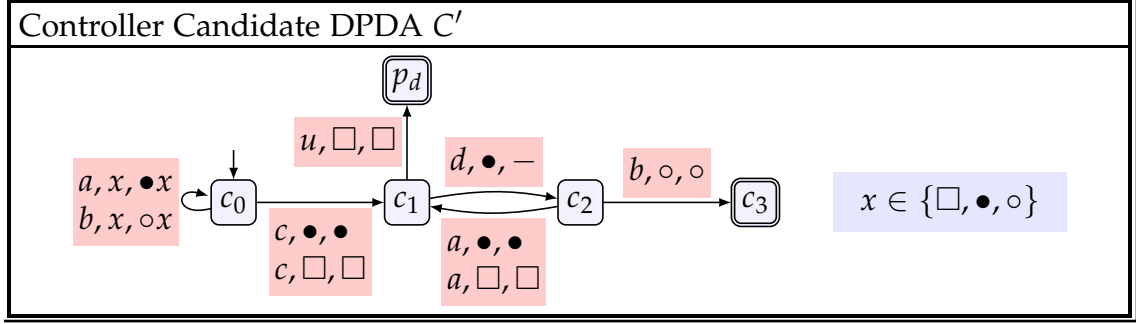


Analysis of Deadlock Problems: The deadlock problems are given for state c_1 with the stack \square . However, the state c_1 can be reached with stacks represented by the regular expression $(\bullet(\bullet + \circ)^*\square) + \square$ and also reaching the deadlock is *unavoidable* whenever the state c_1 is reached with a stack in $\bullet^*\square$ (the admissible stacks for state c_1 are given by the regular expression $\bullet^*\circ(\bullet + \circ)^*\square$) because the controller must always permit the event a to avoid the deadlock in c_2 but it prevents eventually all further execution in c_1 .

Note that this analysis is quite similar to the analysis in Example 8.5|p.193.

Reduction of Deadlocks to Controllability: We obtain a plant P' , a controller candidate C' , and a set of uncontrollable events Σ'_{uc} by adapting the plant P , the controller candidate C , and the set of uncontrollable events Σ_{uc} as follows to establish a controllability problem in C' for every deadlock problem present in C . For this purpose, we add a new marking state p_d to P and C , add an uncontrollable event u to Σ_{uc} , add edges of the form $(edge\text{-}src=c_i, edge\text{-}event=u, edge\text{-}pop=X, edge\text{-}push=X, edge\text{-}trg=p_d)$ to C for every non-marking state c_i for which no edge exits for the stack X , and add the corresponding edges also to P .



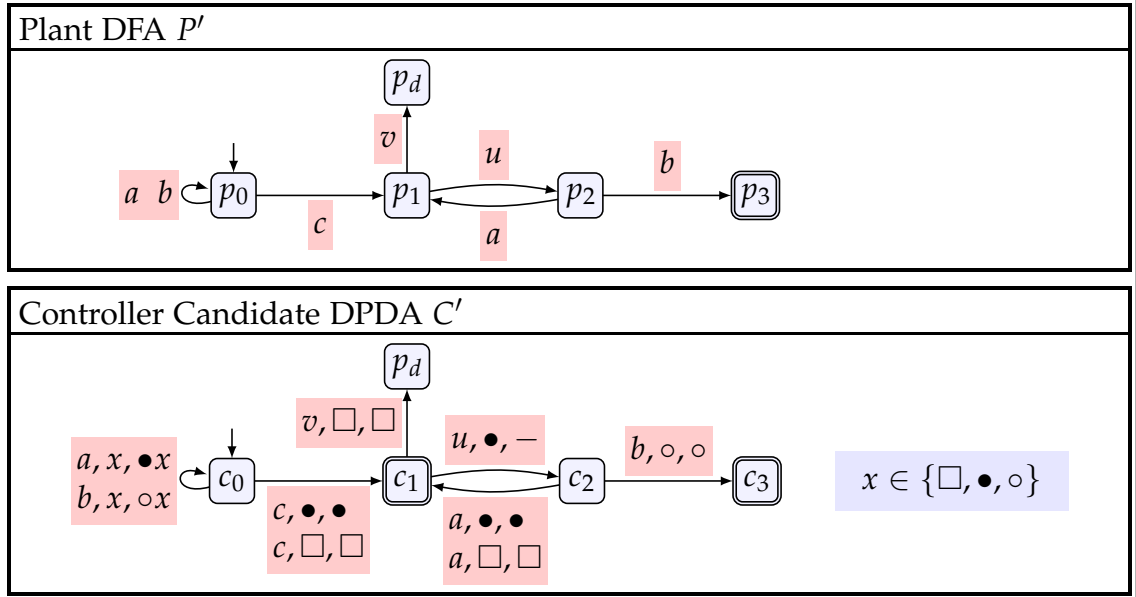


As a last reduction, we demonstrate that controllability problems can be reduced to deadlocks. While this last reduction is not practical, we can now conclude, together with the other reductions discussed before, that enforcing nonblockingness is strictly harder than enforcing controllability and the absence of deadlocks, which are equally hard (up to termination and when only relying on local modifications of the DPDA in the reductions).

Example 8.7: «Reduce Controllability to Deadlocks for DPDA»

Inputs and Analysis of Controllability Problems: See Example 8.5|p.193 above for P , C , Σ_{uc} , and the analysis why (c_1, \square) is the only controllability problem.

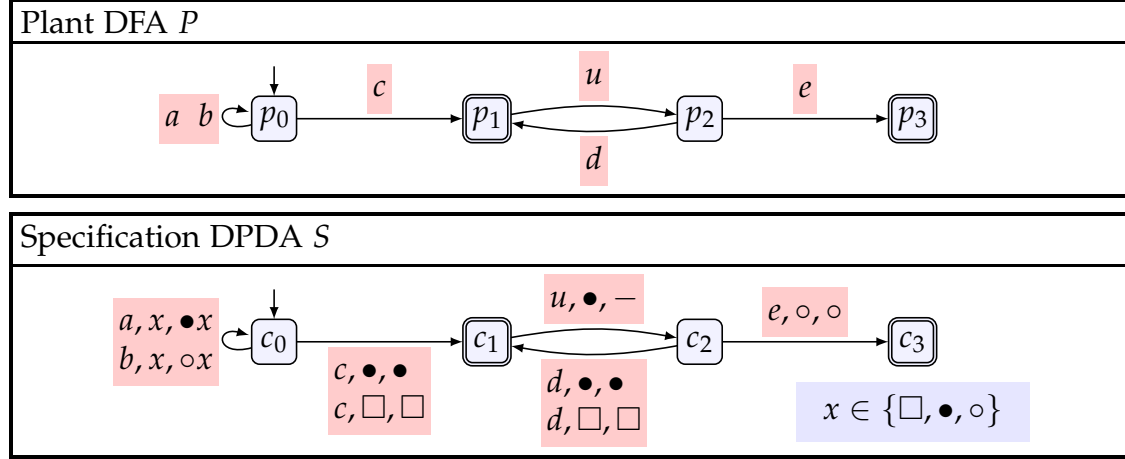
Reduction of Controllability to Deadlocks: We obtain a plant P' , a controller candidate C' , and a set of uncontrollable events Σ'_{uc} by adapting the plant P , the controller candidate C , and the set of uncontrollable events Σ_{uc} as follows to establish a deadlock problem in C' for every controllability problem present in C . For this purpose, we add a new marking state p_d to P and C , add an uncontrollable event v to Σ_{uc} , add edges of the form $(edge\text{-}src=c_i, edge\text{-}event=v, edge\text{-}pop=X, edge\text{-}push=X, edge\text{-}trg=p_d)$ to C for every state c_i with a potential controllability problem for which no edge exits for the stack X , and add the corresponding edges also to P .



We now propose, by providing an example along the way, a controller synthesis algorithm in which we reduce controllability to nonblockingness for a given LR(1)-CFG controller candidate and where we enforce nonblockingness by trimming a CFG. Also, to compute the initial nonblocking LR(1)-CFG controller candidate, we reuse various operations used in our concrete controller synthesis algorithm from chapter 5|p.61.

Example 8.8: «Controller Synthesis Algorithm for LR(1)-CFG»

(1) *Inputs:* The plant and the specification are given as follows where $\Sigma_{uc} = \{u\}$.



(2) *Analysis:* Both of the automata P and S satisfy the nonblockingness property, the marked language of P contains the marked language of S , and the marked language of S is the union of the following disjoint sets.

$$\{ \quad \quad \quad c \quad \quad \quad . \quad \quad \quad n \in \mathbf{N} \} \quad (8.1)$$

$$\{ aa^n \quad c \quad \quad \quad . \quad \quad \quad n \in \mathbf{N} \} \quad (8.2)$$

$$\{ wbaa^n \quad c \quad \quad \quad . w \in \{a, b\}^*, \quad n \in \mathbf{N} \} \quad (8.3)$$

$$\{ aa^n \quad a^{m+1}c(ud)^{m+1} . w \in \{a, b\}^*, n, m \in \mathbf{N} \} \quad (8.4)$$

$$\{ \quad \quad \quad a^{m+1}c(ud)^{m+1} . w \in \{a, b\}^*, \quad m \in \mathbf{N} \} \quad (8.5)$$

$$\{ wbaa^n a^{m+1}c(ud)^{m+1} . w \in \{a, b\}^*, n, m \in \mathbf{N} \} \quad (8.6)$$

$$\{ wba \quad a^{m+1}c(ud)^m ue . w \in \{a, b\}^*, \quad m \in \mathbf{N} \} \quad (8.7)$$

When taking S as a controller candidate, there is only the immediate controllability problem (c_1, \square) in which an edge executing u for this state and top-stack is missing in S . However, a controllability violation is also unavoidable if and only if no b occurs before reaching c_1 (that is for $(c_1, \bullet^* \bullet \square)$ and $(c_2, \bullet^* \square)$) because the event e will never be enabled and the controller must return to state c_1 whenever it reaches c_2 to satisfy nonblockingness and, hence, the controllability problem (c_1, \square) is eventually reached. Stated differently, the sets (Eq. 8.1) and (Eq. 8.5) contain marked words with immediate controllability problems, the sets (Eq. 8.2) and (Eq. 8.4) contain marked words with unavoidable controllability problems, and the sets (Eq. 8.3), (Eq. 8.6), and (Eq. 8.7) should be retained during controller synthesis.

(3) *Construction of the initial LR(1)-CFG controller candidate (1/4):* We reuse multiple operations from our concrete controller synthesis algorithm from chapter 5|p.61 to determine a controller candidate in the form of an LR(1)-CFG.

- We determine the DPDA C_0 to be $F_{DPDA-DFA-Product} S P$
See section 5.1|p.64.
- We determine the SDPDA C_1 to be $F_{DPDA \rightarrow SDPDA} C_0$
See section 5.2.1|p.69.
- We determine the SDPDA C_2 to be $F_{SDPDA-EUME} C_1$
See section 5.2.1|p.70.
- We determine the optional LR(1)-CFG C_3^{opt} to be $F_{SDPDA \rightarrow LR(1), Rec} C_2$
See subsection 8.2.1|p.188.
- If C_3^{opt} is *None*, we return *None*
We conclude that there is no satisfactory DPDA controller for C and P .
- If C_3^{opt} is *Some* C_3 , we continue.
- We determine the LR(1)-CFG C_4 to be $F_{CFG-Rename-Nonterminals} C_3$
We construct an injective renaming function that maps each nonterminal A of C_3 to a pair $n p$. The association of A with the plant state p is contained in A due to the product construction. The natural number n is used to ensure injectivity. We only apply this renaming function to ease readability. The default color for nonterminals is blue in the following.

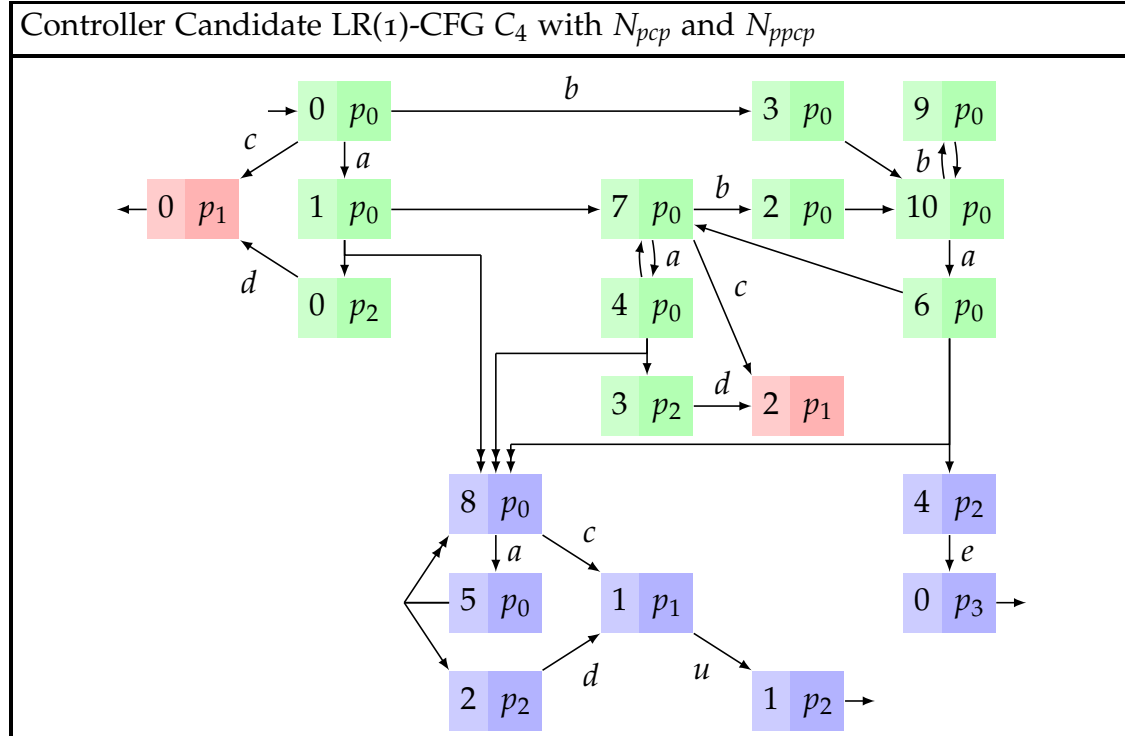
Controller Candidate LR(1)-CFG C_4 with initial nonterminal $0 p_0$

$0 p_0 \rightarrow a 1 p_0$	$7 p_0 \rightarrow c 2 p_1$
$0 p_0 \rightarrow b 3 p_0$	$8 p_0 \rightarrow a 5 p_0$
$0 p_0 \rightarrow c 0 p_1$	$8 p_0 \rightarrow c 1 p_1$
$1 p_0 \rightarrow 7 p_0$	$9 p_0 \rightarrow 10 p_0$
$1 p_0 \rightarrow 8 p_0 0 p_2$	$10 p_0 \rightarrow a 6 p_0$
$2 p_0 \rightarrow 10 p_0$	$10 p_0 \rightarrow b 9 p_0$
$3 p_0 \rightarrow 10 p_0$	$0 p_1 \rightarrow$
$4 p_0 \rightarrow 7 p_0$	$1 p_1 \rightarrow u 1 p_2$
$4 p_0 \rightarrow 8 p_0 3 p_2$	$0 p_2 \rightarrow d 0 p_1$
$5 p_0 \rightarrow 8 p_0 2 p_2$	$1 p_2 \rightarrow$
$6 p_0 \rightarrow 7 p_0$	$2 p_2 \rightarrow d 1 p_1$
$6 p_0 \rightarrow 8 p_0 4 p_2$	$4 p_2 \rightarrow e 0 p_3$
$7 p_0 \rightarrow a 4 p_0$	$0 p_3 \rightarrow$
$7 p_0 \rightarrow b 2 p_0$	

(4) *Construction of the initial LR(1)-CFG controller candidate (2/4)*: We now determine the set N_{pcp} of nonterminals with *potential controllability problems* for which further analysis is required to determine whether they describe true or false controllability problems. Also, we determine their parent nonterminals N_{ppcp} and use these two sets in the next step to disambiguate the LR(1)-CFG C_4 .

- We determine N_{pcp} to be $F_{CFG-Potential-Controllability-Problems} C_4 P$
A nonterminal $n p$ contained in C_4 has a *potential controllability problem* if and only if there is some uncontrollable event $u \in \Sigma_{uc}$ such that P has an edge that exits p and that executes u and u is not in the first set of the nonterminal $n p$ (that is, if $[u]$ is not in $F_{CFG-First} C_4 1 [beA n p]$; see section 5.2.2|p.73). The elements of N_{pcp} are colored red such as in $n p$.
- We determine N_{ppcp} to be $F_{CFG-Parents} C_4 N_{pcp}$
We determine all parent nonterminals of nonterminals in N_{pcp} . Formally, a nonterminal is in N_{ppcp} if there is a finite initial marking derivation d in $cfgSTD$ ending in a configuration that contains a nonterminal from N_{pcp} that is not contained in any other configuration of d . The elements of $N_{ppcp} - N_{pcp}$ are colored green such as in $n p$.

We now visualize the LR(1)-CFG C_4 similarly as in Example 8.4|p.189. Productions such as $1 p_0 \rightarrow 8 p_0 0 p_2$ are represented by a hyperedge, productions such as $0 p_0 \rightarrow a 1 p_0$ are represented by a labelled edge, productions such as $3 p_0 \rightarrow 10 p_0$ are represented by an unlabelled edge, and productions such as $0 p_1 \rightarrow []$ are represented by an unlabelled exiting edge without target.



(5) *Construction of the initial LR(1)-CFG controller candidate (3/4)*: We conduct in this step and in the next step two disambiguations of the LR(1)-CFG C_4 to ensure that the subsequent removal of nonterminals with true controllability problems is least restrictive, which should follow from the fact that all removed initial derivations indeed exhibit a controllability problem.

- We determine the LR(1)-CFG C_5 to be $F_{CFG-Pull} C_4$
To obtain C_5 , we add productions to C_4 as follows. If $\rho_1 : n_1 p_1 \rightarrow n_2 p_2$ and $\rho_2 : n_2 p_2 \rightarrow w$ are two productions of C_4 , we add the production $\rho_3 : n_1 p_1 \rightarrow w$. This additional production does not alter the marked language as it is just the merging of the two given productions. However, if ρ_2 is removed to reduce controllability to nonblockingness, this is then not automatically also the case for ρ_3 and, indeed, retaining ρ_3 can then result in least restrictiveness.

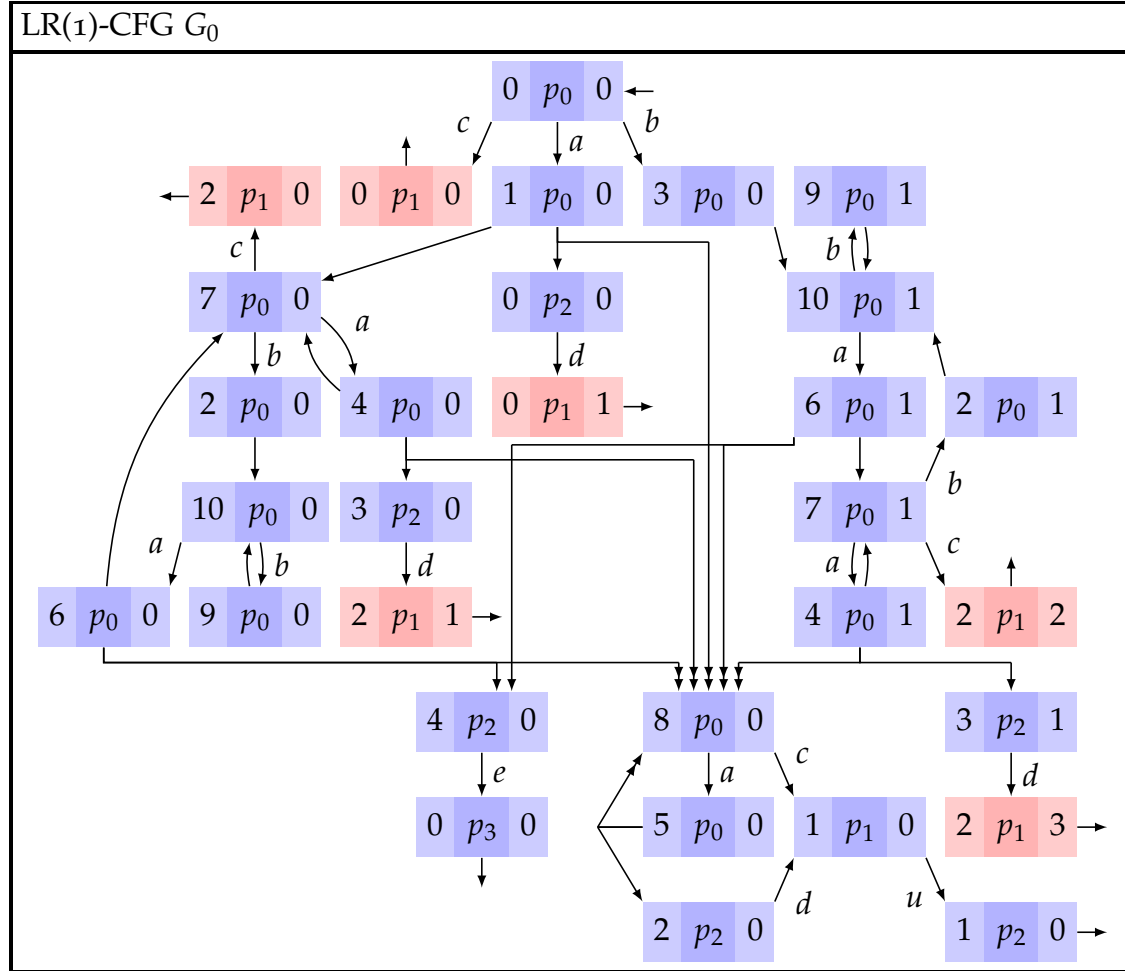
In the LR(1)-CFG C_4 , we have 6 productions of the form ρ_1 and we have to add the following 15 productions to obtain the resulting LR(1)-CFG G_5 .

Production ρ_1	Production ρ_2	Production ρ_3
$1 p_0 \rightarrow 7 p_0$	$7 p_0 \rightarrow a 4 p_0$ $7 p_0 \rightarrow b 2 p_0$ $7 p_0 \rightarrow c 2 p_1$	$1 p_0 \rightarrow a 4 p_0$ $1 p_0 \rightarrow b 2 p_0$ $1 p_0 \rightarrow c 2 p_1$
$4 p_0 \rightarrow 7 p_0$	$7 p_0 \rightarrow a 4 p_0$ $7 p_0 \rightarrow b 2 p_0$ $7 p_0 \rightarrow c 2 p_1$	$4 p_0 \rightarrow a 4 p_0$ $4 p_0 \rightarrow b 2 p_0$ $4 p_0 \rightarrow c 2 p_1$
$6 p_0 \rightarrow 7 p_0$	$7 p_0 \rightarrow a 4 p_0$ $7 p_0 \rightarrow b 2 p_0$ $7 p_0 \rightarrow c 2 p_1$	$6 p_0 \rightarrow a 4 p_0$ $6 p_0 \rightarrow b 2 p_0$ $6 p_0 \rightarrow c 2 p_1$
$2 p_0 \rightarrow 10 p_0$	$10 p_0 \rightarrow a 6 p_0$ $10 p_0 \rightarrow b 6 p_0$	$2 p_0 \rightarrow a 6 p_0$ $2 p_0 \rightarrow b 6 p_0$
$3 p_0 \rightarrow 10 p_0$	$10 p_0 \rightarrow a 6 p_0$ $10 p_0 \rightarrow b 6 p_0$	$3 p_0 \rightarrow a 6 p_0$ $3 p_0 \rightarrow b 6 p_0$
$9 p_0 \rightarrow 10 p_0$	$10 p_0 \rightarrow a 6 p_0$ $10 p_0 \rightarrow b 6 p_0$	$9 p_0 \rightarrow a 6 p_0$ $9 p_0 \rightarrow b 6 p_0$

For our running example, we proceed by applying the disambiguation from the following block 6|p.201 again to C_4 (and not to C_5 as defined by our algorithm) for demonstration purposes. In particular, this allows us to discuss the impact of the disambiguation above more clearly and keeps the intermediately obtained LR(1)-CFGs suitably compact for presentation.

(6) *Construction of the initial LR(1)-CFG controller candidate (4/4):* We conduct a further disambiguation of the LR(1)-CFG C_5 to ensure that the subsequent removal of nonterminals with true controllability problems is least restrictive, which should follow from the fact that all removed initial derivations indeed exhibit a controllability problem.

- We determine the LR(1)-CFG G_0 to be $F_{CFG-Split} N_{ppcp} C_5$
 We consider all paths of C_5 given by its initial derivations in $cfgSTD$ and use in each path a custom replica for each nonterminal in $R = (N_{ppcp} \cup N_{pcp}) - \{0 p_0\}$. Note, we consider the initial derivation only until no fresh nonterminals occur in some configuration (that is, we do not unfold loops). Also, the resulting LR(1)-CFG has a single replica for each nonterminal not in R (e.g. $8 p_0 0$). For example, the two paths $\omega_1 = 0 p_0 \cdot 0 p_1$ and $\omega_2 = 0 p_0 \cdot 1 p_0 \cdot 0 p_2 \cdot 0 p_1$ are disambiguated by using a different replica (constructed by adding another natural number) of $0 p_1$ in the resulting paths $\omega'_1 = 0 p_0 0 \cdot 0 p_1 0$ and $\omega'_2 = 0 p_0 0 \cdot 1 p_0 0 \cdot 0 p_2 0 \cdot 0 p_1 1$ in G_0 .



This last step also replicated occurrences of nonterminals with potential controllability problems. The LR(1)-CFG G_0 above contains now six such nonterminals.

(7) *Reducing Controllability to Nonblockingness*: We obtained an initial controller candidate in the previous steps. In our overall fixed-point algorithm (see next block), we enforce controllability on such controller candidates G_i that are given in the form of an LR(1)-CFG as follows. The controller candidates must have undergone the disambiguation steps in block 5|p.200 and block 6|p.201 to ensure that the subsequently discussed removals do not invalidate least restrictiveness. This is discussed for our example in block 9|p.203.

1. We construct the LR(1)-Machine M_i for the given LR(1)-CFG G_i as in our concrete controller synthesis algorithm.
 We determine the fresh symbol S' not in *cfg-nonterminals* G ,
 we determine the fresh symbol $\$$ not in *cfg-events* G ,
 we determine the augmented CFG G'_i to be $F_{CFG-Augment} G_i S' \$$, and
 we determine the LR(1)-Machine M_i to be $F_{LR(k)-Machine} G'_i 1$.
 See subsection 5.2.3|p.77 for the employed operations.
2. We determine the set N_{acp} to be $F_{CFG-Actual-Controllability-Problems} P M_i N_{pcp}$
 A nonterminal $n_1 p_i n_2$ in N_{pcp} is in the set of nonterminals with actual controllability problems N_{acp} if and only if there is a state q of M_i that contains an item I that has an empty first rhs and that has a left hand side $n_1 p_i n_2$ and where p_i has an outgoing edge executing u in the plant P and q has no outgoing edge executing u in M_i .
3. We determine the optional LR(1)-CFG $G_{i,ec,opt}$ to be $F_{CFG-Remove} G_i N_{acp}$
 We remove the nonterminals N_{acp} from G_i and also remove all productions mentioning one these nonterminals. The result is equal to *None* if the initial nonterminal of G_i is in N_{acp} .
4. We return the obtained optional LR(1)-CFG $G_{i,ec,opt}$ as a result. Note, if $G_{i,ec,opt} = \text{Some } G_{i,ec}$, then $G_{i,ec}$ is included in G_i w.r.t. the sets of nonterminals, events, and productions. Also, $G_{i,ec}$ is either equal to G_i , which means that G_i has no further controllability problems to be removed, or $G_{i,ec}$ is strictly smaller than G_i .

By constructing the LR(1)-Machine for the given LR(1)-CFG, we “bind together” different initial marking derivations (for which the LR(1)-Parser would apply the same parsing rules) and can remove such a set of initial marking derivations together once we determine that they collectively exhibit a controllability problem. For least restrictiveness, it is crucial that such a set either contains only initial marking derivations exhibiting a controllability problems or, and this is our approach, that falsely removed initial marking derivations execute words that are still executable by retained initial marking derivations due to the two disambiguation steps before. We apply this operation in block 9|p.203 below to controller candidates given as LR(1)-CFGs.

(8) *Proposed Controller Synthesis Algorithm*: For a DFA plant P , a DPDA specification S , and a set of uncontrollable events Σ_{uc} such as those given in block 1|p.197, the algorithm proceeds as follows.

1. We obtain the initial LR(1)-CFG controller candidate G_0 according to block 3|p.198, block 4|p.199, block 5|p.200, and block 6|p.201.
2. We apply the following fixed-point iterator on the controller candidates G_i until a fixed point is obtained.
 - a) We obtain the (optional) LR(1)-CFG $G_{i,ec,opt}$ by removing controllability problems according to block 7|p.202.
 - b) If $G_{i,ec,opt} = \text{None}$, we return None and conclude that there is no satisfactory DPDA controller for S and P .
 - c) If $G_{i,ec,opt} = \text{Some } G_i$, we conclude that no controllability problems were removed and we convert this LR(1)-CFG into the resulting DPDA controller by using operations of our concrete controller synthesis algorithm discussed in chapter 5|p.61 and return this DPDA controller.
 - d) If $G_{i,ec,opt} = \text{Some } G_{i,ec}$ (where $G_{i,ec} \neq G_i$), we enforce nonblockingness by trimming $G_{i,ec}$ to obtain the next LR(1)-CFG controller candidate G_{i+1} that is used in the subsequent iteration.

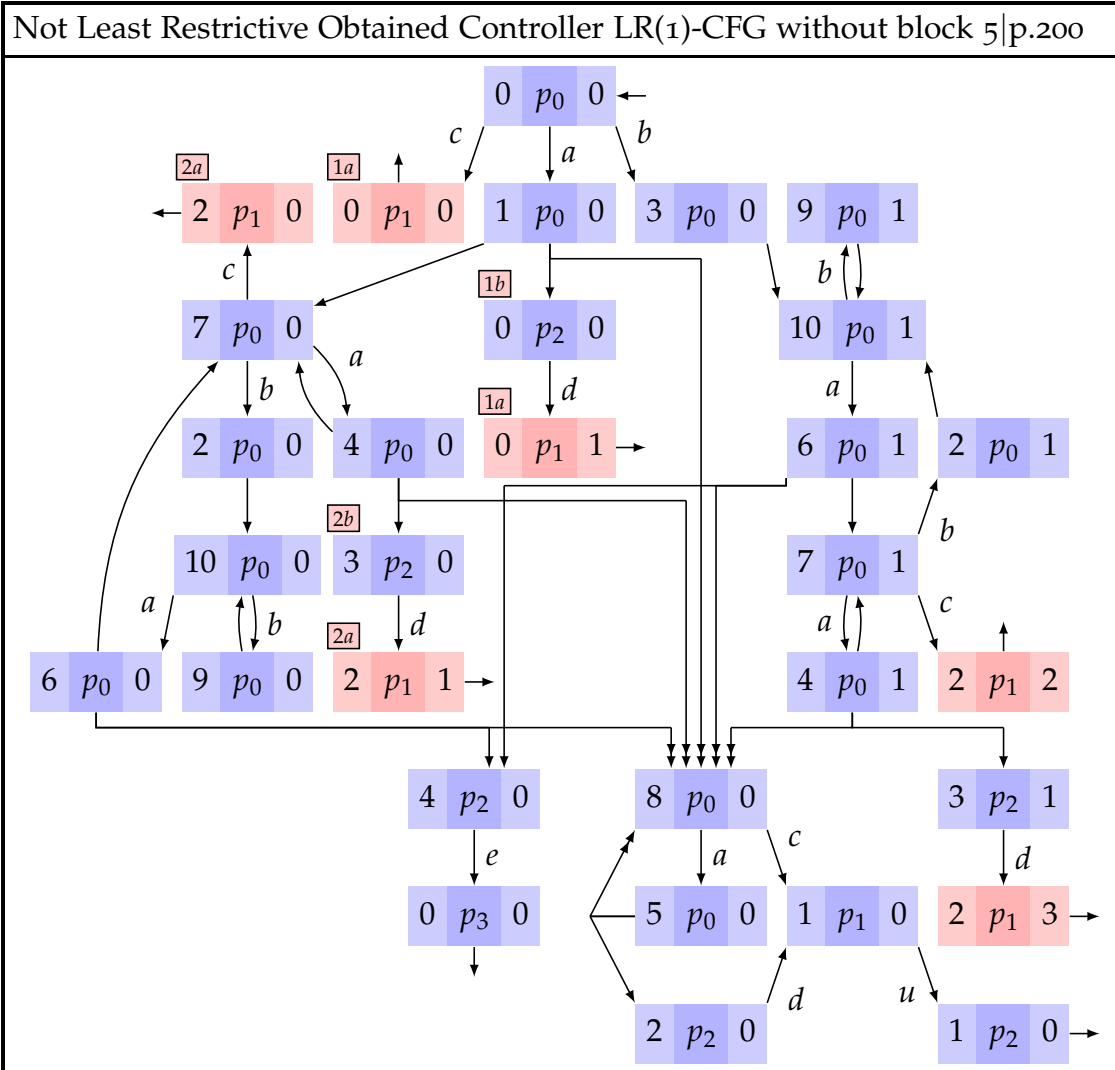
The termination of this procedure is immediate because the size of the sets of the nonterminals of the LR(1)-CFG controller candidates G_i strictly decreases in each iteration (considering for each G_i only the size of the subset of the nonterminals with potential controllability problems is also suitable for this purpose).

(9) *Resulting Satisfactory but not Least-restrictive Controller*: For our running example, we apply this proposed fixed-point computation to the initial LR(1)-CFG controller candidate given in block 6|p.201 (where we omitted the step in block 5|p.200 for now). For this LR(1)-CFG, we determine, see block 2|p.197 for our initial manual analysis of immediate and unavoidable controllability problems, that two nonterminals are to be removed in each of the first two applications of enforcing controllability.

- Iteration 1: Step a: Enforce Controllability
 - The nonterminal $0 p_1 0$ has a controllability problem because it is used for the initial marking derivation executing c , which is in the set of *immediate controllability problems* in (Eq. 8.1)|p.197.
 - The nonterminal $0 p_1 1$ has a controllability problem because it is used for the initial marking derivation executing $acud$, which is in the set of *immediate controllability problems* in (Eq. 8.5)|p.197.
- Iteration 1: Step b: Enforce Nonblockingness
 - The nonterminal $0 p_2 0$ is removed by trimming the LR(1)-CFG.

- Iteration 2: Step a: Enforce Controllability
 - The nonterminal $2 p_1 0$ has a controllability problem because it is used for the initial marking derivation executing ac , which is in the set of *unavoidable controllability problems* in (Eq. 8.2)|p.197.
 - The nonterminal $2 p_1 1$ has a controllability problem because it is used for the initial marking derivation executing $aacud$, which is in the set of *unavoidable controllability problems* in (Eq. 8.4)|p.197.
- Iteration 2: Step b: Enforce Nonblockingness
 - The nonterminal $3 p_2 0$ is removed by trimming the LR(1)-CFG.

In the third iteration, no further nonterminals are removed and the algorithm terminates. The removed nonterminals are marked in the following picture.



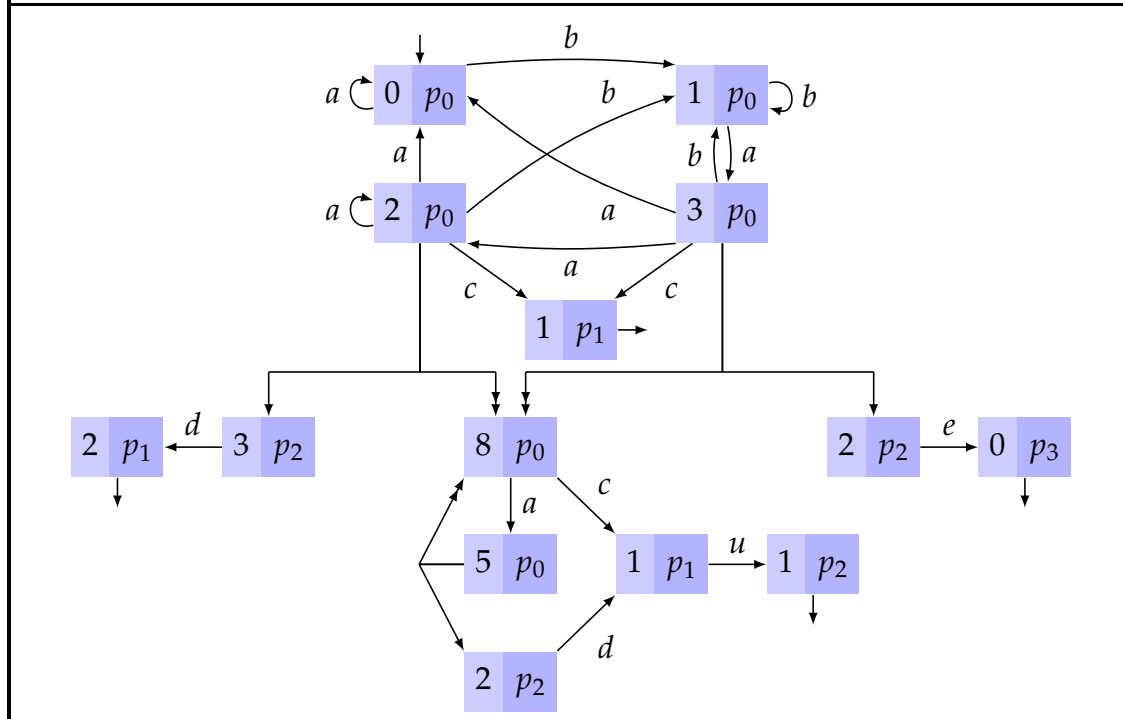
This resulting LR(1)-CFG controller candidate is not least restrictive because (for example) the word $abbac$ is falsely removed from the marked language of the

controller candidate as follows. Firstly, the word *abbac* is, according to block 1|p.197, in the marked language of the specification S . Secondly, the extension *abbacu* is a marked word of the controller candidate above.

The violation of least restrictiveness follows from the removal of the nonterminal $2\ p_1\ 0$ in step 2a that must be removed to remove *ac* but that also removes the word *abbacu* from the marked language. To resolve this problem, we require the application of the disambiguation step in block 5|p.200 in which we add a replica X of $2\ p_1\ 0$ that is reachable from $6\ p_0\ 0$ by executing c . This replica X is then retained in the controller candidate and guarantees that the word *abbacu* is retained in the marked language of the controller candidate. The satisfactory and least restrictive controller is depicted in the next block.

(10) *Resulting Satisfactory and Least-restrictive Controller*: We do not visualize the initial $LR(1)$ -CFG controller candidate that results from applying both disambiguation steps from block 5|p.200 and block 6|p.201 due to its size. After the third iteration our fixed-point computation in which no controllability problems are resolved, we ultimately obtain the satisfactory and least restrictive controller that is given as follows.

Least Restrictive Obtained Controller $LR(1)$ -CFG with block 5|p.200



For example, the word *abbac* discussed above is also a marked word of this controller candidate, which we computed using an extension of our prototype CoSy. A close inspection shows that the marked language of this $LR(1)$ -CFG is equal to the union of the sets of words given in (Eq. 8.3)|p.197, (Eq. 8.6)|p.197, and (Eq. 8.7)|p.197, which was the intended result.

The formal verification of the algorithm that is described and applied in this example remains for future work, but applications to various different examples including the one above are promising. There are some key issues to be verified in addition to the results obtained for our concrete controller synthesis algorithm from chapter 5|p.61. Firstly, the CFG controller candidate obtained upon termination of the fixed-point computation should satisfy the LR(1)-property to allow the translation into a DPDA. We believe that the disambiguation operation from block 6|p.201 and the removal of nonterminals and productions for enforcing controllability and nonblockingness readily preserve the LR(1)-property. The disambiguation operation from block 5|p.200 can invalidate this property but its application can be reverted whenever the existence of multiple remaining replicas results in the violation of the LR(1)-property (for simplicity, we left out this step in the example above). Secondly, a proof for the termination of the fixed-point computation is required. We expect that the reasoning given on this issue in the example will suffice. Thirdly, the CFG controller candidate obtained upon termination of the fixed-point computation should have no further controllability problems and should also satisfy the nonblockingness property to ensure that it is a satisfactory solution. Enforcing nonblockingness for CFG has already been formally verified and our proposed approach using the LR(1)-Machine for identifying controllability problems is directly related to the LR(1)-Parser constructed in our concrete controller synthesis algorithm from chapter 5|p.61 where it is also used to bind together initial marking derivations of the nonblocking LR(1)-CFG. Fourthly, we must verify that the employed disambiguations suffice to ensure least restrictiveness in particular for the operation for enforcing controllability.

If the formal verification of this proposed algorithm reveals unresolvable problems, we may still use the proposed algorithm as follows. Firstly, if it does not remove all controllability problems, we can integrate it into our concrete controller synthesis algorithm from chapter 5|p.61 by applying it in each of its iterations when an LR(1)-CFG is obtained. Secondly, if it is not least restrictive (presumably due to insufficient disambiguation), we can present alleged controllability problems to the user for manual validation (however, we propose the development of further disambiguations in this case or to remove fewer controllability problems). Thirdly, if it does not preserve determinism (presumably due to too extensive or inadequate disambiguation), we can present alleged controllability problems to the user for manual validation as well (however, we propose the development of weaker or more precise disambiguations in this case).

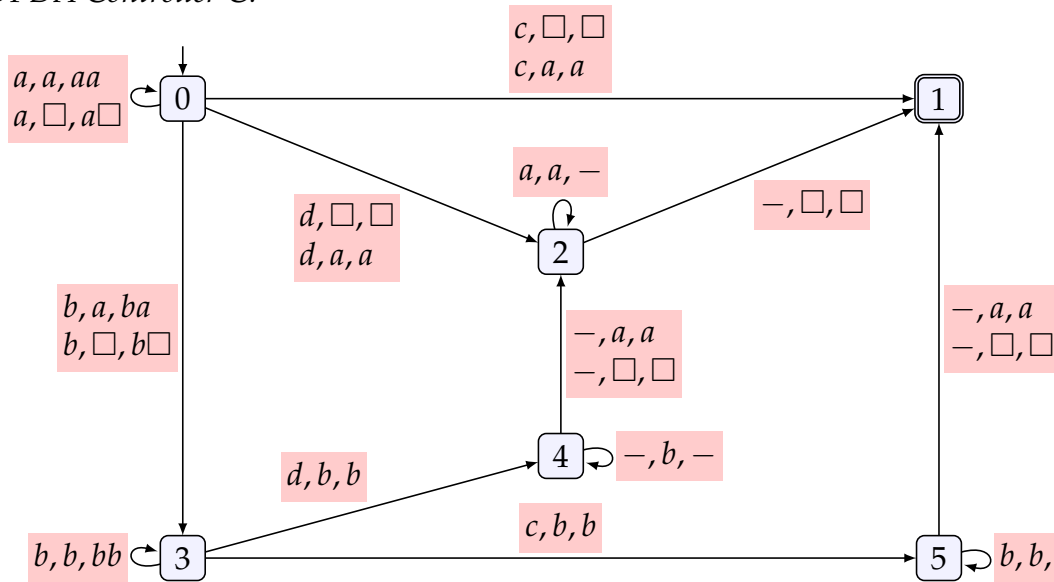
The proposed controller synthesis algorithm has been implemented in our prototype CoSy and back-to-back testing with our concrete controller synthesis algorithm from chapter 5|p.61 for various examples did not reveal differences in the marked and unmarked languages of the resulting DPDA controllers (for inputs where our concrete controller synthesis algorithm from chapter 5|p.61 also terminates). If the formal verification of this proposed algorithm shows that it is sound, least restrictive, and terminating, we believe that it is a reasonable replacement for our concrete controller synthesis algorithm from chapter 5|p.61 because it is more efficient and guaranteed to terminate.

8.2.3. Enforce/Verify Worst-case Execution Times

The worst case execution time (WCET) of a controller is given by the maximal amount of time it requires between two events and is required to ensure that the controller reacts timely upon any event of the plant to ensure overall functional correctness. Synchronous programming languages such as Esterel [39], Lustre [7], and SyncCharts [21] in which controllers can be implemented also rely on the existence of a WCET for a correct closed loop behavior. This requirement has also been mentioned in Par. *Further Controller Characteristics* p.166 for the setting of controller synthesis. For DFA controllers, the WCET is negligible because the step-relation can be encoded suitably and the controller does not have to perform complex computations. However, the DPDA from the following examples does not enjoy a WCET and, moreover, the example suggests that there is no procedure that translates a DPDA without WCET into an equivalent DPDA with WCET.

Example 8.9: «Internal Steps are Essential for DPDA»

DPDA Controller C:



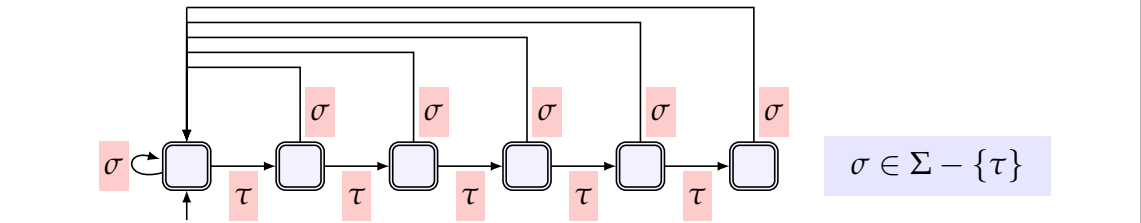
Discussion: The DPDA controller C has the marked language $\{ a^n b^m c b^m \cdot n, m \in \mathbf{N} \} \cup \{ a^n b^m d a^n \cdot n, m \in \mathbf{N} \}$. It counts the number of occurrences of the events a and b as it executes them until executing either the event c or the event d resulting in the stack $b^m a^n \square$. Then, for a word in the first set, the controller C can check that it executes the same number of events b as before by popping one b from the stack in each step (state 5). However, for a word from the second set, the controller C has to drop the prefix b^m from the stack (in state 4) to obtain the stack $a^n \square$ before being able to check that it executes the same number of events a as before by popping one a from the stack in each step (state 2). Hence, the controller has no WCET because for every natural number m there is a possible input such as $b^m d$ where the DPDA requires m steps before it can make a decision on whether or not to execute the event a at least once.

However, an actual encoding of this DPDA controller in a programming language may use more powerful stack-modification operations to guarantee a WCET by (a) realizing that the loop at state 4 in the DPDA above has the effect that all b -instances are removed from the top of the stack, by (b) maintaining a pointer to the point in the stack that is just underneath the unbounded number of b -instances (this pointer would be established when changing state from 0 to 3), and by (c) cropping the stack using this pointer when reaching state 5.

When considering the LR(1)-CFG, the LR(1)-Machine, and the LR(1)-Parser for the DPDA from the example above, we can indeed determine and use pointers along this idea. Intuitively, the LR(1)-CFG for this DPDA contains a production of the form $X \rightarrow YZ$ where Y can execute words in $b^m d$ and where Z can execute a following event a . The pointer mentioned above should be established for the look-ahead event a originating from Z when the first event b (or the event d) is executed using the nonterminal Y . When the identified look-ahead event a is then observed later on, which means that the nonterminal Z will not produce further events, the stack should be cropped using the established pointer. In the example, this avoids the application of an unbounded number of reduce rules that silently revert steps applied to the nonterminal Z and its derivatives. However, these preliminary results require further investigation.

Alternatively a WCET of k can be enforced on a computed controller candidate C by (a) eliminating/merging adjacent edges not executing events, by (b) adapting all edges not executing events to execute the fresh event τ , by (c) constructing the product automaton of C and a DFA such as the one in the figure below, and by (d) reverting τ -executing edges of the resulting automaton back into edges executing no event.

Figure 8.1: «DFA Used for Enforcing a WCET of 5 on a DPDA»



However, the least restrictiveness of this procedure depends on the non-trivial step (a) from above for reducing sequences of internal steps. Also note that step (c) of this procedure may invalidate nonblockingness and controllability because it may restrict the marked language. Hence, the employed controller synthesis algorithm must then be restarted on the obtained automaton to ensure the satisfaction of these two properties.

Also, while we are not aware of suitable optimizations of the definition of the canonical LR(1)-Parser, the desired adaptations enforcing a WCET would also result in more efficient parsing as intermediate reduce steps could be omitted.

8.3. FUTURE WORK

We discuss several envisioned enhancements, extensions, and applications of the contributions presented in this thesis. In particular, we strive to enhance and optimize the concrete controller synthesis algorithm presented in chapter 5|p.61 and its prototype implementation CoSy discussed in section 7.4|p.154 in several directions. Also, we will continue our work from the previous section building upon the promising results presented.

8.3.1. Further Enhancements of the Contributions

We consider enhancements of our concrete controller synthesis algorithm (and also its prototype implementation CoSy) regarding several important aspects of the inputs, the performed computation, and the outputs. In particular, we consider workflows and formalisms for the definition of inputs of our concrete controller synthesis algorithm to ease applicability and to allow for stronger specifications, we discuss aspects of the concrete controller synthesis algorithm such as efficiency, trustworthiness, termination, and a supplementary workflow integration, and we target the problem of minimizing the memory footprint of the synthesized controller when being deployed.

— Inputs of the Concrete Controller Synthesis Algorithm

We now continue our discussion from section 7.1|p.143 and discuss problems regarding the (compositional) definition of the plant and the specification and the use of specifications stronger than DPDA for the synthesis algorithm.

We will utilize a more relaxed CSP-style synchronization operation (see [162]) that synchronizes the steps of a DPDA G_1 and of a DFA G_2 only for the events in an additionally provided set Σ_S , written $G_1 ||_{[\Sigma_S]} G_2$, and that allows for interleaving of steps for events not in Σ_S . Such an operation will ease applicability of our synthesis algorithm because it allows for simplified plant and specification automata (for example, the self loops for various additional events in Example 7.2|p.147 could be omitted). By also using this synchronization operation, we will define the plant DFA as the determinization of the product automaton of a set of (possibly nondeterministic) FSA. Note that the usage of this synchronization operation *within* the concrete controller synthesis algorithm as a replacement for $F_{DPDA-DFA-Product}$ will impact our entire formalization (including the operation *inf* defining the corresponding synchronization operation on DES instances). Additionally, to avoid the model explosion problem caused by the application of the synchronization operation, we will investigate whether the computation of the synchronization automaton can be avoided (see [228] for a DFA-based synthesis approach for State Tree Structures that avoids the model explosion problem in this way). Moreover, we will develop a reasonable input notation for DPDA and DFA to simplify their definition, which was already required for Example 7.4|p.151 where we automatically generated different DPDA specifications and DFA

plants for different sets of parameters for use in section 7.5|p.157. To further increase the expressiveness of the specifications, we will select a logic such as LTL, CTL, or CaRet [13] to be able to state additional relevant properties that are to be satisfied by the closed loop. Then, given the controller candidate in the form of a DPDA or in the form of an $LR(1)$ -CFG, we will develop suitable algorithms for enforcing these properties. Moreover, instead of using the DPDA as a positive specification (as it is customary in supervisory controller synthesis), we will evaluate whether the use of a DPDA for the definition of undesirable behavior is beneficial. Note that we can readily apply our algorithm in this case by applying the well-known complement operation on this negative DPDA (see [168, Theorem 7.29, p. 289]) but that we cannot use a positive and a negative DPDA specification at once because in this case the synchronous product of the positive DPDA and the complement of the negative DPDA would not be a DPDA as required. Moreover, as for the plant, where we may use nondeterministic FSA and translate them to DFA before applying our algorithm, we may use a PDA specification that is nondeterministic and convert it to a CFG. This conversion preserves the marked language of the PDA also in this nondeterministic case and, which warrants further analysis, removes the nondeterminism in some cases. While not every PDA has an equivalent DPDA counterpart, we will analyze the limitations of this conversion. Also, we will investigate whether the synthesis procedure can be adapted to handle and preserve stack-elements with an inner structure representing for example multiple values at once. Similarly, we will determine whether compound operations (given for example by EDPDA edges) that can be represented by multiple DPDA edges and that are used in the input specification can be reestablished in the DPDA that is obtained from the intermediate $LR(1)$ -CFG. This would be beneficial as it would help to reduce the size of the DPDA controller candidates and it would help to retain the connection between the provided specification and DPDA controller candidates that occur during the fixed-point computation. Finally, we note that an integrated workflow accompanied with a tool environment requires also support for the validation of the provided plant and the provided specification. Such validation will be supported by producing diverse subsets of the marked languages, which can be obtained for example by determining words derivable by all distinct subsets of productions of the equivalent $LR(1)$ -CFG. Each word of such a subset then corresponds to a scenario that can be inspected.

— Operations of the Concrete Controller Synthesis Algorithm

We will continue our work on the operations of the concrete controller synthesis algorithm to increase its *efficiency* with respect to time and space, to increase *trustworthiness* of all operations of its prototype implementation CoSy, to ensure *termination* or at least to realize patterns that may indicate nontermination, and to provide *additional feedback* during the fixed-point computation to allow the user to check that the fixed-point computations proceeds as expected.

We already provided the first steps in subsection 8.2.2|p.193 towards an al-

algorithm that is more efficient compared to our concrete controller synthesis algorithm from chapter 5|p.61. In the fixed-point computations carried out by both algorithms, we employ operations on sequences of only slightly changed EPDA, CFGs, and Parsers. Hence, to increase efficiency, we will adapt the employed operations to adapt results from earlier applications to the slightly changed inputs. For example, a certain part of the DPDA controller candidate may have no controllability problems after iteration i and does not need to be reconsidered in subsequent iterations. We will verify that the operation $F_{DPDA-Enforce-Accessible,Opt}$ used at the end of the operation $F_{DPDA-Enforce-Nonblocking,Opt}$ can be skipped because the LR(1)-Parser constructed in $F_{DPDA-Enforce-Nonblocking,Opt}$ is accessible and the subsequent translation of this LR(1)-Parser into a DPDA preserves accessibility. This conjecture is supported by a prototype-based evaluation using CoSy that we carried out for this purpose. This analysis also showed that we can expect a significant gain in efficiency when actually skipping the application of $F_{DPDA-Enforce-Accessible,Opt}$. Similarly, we will attempt to adapt the operation $F_{DPDA-Reduce-Controllable}$ to be able to skip the application of $F_{DPDA-Enforce-Accessible,Opt}$ in $F_{DPDA-Reduce-Controllable}$ as well by ensuring that every auxiliary state that is introduced is always accessible (see section 5.3|p.82). Moreover, when implementing our prototype CoSy, we determined various optimizations of building blocks of our concrete controller synthesis algorithm. Some of these optimizations are given by straightforward implementations using multithreading. Other optimizations avoid recomputations for unchanged inputs using caches, compute the outputs differently as the operation $F_{SDPDA \rightarrow LR(1),Rec}$ presented in subsection 8.2.1|p.188, additionally reduce the size of the resulting EPDA, CFGs, and Parsers by removing inaccessible parts or by merging adjacent edges, productions, and rules, or exploit observations such as the one above on the application of $F_{DPDA-Enforce-Accessible,Opt}$. We will implement our algorithms in C++ by adapting the libFAUDES [32] plugin mentioned in section 7.4|p.154 because our prototype based evaluation from section 7.5|p.157 revealed that, when allocating more memory than required for the Java runtime environment, the default memory handling of Java causes a significant increase of runtime.

The above discussed optimizations of building blocks (and even more so the discussed extensions) are on their own not trustworthy and their integration and use in our prototype CoSy henceforth requires further formalization and verification to reestablish a high degree of trustworthiness. This problem of maintaining trustworthiness upon any kind of adaption is also not significantly alleviated by the back-to-back testing with the formally verified original operations that we typically employ in initial stages of change development. In fact, the manual implementation process from the Isabelle-based formalization of the various building blocks to Java code is obviously subject to failure, but, as of now, the Isabelle support for converting formalizations of building blocks from Isabelle into functional programs is not adequate due to lack of documentation, lack of automatic support, lack of generality (automatic handling of enough language constructs and operators), and lack of code optimizations (such as

multithreading). We will use the Isabelle code generation techniques to obtain a reference implementation once it reached a sufficient level of maturity.

We argued in section 5.5|p.87 that we believe that our concrete controller synthesis algorithm from chapter 5|p.61 already terminates for all reasonable DPDA specifications, which are not specifying the execution of an unbounded number of uncontrollable events according to the elements of a previously established stack. However, we will determine adaptations of our concrete controller synthesis algorithm ensuring termination such as the synthesis algorithm discussed in subsection 8.2.2|p.193 for which termination is not in doubt. When considering various problem instances such as for example Figure 5.9|p.88 where our proposed concrete controller synthesis algorithm did not terminate, we observed that some controllability problems are not removed when the source of uncontrollability is a stack element that may be hidden under an unbounded number of other stack elements. The application of the fixed-point computation then resulted in a sequence of obtained controller candidates that are all having a *similar* instance of the “same” controllability problem. Intuitively, our algorithm fails in these cases to remove *unavoidable* controllability problems as discussed in subsection 8.2.2|p.193. From the considered example, we believe that the controllability problems that are undesirably preserved between two successive controller candidates can be identified in the form of preserved substructures. If the least restrictiveness of the removal of these substructures can be formally verified, we may obtain a terminating algorithm by integrating this check-and-remove procedure. Otherwise, the user may take action based on well-formulated notifications on these preserved controllability problems. However, the use of such an additional check would also require an alteration of our proposed abstract controller synthesis algorithm.

We believe that the synthesis-time analysis of the marked words that the controller synthesis algorithm removes from the controller candidates will improve the workflow of controller synthesis. This information will assist the developer of the plant and the specification in their validation, will allow the developer to make substantiated estimations on the termination of the algorithm, and will enable the developer to manually adapt the specification to resolve identified controllability problems with the benefit of maintaining a small controller candidate (confer to Example 7.4|p.151 where the ultimately retained subset of edges of the specification could be determined using this workflow).

→ Outputs of the Concrete Controller Synthesis Algorithm

In Par. *Further Controller Characteristics*|p.166, we have already discussed the need for a resulting DPDA controller that does not require an excessive amount of memory for its transition relation, in addition to the memory that is required for the stack, when being deployed. In Par. *Additional Property of Minimality for DPDA*|p.39 and in Par. *Additional Property of Accessibility for DPDA*|p.40, we described the notions of size-minimal controllers, language-minimal controllers, accessible controllers, and controllers that are accessible in the closed loop. To reduce the

memory footprint of the DPDA controller, we are interested in size-minimal controllers, but the known concrete controller synthesis algorithms generate language-minimal controllers. This is due to the fact that they all start with a candidate that is either the plant P or (if available) the synchronous product of the plant P and the specification S . See for example the abstract and concrete algorithms from [281] where specifications and marked languages are not used and the abstract and the concrete algorithms from this thesis given in chapter 4|p.45 and chapter 5|p.61. For example, if the specification S does not forbid any behaviors of the plant P and the plant already satisfies the nonblockingness property, then any controller C satisfying $epda\text{-}to\text{-}des\ P \leq epda\text{-}to\text{-}des\ C$ is a least restrictive satisfactory controller. However, the smallest controller w.r.t. the marked and unmarked language is P and the smallest controller w.r.t. the number of states and edges is S (assuming that S contains only one state and one self-loop for each event).

Language-minimal DFA controllers C can be translated into size-minimal DFA controllers with equivalent marked language by standard minimization algorithms. However, we are interested in a different equivalence notion here: the obtained size-minimal DFA controller should result in the same closed loop language instead. A straightforward approach to accomplish this goal should be to add all words to the marked language of C that would be removed by the synchronous composition with the plant and to minimize the obtained DFA afterwards. Of course, we would not add words with events that are not occurring in C . In the discussed example above, this would modify the controller that is equal to P into the controller equal to S resulting in the described minimal DFA controller of one state and one self-loop for every event.

However, there are no standard minimization algorithms for DPDA: there are decision procedures for DPDA equivalence [318, 331, 330] and we use a similar bisimulation-based approach for LR(1)-CFG that we integrated in our prototype CoSy for testing purposes, but further heuristics for DPDA minimization are called for. Nonetheless, we will attempt to apply the outlined approach for DFA also for a DPDA C by modifying the DPDA controller along the same lines as for the DFA above and by enforcing nonblockingness on this obtained DPDA. We will then analyze whether the resulting DPDA is much smaller compared to the initial DPDA C .

Besides these general procedures to minimize intermediately obtained LR(1)-CFG or DPDA controllers, we note that the size of the LR(1)-Machine and the LR(1)-Parsers that are constructed in subsection 5.2.2|p.73 depends on the input LR(1)-CFG used in this step. Therefore, we will, on the one hand, attempt to determine alternative approaches that do not rely on the conversion via LR(1)-CFG and, on the other hand, develop operations such as our mentioned bisimulation-based operations to obtain an equivalent LR(1)-CFG that results in an LR(1)-Parser of reduced size. Moreover, size-minimal Parsers are also important in the field of parsing theory because the Parsers for programming languages can become quite large as well. The construction of the (canonical) LR(k)-Parsers

from [189] has been investigated in [193, 101, 102, 4, 3] for possible reductions with respect to the size resulting in the notions of LALR(k)-CFG and SLR(k)-CFG. The notion of LALR(k)-Parsers reduces the size of the (canonical) LR(k)-Parser by merging all states of the LR(k)-Machine with same sets of item-cores (the core of an item is the production to which it belongs). However, LALR(k)-Parsers (and mixed versions such as LA(l)LR(k)-Parsers for $l \leq k$) do not satisfy the nonblockingness property because they may apply additional reduce rules before determining an erroneous input (confer, [325, Volume II, p. 66]). To accomodate for this problem, we will determine a mechanism that is to be used by the runtime environment for the DPDA controller. This mechanism should ensure that the event observed in such reduce rules is not actually permitted before reaching the end of the finite sequence of reduce rules that is then followed by an application of a shift rule. Also, if this shift rule is then missing, the runtime environment would have to be able to execute a rollback of the applied reduce rules to allow for a different evolution. Similarly, SLR(k)-Parsers can be used but they may detect errors even later than LALR(k)-Parsers [325, Volume II, p. 73].

Finally, based on a set of patterns where each pattern is a word of stack elements that are expected in that order on the stack, a runtime environment for DPDA can reduce the required size of the memory by representing a stack by a list of patterns that are associated with a number of repetitions. Besides this approach that is inspired by our considerations on enforcing a worst case execution time in subsection 8.2.3|p.207, the runtime environment may use lossless compression algorithms such as the one of David Albert Huffman [169] to reduce the memory consumption of the stack.

8.3.2. Further Extensions of the Contributions

We now consider further problems in the context of supervisory controller synthesis that we have not been handled by formally verified algorithms in this thesis. On the one hand, these problems are caused by the use of DPDA controllers and specifications and, on the other hand, we simply consider variations of the supervisory control problem that have been targeted in the past for the setting of DFA controllers and specifications.

We have already considered the problem of enforcing a worst case execution time (WCET) in subsection 8.2.3|p.207 in the context of DPDA. This work will be continued along the presented lines to derive runtime environments for DPDA controllers with support for compound stack operations that are executed instead of a possibly unbounded sequence of internal steps. To this end, we will attempt to ensure that there is a maximal number of steps between any two uncontrollable events determining a WCET of the controller.

The risk of a stack overflow at runtime should be minimized as discussed in Par. *Unbounded Stacks of DPDA*|p.143. For controllers where no upper bound on the depth of the stack can be determined, we will continue the development of stack-usage patterns to reduce the depth of the stack as much as possible

and, additionally, we will use estimated rates for pushing and popping elements to and from the stack or a probabilistic extensions of the DFA plant and the DPDA controller to obtain a probabilistic model in which the likelihood of out-of-memory errors can be bounded.

We have not handled the plant identification problem [349] in this thesis, but the use of an inadequate plant model or robustness issues such as perturbations, drifts, or imperfect communication can result in an actual plant not behaving according to its model from the perspective of the controller. To approach robustness against these adverse effects on the closed loop, we will determine whether error recovery strategies from parsing theory can be used to resync the DPDA controller and the DFA plant when detecting an error. Such error recovery algorithms may use the notion of the minimum distance errors to determine the minimal number of events to be changed in the currently parsed unmarked word to obtain a prefix of a marked word and to change the configuration of the controller accordingly.

The problem of constructing a controller that selects controllable events to be executed to drive the plant into a marking state has been discussed in Par. *Optimal Control*|p.165. This problem can also be extended by assigning costs or rewards to for example events or states and by requiring that the actual derivations that end in marking states have minimal costs. The computation of an optimal schedule that the DPDA controller is then supposed to follow is an open problem (the approach proposed in [143, Chapter 7, pp. 90–] by Christopher Griffin does not properly incorporate the stack as in the proposed synthesis procedure). Moreover, we will investigate whether a probabilistic model can be used to reasonably restrict the adversarial capabilities of the plant issuing uncontrollable events.

It is a well-known limitation of deterministic context free languages (that is, the class of marked languages of DPDA) that they are not closed under intersection. This implies that there is no operation that constructs the synchronous product of two DPDA specifications or of two DPDA controllers and that results in a DPDA in all cases. For the specifications, this impediment hampers our ability to harness a compositional construction of the overall specification because all stack accessing parts must be given collectively in a single DPDA. For the controllers, the problem is similar; in a horizontal decomposition approach (see Par. *Synthesis Considering Horizontal Composition (Modular and Distributed Control)*|p.174) where each of a number of (modular) plants is controlled by one (modular) controller, the synchronous product of these plants (the global plant) is formally controlled by the synchronous product of the controllers (the global controller). The central problem of this approach is, also in the setting of DFA specifications and DFA plants, that the synchronous composition of the controllers yields a satisfactory and least restrictive controller w.r.t. the global plant. The benefit of modular modelling and modular controller synthesis is rescinded when the global plant and global controller obtained must be constructed explicitly for further analysis and restricted to obtain the final controller. Hence, syntactic/type-based guarantees for a property-preserving composition are considered in the literature. However, we expect that the use of DPDA controllers will aggravate the problem

of determining suitable guarantees and, also, only one of the controllers may be a DPDA, which limits its general applicability to sets of plants where only one plant is restricted by a DPDA specification. Visibly Pushdown Languages (VPL) are defined as the class of languages of Visibly Pushdown Automata (VPA) in [16] and also games for VPL have been considered in [226]. The class of VPL is highly interesting because it is closed under union, complement, and intersection and VPA can be determinized. However, some languages that are in DCFL are not in VPL such as $\{ a^n b a^n \cdot n \in \mathbf{N} \}$ but the slight alteration $\{ a^n b c^n \cdot n \in \mathbf{N} \}$ is a VPL. Intuitively, each event can be used either for pushing a symbol to the stack or for popping a symbol from the stack (the end of stack marker \square is never popped and internal events do not alter the stack). This partitioning into the three sets then restrict the synchronous composition sufficiently to ensure the closure property for intersection. We will analyze whether our concrete controller synthesis algorithm can be adapted to deterministic VPA to allow for compositional specification (using multiple VPA specifications), for more complex plants (using multiple VPA plants), and for compositional control (using multiple VPA controllers). Pushdown Tree Automata (PDTA) [147] are extensions of PDA operating on trees instead of on words. Further extensions such as Visibly Pushdown Tree Automata (VPTA) [75], Visibly Tree Automata with Memory (VTAM) [85], and Nested Word Automata (NWA) [14, 15] combine tree based inputs with the event partitioning of VPA. The synthesis of VPTA, VTAM, or NWA as controllers may be the next step once supervisory controller synthesis has been extended to VPA as suggested above.

We will investigate whether controller synthesis for other formalisms such as Markov Decision Processes [294], probabilistic timed automata [206] and hybrid CCS [308] can be enabled by using DFA or VPA as a manageable abstraction formalism for the mentioned more complex formalisms. This idea continues applications of DFA as abstractions of physical plants in hierarchical control.

Similarly to horizontal decomposition as just discussed, we are interested in hierarchical control for which we discussed related work in Par. *Synthesis Considering Vertical Composition (Hierarchical Control)* [p.175]. Initial attempts have already been made in [236] to extend our work to the scenario with DPDA specifications. However, the proposed approach is only combining well-established results on hierarchical control with our proposed concrete controller synthesis algorithm and is still limited by using DPDA instead of for example VPA.

Supervisory controller synthesis for partially observable plants (see Par. *Synthesis Considering Observability and Diagnosability* [p.173]) has been considered for DFA plants and DFA specification already in [279, 222, 256]. We will analyze how these past developments can be recovered in the setting of DPDA specifications and DPDA controllers. Specifically, we believe that the above proposed CSP-style synchronization operation (see Par. *Inputs of the Concrete Controller Synthesis Algorithm* [p.209]) may already be appropriate to allow the plant to perform unobservable events without synchronization with the controller, which is therefore not able to detect the execution of these events.

We discussed the explicit handling of time already in Par. *Synthesis Considering Discrete and Continuous Time* [p.173], but, implicitly, each event may be associated with a certain delay. Besides the task of minimizing the WCET considered above (where internal events are assumed to require some time that may add up to a value above some acceptable threshold), discrete time has been considered already in [219, 60]. However, we believe that meaningful results will be built upon optimal control resolving nondeterminism through explicit choice (see above) for controllable events or by resolving nondeterminism using a probabilistic model for uncontrollable events. Nondeterminism in the models such as in probabilistic timed automata often leads to worst case resolutions of nondeterminism that are unrealistic and result at the same time in unsatisfactory estimates on possible behavior. Resolving nondeterminism (by either eliminating choices or by making certain choices unlikely) is then a well-established approach to obtain better analysis results. However, systems with infinite state spaces such as DPDA pose a problem to the existing techniques. We will apply or adapt existing techniques for formalisms with infinite state spaces (such as hybrid automata [11]) to determine algorithms to check formulas from probabilistic timed logics such as PTCTL [207] or, as it appears more suitable for our context of DPDA closed loops, by considering an extension of CaRet [13] by time and probabilities. Also, the definition and guarantee of a notion of least restrictiveness in these settings when resolving violations of desired properties is a major obstacle.

Chapter 8 | p.163
(Related, Ongoing, and Future Work)

We discussed related work to point out problems similar to supervisory control and alternative approaches to their solution, variations of supervisory control incorporating further aspects, a previous attempt at solving the supervisory control problem for DPDA specifications, available tool-support for supervisory control, and available Isabelle-support for verification of algorithms on the involved formalisms.

We presented three novel approaches that lack formal foundation as of now. Firstly, we presented a conversion operation from DPDA into $LR(1)$ -CFG that is more efficient compared to the algorithm presented in chapter 5 [p.61]. Secondly, we provided an algorithm for enforcing controllability based on an $LR(1)$ -CFG to ensure termination also for unreasonable DPDA specifications and to obtain a more efficient concrete controller synthesis algorithm. Thirdly, we discussed steps towards enriching the runtime environment used for the controller with suitable annotations to ensure that the controller performs at most n internal steps in succession.

We identified important and promising future extension of our contributions according to the presented related and ongoing work as well as according to various observations throughout this thesis.

In the next chapter, we conclude on the results of this thesis.

Summary and Conclusion

In this chapter, we first concisely restate our research problem in its broader context. Building on this foundation, we then summarize our approach and contributions from the perspectives of computer science and control theory. We then evaluate the relevance of our contributions from standard and proposed application areas. As ongoing work, we suggest numerous adaptations and extensions for improving the applicability of our results. Moreover, we identify additional recommendations for future work. Finally, we conclude with an overall assessment.

—• *Research Domain and Problem*

As discussed in more detail in chapter 1|p.1, we contributed to the research domain of *fully automatic supervisory controller synthesis for discrete event systems* in which the *supervisory control problem* initially appeared in the seminal work in [282]. This problem from control theory similarly occurs in computer science [81, 121] where controllers are to be constructed for software systems.

Generally, the manual development of controllers and their synthesis using fully automatic algorithms are competing approaches with specific advantages and limitations as discussed in chapter 1|p.1. Most importantly, the applicability of fully automatic synthesis algorithms for solving the supervisory control problem is limited by an insufficient expressiveness of the formalisms used for plants and specifications regarding the sequences of events they can describe. In fact, synthesis algorithms developed in the past are variations of the algorithm provided in [282] where plants and specifications are required to be given by DFA. Advances towards Petri net models revealed unsolvability of the synthesis problem in this setting (see subsection 8.1.6|p.176) and an earlier synthesis algorithm for DFA plants and DPDA specifications proved to be erroneous (see subsection 8.1.7|p.180).

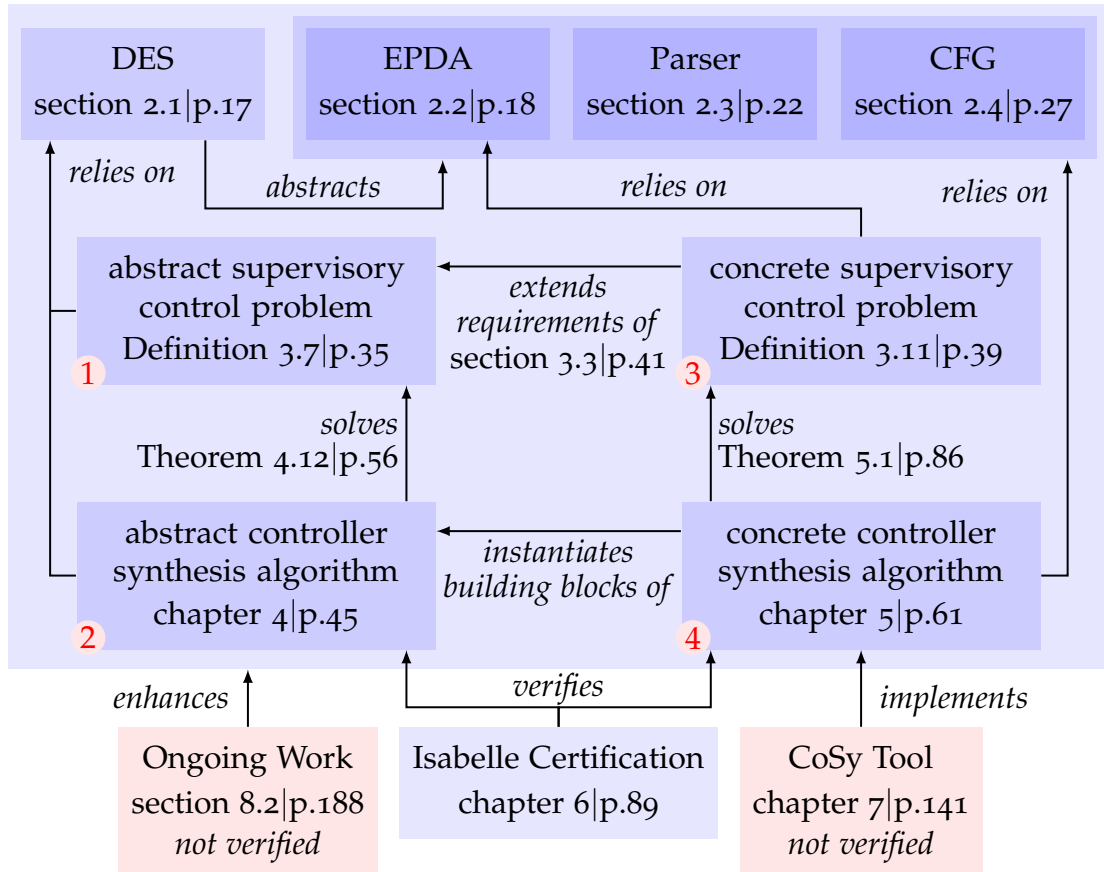
Hence, our main research aim was the development of a fully automatic controller synthesis algorithm with increased expressiveness of the involved formalisms.

→ *Main Contribution, Approach, and Challenge*

Our *main contribution* is the concrete controller synthesis algorithm (see chapter 5|p.61), which solves our concrete supervisory control problem (see Definition 3.11|p.39) by synthesizing a DPDA controller for a DFA plant and a DPDA specification. This algorithm is supported by its prototype implementation CoSy (see chapter 7|p.141 and [304]) and its formal Isabelle-based certification for which we developed a foundational Isabelle framework (see chapter 6|p.89 and [303]).

Our *approach* to achieve this main contribution consists of the four steps 1 – 4 that are given and discussed in detail in the following figure.

Figure 9.1: «Overview of the Contributions and the Approach of This Thesis»



In step 1, we formalized the abstract supervisory control problem following [281] on the basis of our abstractions of DFA and DPDA called discrete event systems (DES). These DES recover both the marked and unmarked languages generated by the automata, which contain the desirable and actual sequences of events, respectively. The notion of DES is sufficient to state the most important *semantical* properties (see below) in our abstract supervisory control problem.

In step 2, we formally verified that our *abstract* (fixed-point) controller synthesis algorithm (see chapter 4|p.45) indeed solves the abstract supervisory control problem. This abstract algorithm is novel in the sense of operating on DES to properly capture the steps of enforcing controllability and nonblockingness.

In step 3, we relied on the notion of least restrictiveness of our abstract supervisory control problem for the formal definition of our concrete supervisory control problem. Moreover, we identified, besides controllability, nonblockingness, and least restrictiveness, several additional syntactical and semantical properties to be satisfied, such as the absence of livelocks in the closed loop, and that are desirable, such as the existence of a worst case execution time (WCET).

In step 4, we constructed our concrete controller synthesis algorithm as an instantiation of our abstract controller synthesis algorithm (see chapter 4|p.45) by determining instantiations for each of its abstract building blocks. The verification of the resulting concrete synthesis algorithm proceeds in two steps. As a first step, we verified that the concrete operations used for the instantiations satisfy the interface specifications of the abstract building blocks. This ensures that the concrete algorithm inherits the semantical properties enforced by the abstract algorithm. As a second step, we verified that the concrete operations also enforce the additional semantical requirements stated in the concrete supervisory control problem such as the absence of livelocks in the closed loop, which completes the certification of the concrete algorithm.

We now highlight one challenge we overcame, which was the instantiation of the building block of the abstract algorithm that enforces nonblockingness on DES. The corresponding concrete building block operates on DPDA and enforces nonblockingness, the absence of deadlocks, the absence of livelocks, and accessibility (see section 5.2|p.65). The difficulty of this instantiation problem stems from the potentially infinite state space of a DPDA, which implies that all algorithms that inspect only a bounded prefix of the stack of a given DPDA configuration are insufficient due to ambiguity: Intuitively, a configuration may suffer from a blocking problem due to a certain stack element that is covered by an unbounded number of stack elements and that potentially remains concealed when considering only a bounded prefix of the stack. To meet this challenge, we employed a *symbolic* removal of blocking problems: The algorithm firstly disambiguates the input DPDA model and, secondly, removes safely and entirely a finite set of elements from the model to prevent the reachability of the potentially infinite set of configurations that exhibit blocking problems. By and large, the recurring problem of defining for *disambiguation* highlighted here was *the main theme* of this part of research.

→ Applications of our Concrete Controller Synthesis Algorithm

Our concrete algorithm is beneficial for applications in control theory and computer science for the synthesis of controllers for cyber-physical and software systems because our concrete algorithm permits DPDA specifications whereas the earlier procedures only permit DFA specifications, which are less expressive and which are unable to capture relevant classes of behaviors.

Also, the use of DPDA specifications proved useful in the domains of program analysis and compiler optimization [16] where the potentially infinite state space of a DPDA still limits the fully automatic analysis (for instance using model

checking). Hence, our symbolic, translation-based controller synthesis algorithm lays a new path for extending analysis and synthesis by equipping the involved models with a stack while checking more expressive specifications using for example the temporal logic CaRet. In general, since our controller synthesis algorithm extends the capabilities of previous synthesis algorithms, it has applications in the domains of manufacturing, robotics, chemical process control, protocol design in communication networks, feature interaction management in telephony, and fault diagnosis [70]. Moreover, the problems of controller synthesis and analysis of closed loops occurs similarly in various examples in computer science such as in submodule construction (which states a similar synthesis problem), in the formalism of input/output automata (which requires instances to satisfy a notion similar to controllability called *input enabledness*), and in adaptive systems (where the usage and switching of controllers is a cornerstone in the employed closed loop methodology). Further applications are discussed in section 8.1|p.164, section 7.2|p.146, and by our examples provided in section 7.3|p.147.

To apply our synthesis algorithm in settings as described above, we require a DFA plant model and a DPDA specification model. Firstly, this may require an over-approximation of the behavior of the actual plant to obtain a DFA plant model. Secondly, we discussed several specification patterns exploiting the stack-based capabilities of DPDA (see section 7.1|p.143).

Our prototype CoSy, which is an implementation of our controller synthesis algorithm, shows promising efficiency albeit subject to further optimization according to our evaluation (see section 7.3|p.147 and section 7.5|p.157). The applicability and efficiency of CoSy was evaluated by considering several examples from manufacturing employing DPDA specifications (in chapter 7|p.141).

Also note that the semantical properties of nonblockingness, absence of deadlocks, absence of livelocks, and accessibility that are enforced by the building block described in the previous paragraph are not specific to control theory (see section 5.2|p.65). Hence, we propose to evaluate their value to other application domains (see section 8.1|p.164) with stack-utilizing models as future work.

— *Adaptations to Mitigate Limitations of our Concrete Controller Synthesis Algorithm*
We identified limitations of our concrete controller synthesis algorithm that hinder its applicability. Note that these limitations have an impact on our approach to construct and analyze controllers with its three enabled use-cases explained in chapter 1|p.1 as it is centered around this concrete controller synthesis algorithm. We presented ongoing developments of suitable adaptations to mitigate these limitations, which reached a reasonable level of maturity. However, these adaptations lack formal foundation as of the writing of this thesis and, moreover, additional future work is called for to address further limitations (see section 8.2|p.188 and subsection 8.3.1|p.209 for a complete discussion).

Our formally verified synthesis algorithm from chapter 5|p.61 is only claimed to be terminating for all reasonable specifications (as substantiated by suitable examples in section 5.5|p.87). To address this problem, we defined a *second*

synthesis algorithm in subsection 8.2.2|p.193, which follows the disambiguation procedure for enforcing nonblockingness on DPDA more closely. This alternative synthesis algorithm is more efficient and guaranteed to terminate.

Besides this (global) optimization, we also developed various (local) enhancements of the building blocks of our formally verified concrete controller synthesis algorithm to increase its efficiency (see section 8.2|p.188).

The usage of DPDA to express more complex specifications results in DPDA controllers, for which new aspects become relevant. Firstly, these DPDA controllers can become large (see section 7.5|p.157) due to our disambiguation procedure. To mitigate this problem, we argued that the size of DFA and DPDA controllers alike can be reduced by including steps that are not imitable by the plant (see section 8.3|p.209). Secondly, DPDA controllers exploit internal steps to operate on their stacks and may not exhibit a WCET. To obtain an equivalent controller *with* WCET, we proposed an annotation of the runtime environment to identify and shortcut the problematic unbounded sequences of stack manipulations (see subsection 8.2.3|p.207). Thirdly, DPDA controllers may require more space for their stack than available at runtime leading to stack overflows. To minimize the likelihood of stack overflows, we proposed usage patterns preventing overly deep stacks. Also, to estimate the likelihood of stack overflows, we proposed probabilistic extensions of the involved models.

These technical difficulties when applying our concrete controller synthesis algorithm are moderated by the possibility to even further extend the specifications' expressiveness in the future using temporal logics such as CaRet (see subsection 8.3.1|p.209). However, increasingly more expressive specifications create the risk of defective specifications rendering processes for their validation indispensable. We proposed use-cases and extended outputs to alleviate this problem by reasonably re-integrating human personnel into the synthesis workflow (see section 7.5|p.157).

—• *Extensions of our Concrete Controller Synthesis Algorithm*

Going forward, future extensions with practical importance of our synthesis algorithm should, following research directions from related work, *incorporate further systems' characteristics* by enforcing stronger semantical properties on more expressive formalisms (see subsection 8.3.2|p.214 for a complete discussion).

Four examples of such extensions are as follows. (a) We recommend to suitably integrate a notion of time in the plants, the specifications, and the controller synthesis algorithms to bridge the gap from logical time (where the models merely describe causality of events) to real time (where the models describe possible delays between events). (b) To obtain a controller that drives the plant towards a desired goal region by implementing a winning strategy, constructions that suitably restrict the synthesized least restrictive DPDA controller are called for. (c) We envision the usage of Visibly Pushdown Automata (VPA) to allow for the synchronous horizontal composition of multiple controlled plants, controllers, and closed loops to further extend applicability. (d) The analysis of probabilistic automata in the form of Markov Decision Processes (MDPs) as well as controllers

in this setting have been developed, but a combination of MDPs and DPDA for the fully automatic synthesis of controllers has not yet been considered.

Finally, as to whether the mentioned extensions permit combination thereof remains to be further investigated.

→ *Focus and Applications of our Isabelle Framework*

For the formal verification of our concrete controller synthesis algorithm as the first main contribution of this thesis, we provided an Isabelle framework, which consists of abstract theories for semantics, definitions, and verified theorems (see section 6.2|p.96). We applied this framework to prove the correctness of our abstract controller synthesis algorithm operating on discrete event systems, to obtain various semantics for the formalisms of EPDA, CFGs, and Parsers including concrete interpretations for several abstract notions such as nonblockingness, and to certify the correctness of our concrete controller synthesis algorithm operating on instances of these three formalisms. This Isabelle framework is a sensible enhancement to previous Isabelle frameworks (see subsection 8.1.8|p.186). It can be applied to define the semantics of a multitude of further standard formalisms and to verify similar constructions on them. Moreover, due to its modularity, it can be extended into various directions (see subsection 8.1.8|p.186) by including further characteristics of semantics or by including further abstract notions.

→ *Overall Assessment*

The problem of discrete event supervisory controller synthesis from control theory, which is similar to the software synthesis problem from computer science, has received great attention since its first publication [282] in 1984 and has resulted in a research field that has been active ever since.

The particular problem of determining a synthesis algorithm for a reasonable class of non-regular specification languages leading to infinite closed loop state spaces was stated in [281] and has thereby been open for more than two decades. It was shown to be unsolvable for Petri net plants and specifications and a previous solution for DPDA specifications turned out to be erroneous.

By adapting existing theory on parsing and controller synthesis as well as on existing technology for formal verification, we determined a mechanically verified and implemented fixed-point algorithm that solves this pivotal problem for the setting where DPDA specifications and DFA plant models are used. Our work is the starting point for a vast potential of future research to enhance effectiveness, efficiency, and applicability of the proposed synthesis algorithm. Most noticeably we recommend the formal verification of the proposed second controller synthesis algorithm that is additionally guaranteed to terminate on all inputs.

Due to the foundational nature of the solved problem, we expect further applications of our controller synthesis algorithm (or its building blocks) in computer science and control theory alike.

A

Disclaimer on Collaborations and Joint Work

—• *Reduction of Controllability to Nonblockingness for DPDA Controllers*

The building block of our concrete controller synthesis algorithm for the reduction of DPDA controllability to DPDA nonblockingness in section 5.3|p.82 is small in terms of the size of the definitions and proofs (see Figure 6.1|p.97) and is a reformulation of a construction that was mainly developed by Anne-Kathrin Schmuck. However, the formal Isabelle-based verification and adaptations to indispensable runtime optimizations of this building block are our contributions.

—• *The DES Fixed Point Iterator Enforce-Marked-Controllable-Subset*

The building block from section 5.3|p.82 implements the fixed point iterator *Enforce-Marked-Controllable-Subset* (see subsection 4.2.4|p.52) as discussed in [302, 301]. This fixed point iterator has been obtained by Anne-Kathrin Schmuck as a modification of a fixed point iterator from [310]. The formalization of this fixed point iterator, the formalization of the mentioned connection to the automata-based building block, the entire theory and framework of DES including the composition of fixed point iterators, and the other employed fixed point iterators are our contributions.

—• *The Automated Fabrication Scenario A*

In Example 7.2|p.147, we presented an application of our concrete controller synthesis algorithm to a manufacturing example presented earlier. The fundamental ideas of (a) two kinds of items that are to be produced, (b) the remembering of the difference/mismatch of the number of the two kinds of items using the stack, (c) the second mode in which the mismatch is reduced to zero by disabling the corresponding production units, and (d) the subsequent reset of the entire system are our contributions. The idea to use multiple machines producing two kinds of items was suggested by Thomas Moor. Finally, we attribute the compositional definition of the plant and the actual realization in terms of [302, Section 5.1, pp. 15–] to our coauthors Anne-Kathrin Schmuck and Jörg Raisch.

B

Isabelle-based Notation

We briefly cover the notation used for definitions and theorems, which is based on the syntax of the interactive theorem prover Isabelle used in this thesis for the formal verification of our results. The syntax and semantics of types and terms in Isabelle is derived from the functional programming language Meta Language (ML), which is based on the typed λ -calculus. Isabelle also enables the use of functions as formal variables in quantifications or as values in arguments to other functions. Moreover, the functional paradigm allows for the encoding of functional programs as mathematical functions in standard notation and Isabelle is therefore well suited for the verification of such functional programs. See [265] for more elaborate introductions for users of Isabelle.

CONTENTS OF THIS CHAPTER

B.1|p.228 Standard and Custom Types and Operations in Isabelle

We introduce basic types (such as Booleans, lists, sets, and functions) together with operations on them.

B.2|p.230 Defining Types, Terms, Functions, and Theorems

We introduce additional syntactical constructs that are used for defining types, terms, functions, and theorems.

B.3|p.231 Custom Notation for Derivations

We introduce our abbreviating notation for derivations along with some basic operations on them.

B.1. STANDARD AND CUSTOM TYPES AND OPERATIONS IN ISABELLE

We discuss various types used in this thesis, which are all equipped with an equality relation $=$.

→ The Type *bool*

The type of Boolean values *bool* is defined using the two constructors *True* and *False* (both without arguments). For formulas, we make use of the standard connectives of disjunction $p_1 \vee p_2$, conjunction $p_1 \wedge p_2$, negation $\neg p$, implication $p_1 \longrightarrow p_2$, equality $p_1 \longleftrightarrow p_2$, existential quantification $\exists x. p$, and universal quantification $\forall x. p$. Further abbreviations are available such as $\forall x \in S. p$ instead of $\forall x. x \in S \longrightarrow p$, $\exists x \in S. p$ instead of $\exists x. x \in S \wedge p$, $\exists x y. p$ instead of $\exists x. \exists y. p$ (also for universal quantification and longer lists of variables), and $\exists! x. p$ instead of $\exists x. p \wedge (\forall y. p \longrightarrow x = y)$. The two choice operators $\iota x. p$ and $\epsilon x. p$ are explained in the next paragraph.

→ The Type α *option*

The type of optional values α *option* is defined using the two constructors *None* and *Some e* where *e* is an element of type α . This type is used when an operation does not return a value (of type α) in every case. That is, the operation then returns no values using *None* or returns a value *e* using *Some e*.

The ϵ -operator selects some element satisfying a property *p* according to the axiom $\exists x. p x \implies p(\epsilon x. p x)$ and the ι -operator selects the unique element satisfying a property *p* according to the axiom $\exists! x. p x \implies p(\iota x. p x)$.

→ The Type *nat*

The type of natural numbers *nat* is defined using the two constructors 0 and *Suc e* where *e* is an element of type *nat*. Moreover, the value 1 abbreviates *Suc 0*.

→ The Type (α, β) *tuple2*

The type of pairs (α, β) *tuple2* is defined using the constructor *tuple2 e₁ e₂* where *e₁* is an element of type α and where *e₂* is an element of type β . We use the selectors *sel21* and *sel22* for the two fields. We avoid the use of the standard pair type $\alpha \times \beta$ because it is only defined for two values. We also used types for more values and provided a consistent naming scheme for the selectors.

→ The Type α *set*

Sets can be defined using set comprehension $\{x. p\}$ where *x* is a variable of type α , which possibly appears in the term *p*. Further abbreviations are available such as the universal set *UNIV* instead of $\{x. \text{True}\}$, the empty set $\{\}$ instead of $\{x. \text{False}\}$, and the set enumeration (e.g. $\{e_1, e_2, e_3\}$ instead of $\{x. x = e_1 \vee x = e_2 \vee x = e_3\}$). For sets, we make use of the standard operations of union $S_1 \cup S_2$, intersection $S_1 \cap S_2$, and inclusion $S_1 \subseteq S_2$. Also, we use the standard operations $\bigcup \mathcal{S} = \{x. \exists S \in \mathcal{S}. x \in S\}$ and $\bigcap \mathcal{S} = \{x. \forall S \in \mathcal{S}. x \in S\}$. The cardinality of a set *card S* is defined in Isabelle to be 0 for infinite sets and to be the number of contained elements for finite sets. That is, finite sets (but not infinite sets) can be readily compared regarding their size using this operation.

→ Functions of Type $\alpha \Rightarrow \beta$

Functions can be defined inside terms using anonymous functions (without a name) using the notation $\lambda x. t$ where x is the formal input variable of function, which possibly appears in the term t . Anonymous functions with multiple arguments can be defined as in $\lambda x y. t$.

The function composition operation \circ of two functions f and g of types $\alpha \Rightarrow \beta$ and $\beta \Rightarrow \gamma$, respectively, results in a function $g \circ f$ of type $\alpha \Rightarrow \gamma$.

The monotone predicate *mono* f on a function f of type $\alpha \Rightarrow \beta$ is defined as usual by $\forall x. \forall y. x \leq y \longrightarrow f\ x \leq f\ y$. Note that this definition makes use of operations \leq and \leq on the types α and β , respectively.

The function image $f ` S$ of a function f of type $\alpha \Rightarrow \beta$ on a set S of type α set returns a set $\{y. \exists x \in S. y = f\ x\}$.

→ The Type α list

The type of lists α list is defined using the two constructors $[]$ and $e \# l$ where e is an element of type α and where l is an element of type α list. Also, list enumeration such as $[e_1, e_2, e_3]$ abbreviates $e_1 \# (e_2 \# (e_3 \# []))$.

We use the operation *length* l to determine the length of the list, the operation $l_1 @ l_2$ for appending lists of common type, the operation *concat* L for appending all lists in the list L (i.e., L is here of type α list list and the returned list is of type α list), the operation *rev* l for reverting the order of the elements in a list, the operation *take* $n\ l$ for selecting the sublist of l consisting of the first n elements of l (or fewer if l contains less than n elements), the operation *drop* $n\ l$ for selecting the sublist of l consisting of the elements subsequent to the first n elements of l (or no elements if l contains less than n elements), and the operation $l ! n$ for selecting the n th element from l .

The prefix predicate $l_1 \sqsubseteq l_2$ is satisfied if there is some l such that $l_1 @ l = l_2$. The strict prefix predicate $l_1 \sqsubset l_2$ is satisfied if $l_1 \sqsubseteq l_2$ and also $l_1 \neq l_2$. The prefix closure operation collects all prefixes of elements of a set S , that is, *prefix-closure* $S = \{l_1. \exists l_2 \in S. l_1 \sqsubseteq l_2\}$.

Moreover, the operation *option-to-list* converts an element of type α option into an element of type α list by using the two equations *option-to-list* *None* = $[]$ and *option-to-list* (*Some* e) = $[e]$.

→ The Type (α, β) bi-elem

For CFGs we introduced the type (α, β) bi-elem in section 2.4|p.27 to be able to use lists of type (α, β) bi-elem list in which nonterminals of type α and events of type β are both contained. The type (α, β) bi-elem is constructed by the two constructors *beA* for nonterminals and *beB* for events. The operation *bi-elem-domain* $N\ \Sigma = \{ beA\ A \mid A . A \in N \} \cup \{ beB\ b \mid b . b \in \Sigma \}$ then determines all elements of this type for given sets N and Σ of nonterminals and events, respectively.

We provide various operations such as *filter-A* l to select the nonterminals contained in the list l and *lift-B* l to convert a list of events of type β list into a list of type (α, β) bi-elem list by applying *beB* to each element.

Note, we use this type also for other purposes in this thesis as in the *cfgPsplit* semantics in section 6.3|p.116 for representing lists of EPDA edges and CFG productions.

B.2. DEFINING TYPES, TERMS, FUNCTIONS, AND THEOREMS

→ Defining Types

Possibly recursive sum types are defined using the **datatype** keyword. See Definition 2.1|p.17 for the definition of the α *des* type (with one constructor) and the following definition of the α *option* type from the previous section (with two constructors and the selector *the*).

Example B.1: «Usage of datatype »

```
datatype  $\alpha$  option =
  None
  Some the: $\alpha$ 
```

Product types can be defined (besides using also **datatype** as for the definition of the α *des* type) using the keyword **record**. In the following example, we define the type (*nonterminal, event*) *cfg* to be a record with four fields.

Example B.2: «Usage of record »

```
record (nonterminal, event) cfg =
  cfg-nonterminals nonterminal set
  cfg-events event set
  cfg-initial nonterminal
  cfg-productions (nonterminal, event) cfg-step-label set
```

This definition also introduces selectors that are given by the names of the fields. Finally, a record *r* of type (*nonterminal, event*) *cfg* can be updated as in $r[\langle \text{cfg-nonterminals} := \{\text{cfg-initial } r\} \rangle]$ where *cfg-initial* has been used as a selector.

→ Defining Terms

The **let-in** construct from functional programming introduces inline abbreviations. For example, $(n + n) * (n + n)$ can be rewritten into **let** *m* = *n* + *n* **in** *m* * *m* or **let** *m* = *n* + *n*; *r* = *m* * *m* **in** *r*.

The **if-then-else** construct from functional programming is available as well such as in **if** *n* = 0 **then** *n* **else** *n* + *n*.

→ Defining Functions

We define nonrecursive functions using the keyword **definition** and recursive definitions using the keyword **function** (the additional keywords of **primrec** and **fun** are used in our formalization but are not used in this document). The keyword **function** permits the definition of functions where the termination of the recursive computation is verified only for a subset of the domain by using the

keyword **domintros**. This is particularly important in cases where the domain type contains more elements than desirable. For example, operations on CFGs can also be applied on CFGs with infinite set of nonterminals but we restrict our considerations to the subset of valid CFGs. Examples of usages of the keywords **definition** and **function** are given in Definition 2.2|p.17 and Definition 4.2|p.47, respectively.

—• *Defining Theorems*

Terms in the meta-logic are used for stating theorems, determine proof states, and are constructed as follows. Firstly, if t is term of type *bool*, then t is also a meta-logical term. Secondly, if P and Q are meta-logical terms, then $P \implies Q$ is meta-logical term, which states that Q holds whenever P holds. Thirdly, if x is a meta-logical variable and P is a meta-logical term, then $\bigwedge x. P$ is meta-logical term, which states that P holds for every x .

The following example is the standard induction theorem on natural numbers and features the universal meta logical quantification in the second assumption. The induction theorem is applicable to any given term P that contains a subterm n of type *nat*.

Theorem B.1: «nat-induct»

$$\begin{array}{l} P\ 0 \\ \implies (\bigwedge n. P\ n \implies P\ (Suc\ n)) \\ \implies P\ n \end{array}$$

B.3. CUSTOM NOTATION FOR DERIVATIONS

We use $c_1 \xrightarrow[\text{semantics,structure}]{\text{labels}} c_2$ to state the existence of a derivation starting in c_1 and ending in c_2 .

- The parameter *semantics* is the (name of the) semantics employed for the derivation.
- The parameter *structure* is the EPDA, CFG, or Parser to which the configurations and step-labels belong.
- The parameter *labels* is the list of edges of an EPDA, the list of productions of a CFG, or the list of rules of a Parser used for the steps of the derivation. That is, the length of the derivation is given by the length of the list of this parameter.

Thereby, we define a relation between two configurations that is additionally equipped with the three given parameters.

Also, we use $c_1 \xrightarrow[\text{semantics,structure}]{\text{labels1}} c_2 \xrightarrow[\text{semantics,structure}]{\text{labels2}} c_3$ when an intermediate configuration c_2 is important.

If the semantics has a unique initial configuration and the derivation starts with this configuration, we write $init \xrightarrow[\text{semantics,structure}]{\text{labels}} c_2$.

Operational Properties for DPDA Controllers

The operational properties formally defined here are used in section 3.2|p.36 to define satisfactory controllers in our concrete supervisory control problem. In our Isabelle-based formalization, we define these properties in our hierarchy of locales using abstract parameters such as marking conditions and then instantiate these locales for EPDA using interpretations for each parameter. For presentation purposes, we provide here equivalent definitions using the corresponding notions of EPDA directly.

Operational controllability means that the controller is able to execute an uncontrollable event u using a derivation in which also other silent steps can be performed, whenever the plant can execute the event u and both, plant and controller, executed the same word of events before.

Definition C.1: «*epda-operational-controllable*»

definition *epda-operational-controllable*

$:: (\text{controller-state}, \text{event}, \text{controller-stack}) \text{ epda}$
 $\Rightarrow (\text{plant-state}, \text{event}, \text{unused-stack}) \text{ epda}$
 $\Rightarrow \text{event set}$
 $\Rightarrow \text{bool}$

where *epda-operational-controllable* $C \ P \ \Sigma_{uc}$

$\equiv \forall \pi_1 \ c_1 \ \pi_2 \ c_2 \ p \ c'_2 \ u.$
 $\quad \text{init} \xrightarrow[\text{epdaH,C}]{\pi_1} c_1$
 $\quad \longrightarrow \text{init} \xrightarrow[\text{epdaH,P}]{\pi_2} c_2 \xrightarrow[\text{epdaH,P}]{[p]} c'_2$
 $\quad \longrightarrow \text{epdaH-conf-history } c_1 = \text{epdaH-conf-history } c_2$
 $\quad \longrightarrow \text{epdaH-conf-history } c'_2 = \text{epdaH-conf-history } c_2 @ [u]$
 $\quad \longrightarrow u \in \Sigma_{uc}$
 $\quad \longrightarrow (\exists \pi_3 \ c'_1.$
 $\quad \quad c_1 \xrightarrow[\text{epdaH,C}]{\pi_3} c'_1$
 $\quad \quad \wedge \text{epdaH-conf-history } c'_1 = \text{epdaH-conf-history } c'_2)$

Operational nonblockingness is satisfied if every initial derivation of the closed loop to a configuration c_1 can be continued to a configuration c_2 in which the state is a marking state of the EPDA.

Definition C.2: «epda-operational-nonblockingness»

definition *epda-operational-nonblockingness*

$:: (\text{state}, \text{event}, \text{stack}) \text{ epda}$
 $\Rightarrow \text{bool}$

where *epda-operational-nonblockingness* CL

$\equiv \forall \pi_1 c_1. \text{init} \xrightarrow{\pi_1}_{\text{epdaH}, CL} c_1$
 $\longrightarrow (\exists \pi_2 c_2. c_1 \xrightarrow{\pi_2}_{\text{epdaH}, CL} c_2 \wedge \text{epdaH-conf-state } c_2 \in \text{epda-marking } CL)$

The closed loop CL satisfies the specification S if every initial derivation of the closed loop can be mimicked by the specification regarding the execution of events and when also the closed loop only marks an executed word if the specification also marks the word.

Definition C.3: «epda-operational-specification-satisfaction»

definition *epda-operational-specification-satisfaction*

$:: (\text{closed-loop-state}, \text{event}, \text{closed-loop-stack}) \text{ epda}$
 $\Rightarrow (\text{specification-state}, \text{event}, \text{specification-stack}) \text{ epda}$
 $\Rightarrow \text{bool}$

where *epda-operational-specification-satisfaction* $CL S$

$\equiv \forall \pi_1 c_1.$
 $\quad \text{init} \xrightarrow{\pi_1}_{\text{epdaH}, CL} c_1$
 $\longrightarrow (\exists \pi_2 c_2.$
 $\quad \text{init} \xrightarrow{\pi_2}_{\text{epdaH}, S} c_2$
 $\quad \wedge \text{epdaH-conf-history } c_1 = \text{epdaH-conf-history } c_2$
 $\quad \wedge (\text{epdaH-conf-state } c_1 \in \text{epda-marking } CL$
 $\quad \longrightarrow \text{epdaH-conf-state } c_2 \in \text{epda-marking } S))$

The absence of deadlocks in the closed loop CL is stated by requiring that at least one edge e' is applicable to every accessible configuration where the current state is not a marking state of the closed loop.

Definition C.4: «epda-deadlock-freedom»

definition *epda-deadlock-freedom*

$:: (\text{state}, \text{event}, \text{stack}) \text{ epda}$
 $\Rightarrow \text{bool}$

where *epda-deadlock-freedom* CL

$\equiv \forall \pi c.$
 $\quad \text{init} \xrightarrow{\pi}_{\text{epdaH}, CL} c$
 $\longrightarrow \text{epdaH-conf-state } c \notin \text{epda-marking } CL$
 $\longrightarrow (\exists e' c'. c \xrightarrow{[e']}_{\text{epdaH}, CL} c')$

For the absence of livelocks, we cannot use our notation for finite (initial) derivations. In the following definition, we state that d is an initial derivation of the closed loop CL in the $epdaH$ semantics using $epdaH.derivation-initial\ CL\ d$. Moreover, we state that d is infinite by stating that it has a configuration for every natural number n using $\forall n. d\ n \neq None$. Furthermore, we collect all unmarked words (that is, the unmarked effects in the terminology of our formalization) executed in d up to index $i \in \{n, N\}$ using $epdaH-unmarked-effect\ CL\ (derivation-take\ d\ i)$ where $derivation-take$ is used to crop the infinite derivation d to its finite prefix of length i . That is, we state the absence of an infinite derivation that does not change its $epdaH-conf-history$ variable after some index N .

Definition C.5: «epda-livelock-freedom»

definition $epda-livelock-freedom :: (state, event, stack) epda \Rightarrow bool$
where $epda-livelock-freedom\ CL$
 $\equiv \neg(\exists d. epdaH.derivation-initial\ CL\ d$
 $\quad \wedge (\forall n. d\ n \neq None)$
 $\quad \wedge (\exists N. \forall n \geq N.$
 $\quad \quad epdaH-unmarked-effect\ CL\ (derivation-take\ d\ N)$
 $\quad \quad = epdaH-unmarked-effect\ CL\ (derivation-take\ d\ n)))$

For accessibility of an EPDA C , we require that every edge e of the EPDA C can be used in some initial derivation and that every state q of the EPDA C is used in some configuration that occurs in some initial derivation.

Definition C.6: «epda-accessible»

definition $epda-accessible :: (state, event, stack) epda \Rightarrow bool$
where $epda-accessible\ C$
 $\equiv (\forall e \in epda-delta\ C. \exists \pi c. init \xrightarrow{\pi @ [e]}_{epdaH,C} c)$
 $\quad \wedge (\forall q \in epda-states\ C. \exists \pi c. init \xrightarrow{\pi}_{epdaH,C} c \wedge q = epdaH-conf-state\ c)$

For accessibility in the closed loop, we obtain an edge of the controller C from an edge of the closed loop CL by projecting on the first state of the pair of states.

Definition C.7: «epda-accessible-in-closed-loop»

definition $epda-accessible-in-closed-loop$
 $:: (controller-state, event, controller-stack) epda$
 $\Rightarrow ((controller-state, plant-state) tuple2, event, controller-stack) epda\ set$
 $\Rightarrow bool$
where $epda-accessible-in-closed-loop\ C\ CL$
 $\equiv (\forall e \in epda-delta\ C. \exists \pi c\ e'. init \xrightarrow{\pi @ [e']}_{epdaH,C} c$
 $\quad \wedge edge-src\ e = sel21\ (edge-src\ e')$
 $\quad \wedge edge-event\ e = edge-event\ e' \wedge edge-pop\ e = edge-pop\ e'$
 $\quad \wedge edge-push\ e = edge-push\ e' \wedge edge-trg\ e = sel21\ (edge-trg\ e'))$
 $\quad \wedge (\forall q \in epda-states\ C. \exists \pi c. init \xrightarrow{\pi}_{epdaH,C} c$
 $\quad \quad \wedge q = sel21\ (epdaH-conf-state\ c'))$

D

The *cfgEsplit* Semantic for LR(1)-CFG

The material of this chapter belongs to section 6.3|p.116 where we presented a proof idea in more detail. We list the well-formedness conditions that are satisfied by the configurations of the *cfgEsplit* semantics, describe the *cfgEsplit* step relation, and provide an example derivation using *cfgEsplit*.

Definition D.1: «Well-formed Configurations of *cfgEsplit*»

The configurations of the semantics *cfgEsplit* for a given CFG G' are given by a list \mathcal{I} of elements I of type *esplit-item* as explained in Definition 6.13|p.127.

*Basic Intuition for the Described *cfgLM*-Derivations:*

Each item I contained in such a list \mathcal{I} satisfies two basic properties. These two properties determine our basic understanding of the semantics and its purpose and are satisfied by the initial configuration of *cfgEsplit* and are preserved by the step relation of *cfgEsplit*.

Firstly, each element of *esplit-item-elim* I (that is, the element at each index i satisfying $0 \leq i < \text{length}(\text{esplit-item-elim } I)$) is eliminated by the left-most derivation given by the corresponding sequence in *esplit-item-elim-prods* I .

$$(\text{esplit-item-elim } I) ! i \xrightarrow[\text{cfgLM}, G']{(\text{esplit-item-elim-prods } I) ! i} [] \quad (\text{D.1})$$

Secondly, if *esplit-item-from* $I = \text{Some } A$, then from A we derive by the left-most derivation given by *esplit-item-prods* I an element χ stored as *Some* χ in *esplit-item-elem* I with post context *esplit-item-to* I .

$$A \xrightarrow[\text{cfgLM}, G']{\text{esplit-item-prods } I} [\chi] (\text{esplit-item-to } I) \quad (\text{D.2})$$

This basic intuition, which explains the fundamental idea of the decomposition captured in the configurations of the *cfgEsplit* semantics, is subsequently extended to the list of actual technical constraints.

Constraints for Well-formed Configurations:

Besides that all productions, nonterminals, and events occurring in such a configuration must belong to the given CFG G' , each well-formed configuration containing a list \mathcal{I} of elements of type *esplit-item* also satisfies the following constraints.

1. We consider two sequences of nonterminals that are given by concatenating three lists each: the first sequence is obtained from *esplit-item-elim* I , *option-to-list* (*esplit-item-from* I), and *esplit-item-ignore* I and the second sequence is obtained from *filter-A* (*option-to-list* (*esplit-item-elem* I)), *esplit-item-to* I , and *esplit-item-ignore* I .

These two sequences are proper l_3 - l_2 sequences of nonterminals of the CFG G' in the sense that (a) all but the last nonterminal are of the shape L_{q_1, σ, q_2} , (b) the last nonterminal is of the shape $L_{q, \sigma}$, and (c) two adjacent nonterminals $L_{q_1, \sigma_1, q_2} L_{q_3, \sigma_2, q_4}$ and $L_{q_1, \sigma_1, q_2} L_{q_3, \sigma_2}$ agree on their interface (that is, $q_2 = q_3$ in both cases). Note that also only such proper l_3 - l_2 sequences occur in left-most derivations of the CFG G' .

2. The list \mathcal{I} of elements of type *esplit-item* is not empty.
3. The first item I of the list \mathcal{I} satisfies (a) and either (b) or (c) where (a) states that *esplit-item-ignore* I is empty, where (b) states that a word with at least one event is derived by stating that *esplit-item-from* $I = \text{Some}(\text{cfg-initial } G')$ and *esplit-item-elim* $I = []$, and where (c) states that the initial nonterminal has already been eliminated to the empty list by stating that *esplit-item-from* $I = \text{None}$ and *esplit-item-elim* $I = [(\text{cfg-initial } G')]$. Note, the case (b) holds throughout the initial derivations of *cfgEsplit* except for the case when the corresponding configuration in *cfgRM* reaches an empty configuration resulting in the satisfaction of case (c).
4. The last item I of \mathcal{I} satisfies that *esplit-item-to* I and *esplit-item-ignore* I are empty. Note, this means in our case (also see Example 6.5[p.126]) that the last left-most derivation (which is represented by I) replaces all remaining nonterminals by possibly empty words of events.
5. All items I contained in the list \mathcal{I} (except for the last item) satisfy that *esplit-item-from* I and *esplit-item-elem* I are not *None*. Note, this means that all these items correspond already to some left-most derivation that leads to an element *esplit-item-elem* I present in the *cfgRM* configuration to be represented.
6. The last item I from the list \mathcal{I} satisfies that if *esplit-item-from* I is *None*, then *esplit-item-elim* I is not empty. Note, this means that the last item also corresponds to some left-most derivation. In fact, throughout initial derivations of *cfgEsplit* we have a last item that is either representing

an empty left-most derivation (see next item) or a left-most derivation eliminating nonterminals to the empty string.

7. All items I from \mathcal{I} (except for the last item) satisfy that an empty post-context *esplit-item-to* I implies that *esplit-item-prods* I is empty and that *esplit-item-from* I equals *esplit-item-to* I . Note, this case represents an empty left-most derivation that appears (besides in the initial configuration) whenever using productions where two nonterminals A and B are on the right-hand-side and where *esplit-item-from* $I = A$ is then not yet modified by some sequence of productions.
8. Two adjacent items $I_1 I_2$ occurring in \mathcal{I} must agree on their interface in the sense that the remainder of I_1 is picked up by the item I_2 . That is, the remainder given by the concatenation of *esplit-item-to* I_1 and *esplit-item-ignore* I_1 of the first item I_1 must equal the starting part given by the concatenation of *esplit-item-elim* I_2 , *option-to-list* (*esplit-item-from* I_2), and *esplit-item-ignore* I_2 of the second item I_2 . Note, this constraint ensures that the left-most derivations given by the items can be merged suitably to a global left-most derivation once all nonterminals occurring in *esplit-item-elem* components have been replaced when the *cfgRM* derivation to be represented reaches a nonterminal-free configuration.
9. All items I contained in the list \mathcal{I} also satisfy for the eliminating left-most derivations (see (Eq. D.1)|p.237 above) that the two lists *esplit-item-elim* I and *esplit-item-elim-prods* I agree on their length. This is required because each list of productions contained at some index i in *esplit-item-elim-prods* I belongs to the corresponding nonterminal at index i in *esplit-item-elim* I .
10. All items I from \mathcal{I} also satisfy (a) and (b) for the generating left-most derivations (see (Eq. D.2)|p.237 above) where (a) states that if *esplit-item-elem* I equals *Some* χ where χ is an event, then the list of employed productions *esplit-item-prods* I has no strict prefix of productions generating this event, that is, the event χ must be generated in the last step of the represented left-most derivation and where (b) states that if *esplit-item-elem* I equals *Some* χ where χ is a nonterminal, then no nonterminals are eliminated simultaneously (that is, *esplit-item-elim-prods* I is empty) and the represented left-most derivation is of left-most-degenerate form (that is, the left-most derivation replaces always the first element of the configuration, which is throughout the derivation a nonterminal).

The initial derivations of the *cfgEsplit* semantics satisfy further invariants when using LR(1)-CFG as constructed in our concrete controller synthesis algorithm. These invariants could have been included here also as well-formedness constraints but they already focus on the actual proof strategy employed and are therefore omitted here.

We now informally introduce the step relation of *cfgEsplit* that is then also subsequently used in Example D.1|p.243. Moreover, we believe that the correspondence between *cfgRM* derivations and *cfgEsplit* derivations is instructive enough for a basic understanding of the step relation of *cfgEsplit*.

Definition D.2: «Step Relation *cfgEsplit*-step-relation»

We define the step relation *cfgEsplit*-step-relation $G' \mathcal{I} \rho \mathcal{I}'$ by describing the basic idea first and by then describing the steps for the different kinds of productions occurring in G' according to the construction given in Definition 6.8|p.117.

General Construction: To achieve a behavior that is compatible with *cfgRM* we apply the production $\rho = \langle A, \omega \rangle$ to the last nonterminal of the configuration given by \mathcal{I} . Firstly, we determine the last item I in \mathcal{I} where *esplit-item-elem* $I = \text{Some } A'$ for some nonterminal A' . Secondly, the production ρ is then applicable if $A = A'$. Thereby we decompose \mathcal{I} into $\mathcal{I}_1 I \mathcal{I}_2$ and distinguish below between empty and nonempty sequences \mathcal{I}_2 . In general, we construct from I a replacement list \mathcal{I}_R (see next block) and then merge \mathcal{I}_R with the first item I' of \mathcal{I}_2 (if existent) into the merged replacement list \mathcal{I}_{MR} (see the block after next) to be used in the resulting configuration \mathcal{I}' , which is equal to $\mathcal{I}_1 \mathcal{I}_{MR}(\text{drop } 1 \mathcal{I}_2)$.

Construction of Replacement List \mathcal{I}_R :

If ω is empty the replacement list \mathcal{I}_R contains only the following item (where I has been modified in the *esplit-item-prods* and *esplit-item-elem* components).

$$\begin{aligned} \langle \text{esplit-item-elim} &= \text{esplit-item-elim } I, \\ \text{esplit-item-from} &= \text{esplit-item-from } I, \\ \text{esplit-item-ignore} &= \text{esplit-item-ignore } I, \\ \text{esplit-item-elim-prods} &= \text{esplit-item-elim-prods } I, \\ \text{esplit-item-prods} &= (\text{esplit-item-prods } I)\rho, \\ \text{esplit-item-elem} &= \text{None}, \\ \text{esplit-item-to} &= \text{esplit-item-to } I \end{aligned} \quad \rangle$$

The list \mathcal{I}_R then only contains this single item, which is then to be merged with the first element of \mathcal{I}_2 (if existent) in the next block.

If ω is not empty it is of the form $[\chi, A]$, $[A]$, or $[A, B]$ where χ is an event and where A and B are nonterminals (see Definition 6.8|p.117). We construct one item for each element of ω . Firstly, if ω contains an event χ , we construct the following item (where I has been modified in the *esplit-item-prods*, *esplit-item-elem*, and *esplit-item-to* components). Also, Y denotes the list of nonterminals in ω .

$$\begin{aligned} \langle \text{esplit-item-elim} &= \text{esplit-item-elim } I, \\ \text{esplit-item-from} &= \text{esplit-item-from } I, \\ \text{esplit-item-ignore} &= \text{esplit-item-ignore } I, \\ \text{esplit-item-elim-prods} &= \text{esplit-item-elim-prods } I, \\ \text{esplit-item-prods} &= (\text{esplit-item-prods } I)\rho, \\ \text{esplit-item-elem} &= \text{Some } \chi, \\ \text{esplit-item-to} &= Y(\text{esplit-item-to } I) \end{aligned} \quad \rangle$$

Secondly, we construct for the i th element of Y the following item.

$$\begin{aligned}
\langle \text{esplit-item-elim} &= [], \\
\text{esplit-item-from} &= Y ! i, \\
\text{esplit-item-ignore} &= (\text{drop } (i + 1) Y) \\
&\quad (\text{esplit-item-to } I) \\
&\quad (\text{esplit-item-ignore } I), \\
\text{esplit-item-elim-prods} &= [], \\
\text{esplit-item-prods} &= [], \\
\text{esplit-item-elem} &= \text{Some } (Y ! i), \\
\text{esplit-item-to} &= [] \quad \rangle
\end{aligned}$$

The list of these (at most two) items then determines the replacement list \mathcal{I}_R .

Merging of Replacement List \mathcal{I}_R into \mathcal{I}_{MR} :

For merging, we use the first item I' of \mathcal{I}_2 or (if \mathcal{I}_2 is empty) the following empty item I' to come up with a canonical definition for merging.

$$\begin{aligned}
\langle \text{esplit-item-elim} &= [], \\
\text{esplit-item-from} &= \text{None}, \\
\text{esplit-item-ignore} &= [], \\
\text{esplit-item-elim-prods} &= [], \\
\text{esplit-item-prods} &= [], \\
\text{esplit-item-elem} &= \text{None}, \\
\text{esplit-item-to} &= [] \quad \rangle
\end{aligned}$$

Merging is necessary if ω is empty because in this case we constructed a single item I_s in the previous step with $\text{esplit-item-elem } I_s = \text{None}$, which essentially describes how a certain nonterminal can be eliminated using $\text{esplit-item-prods } I_s$. The merging defined in this block then moves these productions into an $\text{esplit-item-elim-prods}$ component of the resulting item and also constructs other components as necessary.

In the simplest case, the eliminated nonterminal is $\text{esplit-item-from } I_s$ when $\text{esplit-item-prods } I_s$ is of length 1 and $\text{esplit-item-to } I_s$ is empty. However, if $\text{esplit-item-to } I_s$ is not empty, we eliminated some other nonterminal occurring in left-most position during the left-most derivation that is described by $\text{esplit-item-prods } I_s$. Actually, we must consider the two cases (a) and (b) where (a) means that the remainder $\text{esplit-item-to } I_s$ is entirely eliminated by I' ($\text{esplit-item-to } I_s$ is a prefix of $\text{esplit-item-elim } I'$) and where (b) means that at least one trailing nonterminal from $\text{esplit-item-to } I_s$ has not been eliminated in I' .

In case of (a) there is a decomposition $(\text{esplit-item-to } I_s)Y' = \text{esplit-item-elim } I'$ and by the well-definedness of the source configuration there is also a decomposition $\Pi_1\Pi_2 = \text{esplit-item-elim-prods } I'$ where Π_1 and Π_2 are of length $\text{esplit-item-to } I_s$ and Y' , respectively. In this case, we merge the item I_s into the item I' by integrating the eliminating derivation properly in the components esplit-item-elim

and *esplit-item-elim-prods*. Here, *option-to-list* (*esplit-item-from* I_s) is a list containing one nonterminal that is eliminated by $[(\text{esplit-item-prods } I_s)(\text{concat } \Pi_1)]$ according to the productions of I_s and the sequences of eliminating productions existing in I' for the remaining nonterminals given in *esplit-item-to* I_s .

$$\begin{aligned}
 \langle \text{esplit-item-elim} &= (\text{esplit-item-elim } I_s) \\
 &\quad (\text{option-to-list } (\text{esplit-item-from } I_s)) \\
 &\quad Y', \\
 \text{esplit-item-from} &= \text{esplit-item-from } I', \\
 \text{esplit-item-ignore} &= \text{esplit-item-ignore } I', \\
 \text{esplit-item-elim-prods} &= (\text{esplit-item-elim-prods } I_s) \\
 &\quad [(\text{esplit-item-prods } I_s)(\text{concat } \Pi_1)] \\
 &\quad \Pi_2, \\
 \text{esplit-item-prods} &= \text{esplit-item-prods } I', \\
 \text{esplit-item-elem} &= \text{esplit-item-elem } I', \\
 \text{esplit-item-to} &= \text{esplit-item-to } I' \rangle
 \end{aligned}$$

In case of (b) there is a decomposition $\text{esplit-item-to } I_s = (\text{esplit-item-elim } I')AY'$, which shows that there is insufficient information in I' to allow for the entire elimination of *esplit-item-to* I_s (hence, \mathcal{I}_2 is not empty). Since I' continues with the remainder of I_s , we derive from the well-definedness of configurations that *esplit-item-from* I' equals A and *esplit-item-ignore* I' equals $Y'(\text{esplit-item-ignore } I_s)$. Again, from the well-definedness of the configurations, we derive that the item I' handles the three parts of the decomposition above, which are all contained in I' , by using the eliminating derivations in *esplit-item-elim-prods* I' , by using the derivation on A from *esplit-item-prods* I' , and by ignoring the elements in the *esplit-item-ignore* I' component. We include all productions from I' into I_s in the way that they further derive the remainder $(\text{esplit-item-elim } I')AY'(\text{esplit-item-ignore } I_s)$ of I_s .

$$\begin{aligned}
 \langle \text{esplit-item-elim} &= \text{esplit-item-elim } I_s, \\
 \text{esplit-item-from} &= \text{esplit-item-from } I_s, \\
 \text{esplit-item-ignore} &= \text{esplit-item-ignore } I_s, \\
 \text{esplit-item-elim-prods} &= \text{esplit-item-elim-prods } I_s, \\
 \text{esplit-item-prods} &= (\text{esplit-item-prods } I_s) \\
 &\quad (\text{concat } (\text{esplit-item-elim-prods } I')) \\
 &\quad (\text{esplit-item-prods } I'), \\
 \text{esplit-item-elem} &= \text{esplit-item-elem } I', \\
 \text{esplit-item-to} &= (\text{esplit-item-to } I')Y' \rangle
 \end{aligned}$$

In the following example, we provide the two full initial derivations of *cfgEsplit* (corresponding to $d_{1,rm}$ and $d_{2,rm}$ from block 5|p.120) that were abbreviated in Example 6.6|p.128.

Example D.1: «Running Example for section 6.3|p.116 (Additional Part)»

(1) Derivation $d_{1,es}$ Corresponding to $d_{1,rm}$:

We provide the initial *cfgEsplit* derivation $d_{1,es}$, which is constructed using the same productions as used for the initial *cfgRM* derivation $d_{1,rm}$.

	<i>esplit-item-elim</i>	<i>esplit-item-from</i>	<i>esplit-item-ignore</i>	<i>esplit-item-elim-prods</i>	<i>esplit-item-prods</i>	<i>esplit-item-elim</i>	<i>esplit-item-to</i>
		$L_{1,\square}$				$L_{1,\square}$	
$\xrightarrow{\rho_1}_{cfgEsplit,G'}$		$L_{1,\square}$ $L_{15,\square}$			ρ_1	$L_{2,\bullet,15}$ $L_{15,\square}$	$L_{15,\square}$
$\xrightarrow{\rho_2}_{cfgEsplit,G'}$	$L_{15,\square}$	$L_{1,\square}$		ρ_2	ρ_1	$L_{2,\bullet,15}$	$L_{15,\square}$
$\xrightarrow{\rho_3}_{cfgEsplit,G'}$	$L_{15,\square}$	$L_{1,\square}$ $L_{5,\bullet,15}$	$L_{15,\square}$	ρ_2	$\rho_1\rho_3$	$L_{3,\bullet,5}$ $L_{5,\bullet,15}$	$L_{5,\bullet,15}L_{15,\square}$
$\xrightarrow{\rho_4}_{cfgEsplit,G'}$	$L_{15,\square}$	$L_{1,\square}$ $L_{5,\bullet,15}$ $L_{14,\bullet,15}$	$L_{15,\square}$ $L_{15,\square}$	ρ_2	$\rho_1\rho_3$ ρ_4	$L_{3,\bullet,5}$ $L_{6,\bullet,14}$ $L_{14,\bullet,15}$	$L_{5,\bullet,15}L_{15,\square}$ $L_{14,\bullet,15}$

continued

<i>continued</i>							
$\rightarrow_{\text{cfgEsplit}, G'}^{\rho_5}$	$L_{14, \bullet, 15} L_{15, \square}$	$L_{1, \square}$ $L_{5, \bullet, 15}$	$L_{15, \square}$	ρ_5, ρ_2	$\rho_1 \rho_3$ ρ_4	$L_{3, \bullet, 5}$ $L_{6, \bullet, 14}$	$L_{5, \bullet, 15} L_{15, \square}$ $L_{14, \bullet, 15}$
$\rightarrow_{\text{cfgEsplit}, G'}^{\rho_6}$	$L_{14, \bullet, 15} L_{15, \square}$	$L_{1, \square}$ $L_{5, \bullet, 15}$ $L_{9, \bullet, 14}$	$L_{15, \square}$ $L_{14, \bullet, 15} L_{15, \square}$	ρ_5, ρ_2	$\rho_1 \rho_3$ $\rho_4 \rho_6$	$L_{3, \bullet, 5}$ $L_{7, \bullet, 9}$ $L_{9, \bullet, 14}$	$L_{5, \bullet, 15} L_{15, \square}$ $L_{9, \bullet, 14} L_{14, \bullet, 15}$
$\rightarrow_{\text{cfgEsplit}, G'}^{\rho_7}$	$L_{14, \bullet, 15} L_{15, \square}$	$L_{1, \square}$ $L_{5, \bullet, 15}$ $L_{9, \bullet, 14}$ $L_{10, \bullet, 14}$	$L_{15, \square}$ $L_{14, \bullet, 15} L_{15, \square}$ $L_{14, \bullet, 15} L_{15, \square}$	ρ_5, ρ_2	$\rho_1 \rho_3$ $\rho_4 \rho_6$ ρ_7	$L_{3, \bullet, 5}$ $L_{7, \bullet, 9}$ b $L_{10, \bullet, 14}$	$L_{5, \bullet, 15} L_{15, \square}$ $L_{9, \bullet, 14} L_{14, \bullet, 15}$ $L_{10, \bullet, 14}$
$\rightarrow_{\text{cfgEsplit}, G'}^{\rho_8}$	$L_{14, \bullet, 15} L_{15, \square}$	$L_{1, \square}$ $L_{5, \bullet, 15}$ $L_{9, \bullet, 14}$ $L_{10, \bullet, 14}$ $L_{12, \bullet, 14}$	$L_{15, \square}$ $L_{14, \bullet, 15} L_{15, \square}$ $L_{14, \bullet, 15} L_{15, \square}$ $L_{14, \bullet, 15} L_{15, \square}$	ρ_5, ρ_2	$\rho_1 \rho_3$ $\rho_4 \rho_6$ ρ_7 ρ_8	$L_{3, \bullet, 5}$ $L_{7, \bullet, 9}$ b d $L_{12, \bullet, 14}$	$L_{5, \bullet, 15} L_{15, \square}$ $L_{9, \bullet, 14} L_{14, \bullet, 15}$ $L_{10, \bullet, 14}$ $L_{12, \bullet, 14}$
$\rightarrow_{\text{cfgEsplit}, G'}^{\rho_9}$	$L_{12, \bullet, 14} L_{14, \bullet, 15} L_{15, \square}$	$L_{1, \square}$ $L_{5, \bullet, 15}$ $L_{9, \bullet, 14}$ $L_{10, \bullet, 14}$	$L_{15, \square}$ $L_{14, \bullet, 15} L_{15, \square}$ $L_{14, \bullet, 15} L_{15, \square}$	ρ_9, ρ_5, ρ_2	$\rho_1 \rho_3$ $\rho_4 \rho_6$ ρ_7 ρ_8	$L_{3, \bullet, 5}$ $L_{7, \bullet, 9}$ b d	$L_{5, \bullet, 15} L_{15, \square}$ $L_{9, \bullet, 14} L_{14, \bullet, 15}$ $L_{10, \bullet, 14}$ $L_{12, \bullet, 14}$
<i>continued</i>							

continued

$\rightarrow_{\text{cfgEsplit}, G'}^{\rho_{10}}$	$L_{12, \bullet, 14} L_{14, \bullet, 15} L_{15, \square}$	$L_{1, \square}$ $L_{5, \bullet, 15}$ $L_{8, \bullet, 9}$ $L_{9, \bullet, 14}$ $L_{10, \bullet, 14}$	$L_{15, \square}$ $L_{9, \bullet, 14} L_{14, \bullet, 15} L_{15, \square}$ $L_{14, \bullet, 15} L_{15, \square}$ $L_{14, \bullet, 15} L_{15, \square}$	ρ_9, ρ_5, ρ_2	$\rho_1 \rho_3$ $\rho_4 \rho_6 \rho_{10}$ ρ_7 ρ_8	$L_{3, \bullet, 5}$ a $L_{8, \bullet, 9}$ b d	$L_{5, \bullet, 15} L_{15, \square}$ $L_{8, \bullet, 9} L_{9, \bullet, 14} L_{14, \bullet, 15}$ $L_{10, \bullet, 14}$ $L_{12, \bullet, 14}$
$\rightarrow_{\text{cfgEsplit}, G'}^{\rho_{11}}$	$L_{8, \bullet, 9}$ $L_{12, \bullet, 14} L_{14, \bullet, 15} L_{15, \square}$	$L_{1, \square}$ $L_{5, \bullet, 15}$ $L_{9, \bullet, 14}$ $L_{10, \bullet, 14}$	$L_{15, \square}$ $L_{14, \bullet, 15} L_{15, \square}$ $L_{14, \bullet, 15} L_{15, \square}$	ρ_{11} ρ_9, ρ_5, ρ_2	$\rho_1 \rho_3$ $\rho_4 \rho_6 \rho_{10}$ ρ_7 ρ_8	$L_{3, \bullet, 5}$ a b d	$L_{5, \bullet, 15} L_{15, \square}$ $L_{8, \bullet, 9} L_{9, \bullet, 14} L_{14, \bullet, 15}$ $L_{10, \bullet, 14}$ $L_{12, \bullet, 14}$
$\rightarrow_{\text{cfgEsplit}, G'}^{\rho_{12}}$	$L_{8, \bullet, 9}$ $L_{12, \bullet, 14} L_{14, \bullet, 15} L_{15, \square}$	$L_{1, \square}$ $L_{4, \bullet, 5}$ $L_{5, \bullet, 15}$ $L_{9, \bullet, 14}$ $L_{10, \bullet, 14}$	$L_{5, \bullet, 15} L_{15, \square}$ $L_{15, \square}$ $L_{14, \bullet, 15} L_{15, \square}$ $L_{14, \bullet, 15} L_{15, \square}$	ρ_{11} ρ_9, ρ_5, ρ_2	$\rho_1 \rho_3 \rho_{12}$ $\rho_4 \rho_6 \rho_{10}$ ρ_7 ρ_8	e $L_{4, \bullet, 5}$ a b d	$L_{4, \bullet, 5} L_{5, \bullet, 15} L_{15, \square}$ $L_{8, \bullet, 9} L_{9, \bullet, 14} L_{14, \bullet, 15}$ $L_{10, \bullet, 14}$ $L_{12, \bullet, 14}$
$\rightarrow_{\text{cfgEsplit}, G'}^{\rho_{13}}$	$L_{4, \bullet, 5}$ $L_{8, \bullet, 9}$ $L_{12, \bullet, 14} L_{14, \bullet, 15} L_{15, \square}$	$L_{1, \square}$ $L_{5, \bullet, 15}$ $L_{9, \bullet, 14}$ $L_{10, \bullet, 14}$	$L_{15, \square}$ $L_{14, \bullet, 15} L_{15, \square}$ $L_{14, \bullet, 15} L_{15, \square}$	ρ_{13} ρ_{11} ρ_9, ρ_5, ρ_2	$\rho_1 \rho_3 \rho_{12}$ $\rho_4 \rho_6 \rho_{10}$ ρ_7 ρ_8	e a b d	$L_{4, \bullet, 5} L_{5, \bullet, 15} L_{15, \square}$ $L_{8, \bullet, 9} L_{9, \bullet, 14} L_{14, \bullet, 15}$ $L_{10, \bullet, 14}$ $L_{12, \bullet, 14}$

<i>continued</i>						
$\rightarrow^{\rho_{18}}_{\text{cfgEsplit}, G'}$	$L_{13, \bullet}$	$L_{1, \square}$ $L_{5, \bullet}$ $L_{9, \bullet, 13} L_{13, \bullet}$		ρ_{17}	$\rho_{14} \rho_{15}$ $\rho_{16} \rho_{18}$	$L_{3, \bullet, 5}$ $L_{7, \bullet, 9}$ $L_{9, \bullet, 13}$ $L_{5, \bullet}$ $L_{9, \bullet, 13} L_{13, \bullet}$ $L_{13, \bullet}$
$\rightarrow^{\rho_{19}}_{\text{cfgEsplit}, G'}$	$L_{13, \bullet}$	$L_{1, \square}$ $L_{5, \bullet}$ $L_{9, \bullet, 13}$ $L_{10, \bullet, 13}$	$L_{13, \bullet}$ $L_{13, \bullet}$	ρ_{17}	$\rho_{14} \rho_{15}$ $\rho_{16} \rho_{18}$ ρ_{19}	$L_{3, \bullet, 5}$ $L_{7, \bullet, 9}$ b $L_{10, \bullet, 13}$ $L_{5, \bullet}$ $L_{9, \bullet, 13} L_{13, \bullet}$ $L_{10, \bullet, 13}$
$\rightarrow^{\rho_{20}}_{\text{cfgEsplit}, G'}$	$L_{13, \bullet}$	$L_{1, \square}$ $L_{5, \bullet}$ $L_{9, \bullet, 13}$ $L_{10, \bullet, 13}$ $L_{11, \bullet, 13}$	$L_{13, \bullet}$ $L_{13, \bullet}$ $L_{13, \bullet}$	ρ_{17}	$\rho_{14} \rho_{15}$ $\rho_{16} \rho_{18}$ ρ_{19} ρ_{20}	$L_{3, \bullet, 5}$ $L_{7, \bullet, 9}$ b c $L_{11, \bullet, 13}$ $L_{5, \bullet}$ $L_{9, \bullet, 13} L_{13, \bullet}$ $L_{10, \bullet, 13}$ $L_{11, \bullet, 13}$
$\rightarrow^{\rho_{21}}_{\text{cfgEsplit}, G'}$	$L_{11, \bullet, 13} L_{13, \bullet}$	$L_{1, \square}$ $L_{5, \bullet}$ $L_{9, \bullet, 13}$ $L_{10, \bullet, 13}$	$L_{13, \bullet}$ $L_{13, \bullet}$	ρ_{21}, ρ_{17}	$\rho_{14} \rho_{15}$ $\rho_{16} \rho_{18}$ ρ_{19} ρ_{20}	$L_{3, \bullet, 5}$ $L_{7, \bullet, 9}$ b c $L_{5, \bullet}$ $L_{9, \bullet, 13} L_{13, \bullet}$ $L_{10, \bullet, 13}$ $L_{11, \bullet, 13}$
<i>continued</i>						

<i>continued</i>							
$\rightarrow_{\text{cfgEsplit}, G'}^{\rho_{10}}$	$L_{11, \bullet, 13} L_{13, \bullet}$	$L_{1, \square}$ $L_{5, \bullet}$ $L_{8, \bullet, 9}$ $L_{9, \bullet, 13}$ $L_{10, \bullet, 13}$	$L_{9, \bullet, 13} L_{13, \bullet}$ $L_{13, \bullet}$ $L_{13, \bullet}$	ρ_{21}, ρ_{17}	$\rho_{14} \rho_{15}$ $\rho_{16} \rho_{18} \rho_{10}$ ρ_{19} ρ_{20}	$L_{3, \bullet, 5}$ a $L_{8, \bullet, 9}$ b c	$L_{5, \bullet}$ $L_{8, \bullet, 9} L_{9, \bullet, 13} L_{13, \bullet}$ $L_{10, \bullet, 13}$ $L_{11, \bullet, 13}$
$\rightarrow_{\text{cfgEsplit}, G'}^{\rho_{11}}$	$L_{11, \bullet, 13} L_{13, \bullet}$	$L_{1, \square}$ $L_{5, \bullet}$ $L_{9, \bullet, 13}$ $L_{10, \bullet, 13}$	$L_{13, \bullet}$ $L_{13, \bullet}$	ρ_{11} ρ_{21}, ρ_{17}	$\rho_{14} \rho_{15}$ $\rho_{16} \rho_{18} \rho_{10}$ ρ_{19} ρ_{20}	$L_{3, \bullet, 5}$ a b c	$L_{5, \bullet}$ $L_{8, \bullet, 9} L_{9, \bullet, 13} L_{13, \bullet}$ $L_{10, \bullet, 13}$ $L_{11, \bullet, 13}$
$\rightarrow_{\text{cfgEsplit}, G'}^{\rho_{12}}$	$L_{11, \bullet, 13} L_{13, \bullet}$	$L_{1, \square}$ $L_{4, \bullet, 5}$ $L_{5, \bullet}$ $L_{9, \bullet, 13}$ $L_{10, \bullet, 13}$	$L_{5, \bullet}$ $L_{13, \bullet}$ $L_{13, \bullet}$	ρ_{11} ρ_{21}, ρ_{17}	$\rho_{14} \rho_{15} \rho_{12}$ $\rho_{16} \rho_{18} \rho_{10}$ ρ_{19} ρ_{20}	e $L_{4, \bullet, 5}$ a b c	$L_{4, \bullet, 5} L_{5, \bullet}$ $L_{8, \bullet, 9} L_{9, \bullet, 13} L_{13, \bullet}$ $L_{10, \bullet, 13}$ $L_{11, \bullet, 13}$
$\rightarrow_{\text{cfgEsplit}, G'}^{\rho_{13}}$	$L_{11, \bullet, 13} L_{13, \bullet}$	$L_{1, \square}$ $L_{4, \bullet, 5}$ $L_{5, \bullet}$ $L_{9, \bullet, 13}$ $L_{10, \bullet, 13}$	$L_{13, \bullet}$ $L_{13, \bullet}$	ρ_{13} ρ_{11} ρ_{21}, ρ_{17}	$\rho_{14} \rho_{15} \rho_{12}$ $\rho_{16} \rho_{18} \rho_{10}$ ρ_{19} ρ_{20}	e a b c	$L_{4, \bullet, 5} L_{5, \bullet}$ $L_{8, \bullet, 9} L_{9, \bullet, 13} L_{13, \bullet}$ $L_{10, \bullet, 13}$ $L_{11, \bullet, 13}$

Bibliography

- [1] Martín Abadi, Leslie Lamport, and Pierre Wolper. “Realizable and Unrealizable Specifications of Reactive Systems”. In: *Automata, Languages and Programming, 16th International Colloquium, ICALP89, Stresa, Italy, July 11-15, 1989, Proceedings*. Ed. by Giorgio Ausiello, Mariangiola Dezani-Ciancaglini, and Simona Ronchi Della Rocca. Vol. 372. Lecture Notes in Computer Science. Springer, 1989, pp. 1–17. DOI: 10.1007/BFb0035748 (cit. on p. 168).
- [2] T. F. Abdelzaher, J. A. Stankovic, Chenyang Lu, Ronghua Zhang, and Ying Lu. “Feedback performance control in software services”. In: *IEEE Control Systems* 23.3 (June 2003), pp. 74–90. DOI: 10.1109/MCS.2003.1200252 (cit. on p. 170).
- [3] Alfred Vaino Aho and Jeffrey David Ullman. “A Technique for Speeding up LR(k) Parsers”. In: *SIAM Journal on Computing (SICOMP)* 2.2 (1973), pp. 106–127. DOI: 10.1137/0202010 (cit. on p. 214).
- [4] Alfred Vaino Aho and Jeffrey David Ullman. “Optimization of LR(k) Parsers”. In: *Journal of Computer and System Sciences* 6.6 (1972), pp. 573–602. DOI: 10.1016/S0022-0000(72)80031-X (cit. on p. 214).
- [5] Alfred Vaino Aho and Jeffrey David Ullman. *The Theory of Parsing, Translation, and Compiling*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1972. ISBN: 0-13-914556-7 (cit. on p. 73).
- [6] Fabian Akesson and Vahidi Flordal. *Supremica*. 2017. URL: <http://supremica.org/> (cit. on pp. 164, 172).
- [7] Université Grenoble Alpes. *Lustre Programming Language*. 2015. URL: www-verimag.imag.fr/The-Lustre-Programming-Language.html (cit. on p. 207).
- [8] Karine Altisen, Aurélie Clodic, Florence Maraninchi, and Éric Rutten. “Using Controller-Synthesis Techniques to Build Property-Enforcing Layers”. In: *Programming Languages and Systems, 12th European Symposium on Programming, ESOP 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*. Ed. by Pierpaolo Degano. Vol. 2618. Lecture Notes in Computer Science. Springer, 2003, pp. 174–188. DOI: 10.1007/3-540-36575-3_13 (cit. on p. 167).
- [9] Rajeev Alur, Marcelo Arenas, Pablo Barceló, Kousha Etessami, Neil Immerman, and Leonid Libkin. “First-Order and Temporal Logics for Nested Words”. In: *Logical Methods in Computer Science* 4.4 (2008). DOI: 10.2168/LMCS-4(4:11)2008 (cit. on p. 166).

- [10] Rajeev Alur, Costas Courcoubetis, Nicolas Halbwachs, Thomas A. Henzinger, Pei-Hsin Ho, Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. "The Algorithmic Analysis of Hybrid Systems". In: *Theoretical Computer Science* 138.1 (1995), pp. 3–34. DOI: 10.1016/0304-3975(94)00202-T (cit. on p. 111).
- [11] Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger, and Pei-Hsin Ho. "Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems". In: *Hybrid Systems*. Ed. by Robert L. Grossman, Anil Nerode, Anders P. Ravn, and Hans Rischel. Vol. 736. Lecture Notes in Computer Science. Springer, 1992, pp. 209–229. DOI: 10.1007/3-540-57318-6_30 (cit. on pp. 164, 217).
- [12] Rajeev Alur and David Lansing Dill. "The Theory of Timed Automata". In: *Real-Time: Theory in Practice, REX Workshop, Mook, The Netherlands, June 3-7, 1991, Proceedings*. Ed. by J. W. de Bakker, Cornelis Huizing, Willem-Paul de Roever, and Grzegorz Rozenberg. Vol. 600. Lecture Notes in Computer Science. Springer, 1991, pp. 45–73. DOI: 10.1007/BFb0031987 (cit. on pp. 111, 164).
- [13] Rajeev Alur, Kousha Etessami, and Parthasarathy Madhusudan. "A Temporal Logic of Nested Calls and Returns". In: *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*. Ed. by Kurt Jensen and Andreas Podelski. Vol. 2988. Lecture Notes in Computer Science. Springer, 2004, pp. 467–481. DOI: 10.1007/978-3-540-24730-2_35 (cit. on pp. 166, 210, 217).
- [14] Rajeev Alur and Parthasarathy Madhusudan. "Adding Nesting Structure to Words". In: *Developments in Language Theory, 10th International Conference, DLT 2006, Santa Barbara, CA, USA, June 26-29, 2006, Proceedings*. Ed. by Oscar H. Ibarra and Zhe Dang. Vol. 4036. Lecture Notes in Computer Science. Springer, 2006, pp. 1–13. DOI: 10.1007/11779148_1 (cit. on p. 216).
- [15] Rajeev Alur and Parthasarathy Madhusudan. "Adding nesting structure to words". In: *The Journal of the ACM (JACM)* 56.3 (2009), pp. 1–43. DOI: 10.1145/1516512.1516518 (cit. on p. 216).
- [16] Rajeev Alur and Parthasarathy Madhusudan. "Visibly pushdown languages". In: *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004*. Ed. by László Babai. ACM, 2004, pp. 202–211. ISBN: 1-58113-852-0. DOI: 10.1145/1007352.1007390 (cit. on pp. 216, 221).
- [17] Henrik Reif Andersen. "Partial Model Checking (Extended Abstract)". In: *Proceedings, 10th Annual IEEE Symposium on Logic in Computer Science, San Diego, California, USA, June 26-29, 1995*. IEEE Computer Society, 1995, pp. 398–407. ISBN: 0-8186-7050-9. DOI: 10.1109/LICS.1995.523274 (cit. on p. 171).
- [18] Henrik Reif Andersen, Colin Stirling, and Glynn Winskel. "A Compositional Proof System for the Modal μ -Calculus". In: *Proceedings of the Ninth Annual Symposium on Logic in Computer Science (LICS '94), Paris, France, July 4-7, 1994*. IEEE Computer Society, 1994, pp. 144–153. ISBN: 0-8186-6310-3. DOI: 10.1109/LICS.1994.316076 (cit. on p. 171).
- [19] Henrik Reif Andersen and Glynn Winskel. "Compositional Checking of Satisfaction". In: *Computer Aided Verification, 3rd International Workshop, CAV '91, Aalborg, Denmark, July, 1-4, 1991, Proceedings*. Ed. by Kim Guldstrand Larsen and Arne Skou. Vol. 575. Lecture Notes in Computer Science. Springer, 1991, pp. 24–36. DOI: 10.1007/3-540-55179-4_4 (cit. on p. 171).
- [20] Jørgen H. Andersen, Kåre J. Kristoffersen, Kim Guldstrand Larsen, and Jesper Nierdermann. "Automatic Synthesis of Real Time Systems". In: *Automata, Languages and Programming, 22nd International Colloquium, ICALP95, Szeged, Hungary, July 10-14, 1995, Proceedings*. Ed. by Zoltán Fülöp and Ferenc Gécseg. Vol. 944. Lecture Notes in Computer Science. Springer, 1995, pp. 535–546. DOI: 10.1007/3-540-60084-1_103 (cit. on pp. 171, 174, 175).

- [21] Charles André. *SyncCharts Programming Language*. 2005. URL: <http://www-sop.inria.fr/members/Charles.Andre/SyncCharts/> (cit. on p. 207).
- [22] Marco Antoniotti and Bud Mishra. “Discrete Events Models + Temporal Logic = Supervisory Controller: Automatic Synthesis of Locomotion Controllers”. In: *Proceedings of the 1995 International Conference on Robotics and Automation, Nagoya, Aichi, Japan, May 21-27, 1995*. IEEE Computer Society, 1995, pp. 1441–1446. DOI: 10.1109/ROBOT.1995.525480 (cit. on p. 172).
- [23] Alasdair Armstrong, Georg Struth, and Tjark Weber. *Kleene Algebra*. 2013. URL: http://www.isa-afp.org/entries/Kleene_Algebra.shtml (cit. on p. 187).
- [24] Tamarah Arons, Jozef Hooman, Hillel Kugler, Amir Pnueli, and Mark van der Zwaag. “Deductive Verification of UML Models in TLPVS”. In: *UML’04 - The Unified Modelling Language: Modelling Languages and Applications. 7th International Conference, Lisbon, Portugal, October 11-15, 2004. Proceedings*. Ed. by Thomas Baar, Alfred Strohmeier, Ana M. D. Moreira, and Stephen J. Mellor. Vol. 3273. Lecture Notes in Computer Science. Springer, 2004, pp. 335–349. DOI: 10.1007/978-3-540-30187-5_24 (cit. on p. 168).
- [25] Anthony Auer, Jürgen Dingel, and Karen Rudie. “Concurrency control generation for dynamic threads using discrete-event systems”. In: *Science of Computer Programming* 82 (2014), pp. 22–43. DOI: 10.1016/j.scico.2013.01.007 (cit. on p. 170).
- [26] Roland Axelsson, Matthew Hague, Stephan Kreutzer, Martin Lange, and Markus Latte. “Extended Computation Tree Logic”. In: *Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings*. Ed. by Christian G. Fermüller and Andrei Voronkov. Vol. 6397. Lecture Notes in Computer Science. Springer, 2010, pp. 67–81. DOI: 10.1007/978-3-642-16242-8_6 (cit. on p. 166).
- [27] Jos C. M. Baeten, Bert van Beek, Allan van Hulst, and Jasen Markovski. “A Process Algebra for Supervisory Coordination”. In: *Proceedings First International Workshop on Process Algebra and Coordination, PACO 2011, Reykjavik, Iceland, 9th June 2011*. Ed. by Luca Aceto and Mohammad Reza Mousavi. Vol. 60. EPTCS. 2011, pp. 36–55. DOI: 10.4204/EPTCS.60.3 (cit. on p. 164).
- [28] Jos C. M. Baeten, Dirk A. van Beek, B. Luttik, Jasen Markovski, and J. E. Rooda. “A process-theoretic approach to supervisory control theory”. In: *Proceedings of the 2011 American Control Conference*. June 2011, pp. 4496–4501. DOI: 10.1109/ACC.2011.5990831 (cit. on p. 164).
- [29] Fabio Bagagiolo, Dario Bauso, Rosario Maccisio, and Marta Zoppello. “Game Theoretic Decentralized Feedback Controls in Markov Jump Processes”. In: *Journal of Optimization Theory and Applications* 173.2 (2017), pp. 704–726. DOI: 10.1007/s10957-017-1078-3 (cit. on p. 168).
- [30] Henry Baker. “Petri Nets and Languages”. In: *Computation Structures Group - Memo No. 68* (May 1972) (cit. on p. 177).
- [31] Ayca Balkan, Moshe Ya’akov Vardi, and Paulo Tabuada. “Mode-Target Games: Reactive Synthesis for Control Applications”. In: *IEEE Transactions on Automatic Control* 63.1 (2018), pp. 196–202. DOI: 10.1109/TAC.2017.2722960 (cit. on p. 168).
- [32] Ramon Barakat, Ruediger Berndt, Christian Breindl, Christine Baier, Tobias Barthel, Christoph Doerr, Marc Duevel, Norman Franchi, Stefan Goetz, Rainer Hartmann, Jochen Hellenschmidt, Stefan Jacobi, Matthias Leinfelder, Tomás Masopust, Michael Meyer, Andreas Mohr, Thomas Moor, Mihai Musunoi, Bernd Opitz, Katja Palaic, Irmgard Petzoldt, Sebastian Perk, Thomas Rempel, Daniel Ritter, Berno Schlein, Ece Schmidt, Klaus Werner Schmidt, Anne-Kathrin Schmuck, Sven Schneider, Matthias Singer, Ulas Turan, Christian Wamser, Zhengying Wang, Thomas Wittmann, Shi Xiaoxun, Yang Yi, Jorgos Zaddach, Hao Zhou, Christian Zwick, and et al. *libFAUDES*. 2018. URL: http://www.rt.eei.uni-erlangen.de/FGdes/faudes/faudes_about.html (cit. on pp. 156, 172, 211).

- [33] Francesco Basile, Pasquale Chiacchio, and Alessandro Giua. “Suboptimal supervisory control of Petri nets in presence of uncontrollable transitions via monitor places”. In: *Automatica* 42.6 (2006), pp. 995–1004. DOI: 10.1016/j.automatica.2006.02.003 (cit. on p. 179).
- [34] Dirk A. van Beek, Wan Fokkink, D. Hendriks, A. Hofkamp, Jasen Markovs-ki, J. M. van de Mortel-Fronczak, and Michel A. Reniers. “CIF 3: Model-Based Engineering of Supervisory Controllers”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*. Ed. by Erika Ábrahám and Klaus Havelund. Vol. 8413. Lecture Notes in Computer Science. Springer, 2014, pp. 575–580. DOI: 10.1007/978-3-642-54862-8_48 (cit. on pp. 172, 174).
- [35] Dirk A. van Beek, Wan Fokkink, D. Hendriks, A. Hofkamp, Jasen Markovs-ki, J. M. van de Mortel-Fronczak, and Michel A. Reniers. *CIF3*. 2018. URL: <http://cif.se.wtb.tue.nl/> (cit. on pp. 172, 174).
- [36] Jesper Bengtson. “Formalizing Process Calculi”. PhD thesis. Uppsala University, 2010 (cit. on p. 187).
- [37] Harsh Beohar, Pieter J. L. Cuijpers, and Jos C. M. Baeten. “Design of asynchronous supervisors”. In: *CoRR* (2009). URL: <http://arxiv.org/abs/0910.0868> (cit. on p. 164).
- [38] Stefan Berghofer and Markus Reiter. *Formalizing the Logic-Automaton Connection*. 2009. URL: <http://www.isa-afp.org/entries/Presburger-Automata.shtml> (cit. on p. 186).
- [39] Gerard Berry, Amar Bouali, Yannis Bres, Loic Henry-gréard, Jean-Paul Marmorat, Fabrice Peix, Dumitru Potop-Butucaru, Annie Ressouche, Robert De_Simone, Xavier Thirioux, and Eric Vecchie. *Esterel Programming Language*. 2012. URL: http://www-sop.inria.fr/esterel.org/files/v5_92/ (cit. on p. 207).
- [40] Nicolas Berthier and Hervé Marchand. “Deadlock-free discrete controller synthesis for infinite state systems”. In: *54th IEEE Conference on Decision and Control, CDC 2015, Osaka, Japan, December 15-18, 2015*. IEEE, 2015, pp. 1000–1007. ISBN: 978-1-4799-7886-1. DOI: 10.1109/CDC.2015.7402003 (cit. on p. 172).
- [41] Nicolas Berthier and Hervé Marchand. “Discrete Controller Synthesis for Infinite State Systems with ReaX”. In: *12th International Workshop on Discrete Event Systems, WODES 2014, Cachan, France, May 14-16, 2014*. Ed. by Jean-Jacques Lesage, Jean-Marc Faure, José E. R. Cury, and Bengt Lennartson. International Federation of Automatic Control, 2014, pp. 46–53. ISBN: 978-3-902823-61-8. DOI: 10.3182/20140514-3-FR-4046.00099 (cit. on p. 172).
- [42] Nicolas Berthier and Hervé Marchand. *reaX*. 2018. URL: <http://reakt.gforge.inria.fr/> (cit. on p. 172).
- [43] Benjamin Bittner, Marco Bozzano, Alessandro Cimatti, and Xavier Olive. “Symbolic Synthesis of Observability Requirements for Diagnosability”. In: *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada*. Ed. by Jörg Hoffmann and Bart Selman. AAAI Press, 2012. URL: <http://www.aaai.org/ocs/index.php/AAAI/AAAI12/paper/view/5056> (cit. on p. 173).
- [44] Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Yaniv Sa’ar. “Synthesis of Reactive(1) designs”. In: *Journal of Computer and System Sciences* 78.3 (2012), pp. 911–938. DOI: 10.1016/j.jcss.2011.08.007 (cit. on p. 170).
- [45] Gregor von Bochmann. “Submodule Construction and Supervisory Control: A Generalization”. In: *Implementation and Application of Automata, 6th International Conference, CIAA 2001, Pretoria, South Africa, July 23-25, 2001, Revised Papers*. Ed. by Bruce W. Watson and Derick Wood. Vol. 2494. Lecture Notes in Computer Science. Springer, 2001, pp. 27–39. DOI: 10.1007/3-540-36390-4_3 (cit. on pp. 146, 170, 171).

- [46] Gregor von Bochmann. "Submodule Construction for Specifications with Input Assumptions and Output Guarantees". In: *Formal Techniques for Networked and Distributed Systems - FORTE 2002, 22nd IFIP WG 6.1 International Conference Houston, Texas, USA, November 11-14, 2002, Proceedings*. Ed. by Doron A. Peled and Moshe Ya'akov Vardi. Vol. 2529. Lecture Notes in Computer Science. Springer, 2002, pp. 17-33. DOI: 10.1007/3-540-36135-9_2 (cit. on p. 171).
- [47] Gregor von Bochmann. "Using First-Order Logic to Reason about Submodule Construction". In: *Formal Techniques for Distributed Systems, Joint 11th IFIP WG 6.1 International Conference FMOODS 2009 and 29th IFIP WG 6.1 International Conference FORTE 2009, Lisboa, Portugal, June 9-12, 2009. Proceedings*. Ed. by David Lee, Antónia Lopes, and Arnd Poetzsch-Heffter. Vol. 5522. Lecture Notes in Computer Science. Springer, 2009, pp. 213-218. DOI: 10.1007/978-3-642-02138-1_14 (cit. on p. 171).
- [48] Nicky O. Bodentien, Jacob Vestergaard, Jakob Friis, Kåre J. Kristoffersen, and Kim Guldstrand Larsen. *Verification of State/Event Systems by Quotienting*. rs RS-99-41. 17 pp. Presented at *Nordic Workshop in Programming Theory*, Uppsala, Sweden, October 6-8, 1999. iesd: brics, Dec. 1999 (cit. on pp. 171, 172).
- [49] Maksym Bortin. *A formalisation of the Cocke-Younger-Kasami algorithm*. 2016. URL: <http://www.isa-afp.org/entries/CYK.shtml> (cit. on p. 186).
- [50] Ahmed Bouajjani. "Languages, Rewriting Systems, and Verification of Infinite-State Systems". In: *Automata, Languages and Programming, 28th International Colloquium, ICALP 2001, Crete, Greece, July 8-12, 2001, Proceedings*. Ed. by Fernando Orejas, Paul G. Spirakis, and Jan van Leeuwen. Vol. 2076. Lecture Notes in Computer Science. Springer, 2001, pp. 24-39. DOI: 10.1007/3-540-48224-5_3 (cit. on p. 168).
- [51] Ahmed Bouajjani, Rachid Echahed, and Peter Habermehl. "On the Verification Problem of Nonregular Properties for Nonregular Processes". In: *Proceedings, 10th Annual IEEE Symposium on Logic in Computer Science, San Diego, California, USA, June 26-29, 1995*. IEEE Computer Society, 1995, pp. 123-133. ISBN: 0-8186-7050-9. DOI: 10.1109/LICS.1995.523250 (cit. on p. 168).
- [52] Ahmed Bouajjani, Rachid Echahed, and Peter Habermehl. "Verifying Infinite State Processes with Sequential and Parallel Composition". In: *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*. Ed. by Ron K. Cytron and Peter Lee. ACM Press, 1995, pp. 95-106. ISBN: 0-89791-692-1. DOI: 10.1145/199448.199470 (cit. on p. 168).
- [53] Ahmed Bouajjani, Rachid Echahed, and Riadh Robbana. "Verification of Context-Free Timed Systems Using Linear Hybrid Observers". In: *Computer Aided Verification, 6th International Conference, CAV '94, Stanford, California, USA, June 21-23, 1994, Proceedings*. Ed. by David Lansing Dill. Vol. 818. Lecture Notes in Computer Science. Springer, 1994, pp. 118-131. DOI: 10.1007/3-540-58179-0_48 (cit. on p. 168).
- [54] Ahmed Bouajjani, Rachid Echahed, and Riadh Robbana. "Verification of Nonregular Temporal Properties for Context-Free Processes". In: *CONCUR '94, Concurrency Theory, 5th International Conference, Uppsala, Sweden, August 22-25, 1994, Proceedings*. Ed. by Bengt Jonsson and Joachim Parrow. Vol. 836. Lecture Notes in Computer Science. Springer, 1994, pp. 81-97. DOI: 10.1007/978-3-540-48654-1_8 (cit. on p. 168).
- [55] Tayeb Bouhadiba, Quentin Sabah, Gwenaël Delaval, and Éric Rutten. "Synchronous control of reconfiguration in fractal component-based systems: a case study". In: *Proceedings of the 11th International Conference on Embedded Software, EMSOFT 2011, part of the Seventh Embedded Systems Week, ESWeek 2011, Taipei, Taiwan, October 9-14, 2011*. Ed. by Samarjit Chakraborty, Ahmed Jerraya, Sanjoy K. Baruah, and Sebastian Fischmeister. ACM, 2011, pp. 309-318. ISBN: 978-1-4503-0714-7. DOI: 10.1145/2038642.2038690 (cit. on p. 172).
- [56] Sean E. Bourdon, Mark Lawford, and Walter Murray Wonham. "Robust nonblocking supervisory control of discrete-event systems". In: *IEEE Transactions on Automatic Control* 50.12 (2005), pp. 2015-2021. DOI: 10.1109/TAC.2005.860237 (cit. on pp. 166, 174).

- [57] Víctor A. Braberman, Nicolás D'Ippolito, Jeff Kramer, Daniel Sykes, and Sebastián Uchitel. "MORPH: a reference architecture for configuration and behaviour self-adaptation". In: *Proceedings of the 1st International Workshop on Control Theory for Software Engineering, CTSE@SIGSOFT FSE 2015, Bergamo, Italy, August 31 - September 04, 2015*. Ed. by Antonio Filieri and Martina Maggio. ACM, 2015, pp. 9–16. ISBN: 978-1-4503-3814-1. DOI: 10.1145/2804337.2804339 (cit. on p. 170).
- [58] Bertil A. Brandin and Walter Murray Wonham. "Supervisory control of timed discrete-event systems". In: *IEEE Transactions on Automatic Control* 39.2 (Feb. 1994), pp. 329–342. DOI: 10.1109/9.272327 (cit. on pp. 171, 173).
- [59] Bertil A. Brandin, Walter Murray Wonham, and Beno Benhabib. "Manufacturing cell supervisory control-a timed discrete event system approach". In: *Proceedings of the 1992 IEEE International Conference on Robotics and Automation, Nice, France, May 12-14, 1992*. IEEE, 1992, pp. 931–936. ISBN: 0-8186-2720-4. DOI: 10.1109/ROBOT.1992.220177 (cit. on pp. 173, 174).
- [60] Yitzhak Brave and Michael Heymann. "Formulation and control of real time discrete event processes". In: *Proceedings of the 27th IEEE Conference on Decision and Control*. Dec. 1988, pp. 1131–1132. DOI: 10.1109/CDC.1988.194493 (cit. on pp. 173, 217).
- [61] Yitzhak Brave and Michael Heymann. "On optimal attraction in discrete-event processes". In: *Information Sciences* 67.3 (1993), pp. 245–276. DOI: 10.1016/0020-0255(93)90075-W (cit. on p. 165).
- [62] Julian Brunner. *Transition Systems and Automata*. 2017. URL: http://isa-afp.org/entries/Transition_Systems_and_Automata.html (cit. on p. 187).
- [63] Giorgio Bruno and Giuseppe Marchetto. "Process-Translatable Petri Nets for the Rapid Prototyping of Process Control Systems". In: *IEEE Transactions on Software Engineering* 12.2 (1986), pp. 346–357. DOI: 10.1109/TSE.1986.6312948 (cit. on p. 178).
- [64] Julius Richard Büchi. "State-Strategies for Games in $F_{\sigma,\delta}$ $G_{\delta\sigma}$ ". In: *The Journal of Symbolic Logic* 48.4 (1983), pp. 1171–1198. DOI: 10.2307/2273681 (cit. on p. 168).
- [65] Julius Richard Büchi. "Symposium on Decision Problems: On a Decision Method in Restricted Second Order Arithmetic". In: *Logic, Methodology and Philosophy of Science Proceeding of the 1960 International Congress*. Ed. by Ernest Nagel, Patrick Suppes, and Alfred Tarski. Vol. 44. Studies in Logic and the Foundations of Mathematics. Elsevier, 1966, pp. 1–11. DOI: 10.1016/S0049-237X(09)70564-6 (cit. on pp. 111, 164).
- [66] Julius Richard Buchi and Lawrence Hugh Landweber. "Solving Sequential Conditions by Finite-State Strategies". In: *Transactions of the American Mathematical Society* 138 (1969), pp. 295–311. ISSN: 00029947. DOI: 10.2307/1994916. URL: <http://www.jstor.org/stable/1994916> (cit. on p. 1).
- [67] Maria Paola Cabasino, Alessandro Giua, Laura Marcias, and Carla Seatzu. "A comparison among tools for the diagnosability of discrete event systems". In: *2012 IEEE International Conference on Automation Science and Engineering, CASE 2012, Seoul, Korea (South), August 20-24, 2012*. IEEE, 2012, pp. 218–223. ISBN: 978-1-4673-0429-0. DOI: 10.1109/CoASE.2012.6386425 (cit. on p. 173).
- [68] Maria Paola Cabasino, Alessandro Giua, and Carla Seatzu. "Diagnosability of Discrete-Event Systems Using Labeled Petri Nets". In: *IEEE Transactions on Automation Science and Engineering* 11.1 (2014), pp. 144–153. DOI: 10.1109/TASE.2013.2289360 (cit. on p. 173).
- [69] Kai Cai and Walter Murray Wonham. "Supervisor Localization of Discrete-Event Systems based on State Tree Structures". In: *CoRR* (2013). URL: <http://arxiv.org/abs/1306.5441> (cit. on pp. 175, 176).
- [70] Xi-Ren Cao, Guy Cohen, Alessandro Giua, Walter Murray Wonham, and Jan Hendrik van Schuppen. "Unity in Diversity, Diversity in Unity: Retrospective and Prospective Views on Control of Discrete Event Systems". In: *Discrete Event Dynamic Systems* 12.3 (2002), pp. 253–264. DOI: 10.1023/A:1015617431563 (cit. on pp. 8, 172, 222).

- [71] Christos G. Cassandras and Stéphane Lafortune. *Introduction to Discrete Event Systems*. 2nd ed. Springer Link Engineering. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2008. ISBN: 9780387333328 (cit. on pp. 8, 143, 172–174).
- [72] Christos G. Cassandras and Stéphane Lafortune. *UMDES*. 2018. URL: <http://www.swmat.h.org/software/9523> (cit. on p. 172).
- [73] Franck Cassez and François Laroussinie. “Model-Checking for Hybrid Systems by Quotienting and Constraints Solving”. In: *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*. Ed. by E. Allen Emerson and A. Prasad Sistla. Vol. 1855. Lecture Notes in Computer Science. Springer, 2000, pp. 373–388. DOI: 10.1007/10722167_29 (cit. on pp. 171, 172).
- [74] Giovanni Castagnetti, Matteo Piccolo, Tiziano Villa, Nina Yevtushenko, Robert K. Brayton, and Alan Mishchenko. “Automated Synthesis of Protocol Converters with BALM-II”. In: *Software Engineering and Formal Methods - SEFM 2015 Collocated Workshops: ATSE, HOFM, MoKMaSD, and VERY*SCART, York, UK, September 7-8, 2015, Revised Selected Papers*. Ed. by Domenico Bianculli, Radu Calinescu, and Bernhard Rumpe. Vol. 9509. Lecture Notes in Computer Science. Springer, 2015, pp. 281–296. DOI: 10.1007/978-3-662-49224-6_23 (cit. on p. 171).
- [75] Jacques Chabin and Pierre Réty. “Visibly Pushdown Languages and Term Rewriting”. In: *Frontiers of Combining Systems, 6th International Symposium, FroCoS 2007, Liverpool, UK, September 10-12, 2007, Proceedings*. Ed. by Boris Konev and Frank Wolter. Vol. 4720. Lecture Notes in Computer Science. Springer, 2007, pp. 252–266. DOI: 10.1007/978-3-54-0-74621-8_17 (cit. on p. 216).
- [76] Wujie Chao, Yongmei Gan, Zhaoan Wang, and Walter Murray Wonham. “Modular supervisory control and coordination of state tree structures”. In: *International Journal of Control* 86.1 (2013), pp. 9–21. DOI: 10.1080/00207179.2012.715754 (cit. on pp. 175, 176).
- [77] Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, eds. *Software Engineering for Self-Adaptive Systems [outcome of a Dagstuhl Seminar]*. Vol. 5525. Lecture Notes in Computer Science. Springer, 2009. DOI: 10.1007/978-3-642-02161-9 (cit. on p. 169).
- [78] Kwang-Hyun Cho and Jong-Tae Lim. “Fault-tolerant supervisory control of discrete event dynamical systems”. In: *International Journal of Systems Science* 28.10 (1997), pp. 1001–1009. DOI: 10.1080/00207729708929463 (cit. on p. 173).
- [79] Noam Chomsky. “On Certain Formal Properties of Grammars”. In: *Information and Control* 2.2 (1959), pp. 137–167. DOI: 10.1016/S0019-9958(59)90362-6 (cit. on p. 111).
- [80] Noam Chomsky. “Three models for the description of language”. In: *IRE Transactions on Information Theory* 2.3 (1956), pp. 113–124. DOI: 10.1109/TIT.1956.1056813 (cit. on p. 27).
- [81] Alonzo Church. *Applications of recursive arithmetic to the problem of circuit synthesis*. Vol. 1. In *Summaries of the Summer Institute of Symbolic Logic*. Cornell University, 1957, pp. 3–50 (cit. on pp. 1, 219).
- [82] Randy Cieslak, C. Desclaux, Ayman S. Fawaz, and Pravin Varaiya. “Supervisory control of discrete-event processes with partial observations”. In: *IEEE Transactions on Automatic Control* 33.3 (Mar. 1988), pp. 249–260. DOI: 10.1109/9.402 (cit. on p. 173).
- [83] Edmund M. Clarke and E. Allen Emerson. “Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic”. In: *Logics of Programs, Workshop, Yorktown Heights, New York, May 1981*. Ed. by Dexter Kozen. Vol. 131. Lecture Notes in Computer Science. Springer, 1981, pp. 52–71. DOI: 10.1007/BFb0025774 (cit. on p. 171).
- [84] Edmund Melson Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. “Counterexample-Guided Abstraction Refinement”. In: *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*. Ed. by E. Allen Emerson and A. Prasad Sistla. Vol. 1855. Lecture Notes in Computer Science. Springer, 2000, pp. 154–169. DOI: 10.1007/10722167_15 (cit. on p. 167).

- [85] Hubert Comon-Lundh, Florent Jacquemard, and Nicolas Perrin. “Visibly Tree Automata with Memory and Constraints”. In: *Logical Methods in Computer Science* 4.2 (2008). DOI: 10.2168/LMCS-4(2:8)2008 (cit. on p. 216).
- [86] Roger C. Conant. “Channel Capacity of Moore Automata”. In: *Information and Control* 12.5/6 (1968), pp. 453–465. DOI: 10.1016/S0019-9958(68)90496-8 (cit. on p. 164).
- [87] José Eduardo Ribeiro Cury and Bruce H. Krogh. “Design of robust supervisors for discrete event systems with infinite behaviors”. In: *Proceedings of 35th IEEE Conference on Decision and Control*. Vol. 2. Dec. 1996, pp. 2219–2224. DOI: 10.1109/CDC.1996.572972 (cit. on p. 166).
- [88] José Eduardo Ribeiro Cury and Bruce H. Krogh. “Robustness of supervisors for discrete-event systems”. In: *IEEE Transactions on Automatic Control* 44.2 (1999), pp. 376–379. DOI: 10.1109/9.746270 (cit. on p. 166).
- [89] Nicolás D’Ippolito, Víctor A. Braberman, Jeff Kramer, Jeff Magee, Daniel Sykes, and Sebastián Uchitel. “Hope for the best, prepare for the worst: multi-tier control for adaptive systems”. In: *36th International Conference on Software Engineering, ICSE ’14, Hyderabad, India - May 31 - June 07, 2014*. Ed. by Pankaj Jalote, Lionel C. Briand, and André van der Hoek. ACM, 2014, pp. 688–699. ISBN: 978-1-4503-2756-5. DOI: 10.1145/2568225.2568264 (cit. on p. 170).
- [90] Nicolás D’Ippolito, Víctor A. Braberman, Nir Piterman, and Sebastián Uchitel. “Synthesizing nonanomalous event-based controllers for liveness goals”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 22.1 (2013), pp. 1–36. DOI: 10.1145/2430536.2430543 (cit. on p. 170).
- [91] Eric Dallal, Daniel Neider, and Paulo Tabuada. “Synthesis of safety controllers robust to unmodeled intermittent disturbances”. In: *55th IEEE Conference on Decision and Control, CDC 2016, Las Vegas, NV, USA, December 12-14, 2016*. IEEE, 2016, pp. 7425–7430. ISBN: 978-1-5090-1837-6. DOI: 10.1109/CDC.2016.7799416 (cit. on p. 166).
- [92] Zhe Dang, Oscar H. Ibarra, Tefvik Bultan, Richard A. Kemmerer, and Jianwen Su. “Binary Reachability Analysis of Discrete Pushdown Timed Automata”. In: *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*. Ed. by E. Allen Emerson and A. Prasad Sistla. Vol. 1855. Lecture Notes in Computer Science. Springer, 2000, pp. 69–84. DOI: 10.1007/10722167_9 (cit. on p. 169).
- [93] Bassel Daou and Gregor von Bochmann. “Generalizing the Submodule Construction Techniques for Extended State Machine Models”. In: *Formal Techniques for Networked and Distributed Systems - FORTE 2006, 26th IFIP WG 6.1 International Conference, Paris, France, September 26-29, 2006*. Ed. by Elie Najm, Jean-François Pradat-Peyre, and Véronique Donzeau-Gouge. Vol. 4229. Lecture Notes in Computer Science. Springer, 2006, pp. 191–195. DOI: 10.1007/11888116_15 (cit. on p. 171).
- [94] Bassel Daou and Gregor von Bochmann. “Submodule Construction for Extended State Machine Models”. In: *Formal Techniques for Networked and Distributed Systems - FORTE 2005, 25th IFIP WG 6.1 International Conference, Taipei, Taiwan, October 2-5, 2005, Proceedings*. Ed. by Farn Wang. Vol. 3731. Lecture Notes in Computer Science. Springer, 2005, pp. 396–410. DOI: 10.1007/11562436_29 (cit. on p. 171).
- [95] Ajoy Kumar Datta and Sukumar Ghosh. “Synthesis of a Class of Deadlock-Free Petri Nets”. In: *The Journal of the ACM (JACM)* 31.3 (1984), pp. 486–506. DOI: 10.1145/828.322441 (cit. on p. 178).
- [96] Jennifer M. Davoren, Thomas Moor, and Anil Nerode. “Hybrid Control Loops, A/D Maps, and Dynamic Specifications”. In: *Hybrid Systems: Computation and Control, 5th International Workshop, HSCC 2002, Stanford, CA, USA, March 25-27, 2002, Proceedings*. Ed. by Claire Tomlin and Mark R. Greenstreet. Vol. 2289. Lecture Notes in Computer Science. Springer, 2002, pp. 149–163. DOI: 10.1007/3-540-45873-5_14 (cit. on p. 176).

- [97] Gwenaél Delaval. “Modular Distribution and Application to Discrete Controller Synthesis”. In: *Electronic Notes in Theoretical Computer Science* 238.1 (2009), pp. 3–19. DOI: 10.1016/j.entcs.2008.01.003 (cit. on p. 167).
- [98] Gwenaél Delaval, Soguy Mak Karé Gueye, Éric Rutten, and Noël De Palma. “Modular coordination of multiple autonomic managers”. In: *CBSE’14, Proceedings of the 17th International ACM SIGSOFT Symposium on Component-Based Software Engineering (part of CompArch 2014), Marcq-en-Baroeul, Lille, France, June 30 - July 4, 2014*. Ed. by Lionel Seinturier, Eduardo Santana de Almeida, and Jan Carlson. ACM, 2014, pp. 3–12. ISBN: 978-1-4503-2577-6. DOI: 10.1145/2602458.2602465 (cit. on p. 170).
- [99] Gwenaél Delaval, Hervé Marchand, Marc Pouzet, and Éric Rutten. BZR. 2000. URL: <http://heptagon.gforge.inria.fr/> (cit. on p. 172).
- [100] Gwenaél Delaval, Hervé Marchand, and Éric Rutten. “Contracts for modular discrete controller synthesis”. In: *Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems, LCTES 2010, Stockholm, Sweden, April 13-15, 2010*. Ed. by Jaejin Lee and Bruce R. Childers. ACM, 2010, pp. 57–66. ISBN: 978-1-60558-953-4. DOI: 10.1145/1755888.1755898 (cit. on p. 172).
- [101] Franklin Lewis DeRemer. *Practical Translators for LR(k) Languages*. Tech. rep. Cambridge, MA, USA: Massachusetts Institute of Technology, 1969 (cit. on p. 214).
- [102] Franklin Lewis DeRemer. “Simple LR(k) Grammars”. In: *Communications of the ACM* 14.7 (July 1971), pp. 453–460. DOI: 10.1145/362619.362625 (cit. on p. 214).
- [103] Yixin Diao, Neha Gandhi, Joseph L. Hellerstein, Sujay S. Parekh, and Dawn M. Tilbury. “Using MIMO feedback control to enforce policies for interrelated metrics with application to the Apache Web server”. In: *Management Solutions for the New Communications World, 8th IEEE/IFIP Network Operations and Management Symposium, NOMS 2002, Florence, Italy, April 15-19, 2002. Proceedings*. Ed. by Rolf Stadler and Mehmet Ulema. IEEE, 2002, pp. 219–234. ISBN: 0-7803-7383-9. DOI: 10.1109/NOMS.2002.1015566 (cit. on p. 170).
- [104] Abbas Dideban and Hassane Alla. “Feedback control logic synthesis for non safe Petri nets”. In: *13th IFAC Symposium on Information Control Problems in Manufacturing (INCOM09)*. Moscou, Russia, June 2009, pp. 946–951. URL: <https://hal.archives-ouvertes.fr/hal-00463404> (cit. on p. 180).
- [105] David Lansing Dill. *Trace theory for automatic hierarchical verification of speed-independent circuits*. ACM distinguished dissertations. MIT Press, 1989. ISBN: 978-0-262-04101-0 (cit. on p. 168).
- [106] Rayna Dimitrova. “Synthesis and control of infinite-state systems with partial observability”. PhD thesis. Saarland University, 2014. URL: <http://scidok.sulb.uni-saarland.de/volltexte/2014/5946/> (cit. on pp. 168, 173).
- [107] Christopher Dragert, Jürgen Dingel, and Karen Rudie. “Generation of concurrency control code using discrete-event systems theory”. In: *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, Georgia, USA, November 9-14, 2008*. Ed. by Mary Jean Harrold and Gail C. Murphy. ACM, 2008, pp. 146–157. ISBN: 978-1-59593-995-1. DOI: 10.1145/1453101.1453122 (cit. on p. 170).
- [108] Jawad Drissi and Gregor von Bochmann. *Submodule construction for systems of I/O automata*. Tech. rep. Dept. d’IRO, Université de Montréal, School of Information Technology & Engineering, University of Ottawa, 1999 (cit. on p. 171).
- [109] Jawad Drissi and Gregor von Bochmann. *Submodule construction for systems of timed I/O automata*. Tech. rep. Dept. d’IRO, Université de Montréal, School of Information Technology & Engineering, University of Ottawa, Jan. 2000 (cit. on p. 171).
- [110] Jay Clark Earley. “An Efficient Context-free Parsing Algorithm”. PhD thesis. Pittsburgh, PA, USA: Carnegie Mellon University Pittsburgh, 1968 (cit. on p. 157).

- [111] Rüdiger Ehlers, Stéphane Lafortune, Stavros Tripakis, and Moshe Ya'akov Vardi. "Supervisory control and reactive synthesis: a comparative introduction". In: *Discrete Event Dynamic Systems* 27.2 (2017), pp. 209–260. DOI: 10.1007/s10626-015-0223-0 (cit. on p. 168).
- [112] E. Allen Emerson. "Uniform Inevitability is Tree Automaton Ineffable". In: *Information Processing Letters* 24.2 (1987), pp. 77–79. DOI: 10.1016/0020-0190(87)90097-4 (cit. on p. 166).
- [113] Javier Esparza, David Hansel, Peter Rossmanith, and Stefan Schwoon. "Efficient Algorithms for Model Checking Pushdown Systems". In: *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*. Ed. by E. Allen Emerson and A. Prasad Sistla. Vol. 1855. Lecture Notes in Computer Science. Springer, 2000, pp. 232–247. DOI: 10.1007/10722167_20 (cit. on p. 169).
- [114] Martin Fabian and Anders Hellgren. *Descos*. 2018. URL: <http://www.swmath.org/software/1223> (cit. on p. 172).
- [115] Lei Feng and Walter Murray Wonham. "Nonblocking coordination of discrete-event systems by control-flow nets". In: *46th IEEE Conference on Decision and Control, CDC 2007, New Orleans, LA, USA, December 12-14, 2007*. IEEE, 2007, pp. 3375–3380. DOI: 10.1109/CDC.2007.4434160 (cit. on p. 164).
- [116] Lei Feng and Walter Murray Wonham. "Supervisory Control Architecture for Discrete-Event Systems". In: *IEEE Transactions on Automatic Control* 53.6 (2008), pp. 1449–1461. DOI: 10.1109/TAC.2008.927679 (cit. on pp. 175, 176).
- [117] Lei Feng and Walter Murray Wonham. *TCT*. 2006. URL: <http://www.lsv.fr/~fl/cmcweb.html> (cit. on p. 172).
- [118] Lei Feng and Walter Murray Wonham. "TCT: A Computation Tool for Supervisory Control Synthesis". In: *2006 8th International Workshop on Discrete Event Systems*. July 2006, pp. 388–389. DOI: 10.1109/WODES.2006.382399 (cit. on p. 172).
- [119] Lei Feng, Walter Murray Wonham, and P. S. Thiagarajan. "Designing communicating transaction processes by supervisory control theory". In: *Formal Methods in System Design* 30.2 (2007), pp. 117–141. DOI: 10.1007/s10703-006-0023-0 (cit. on p. 175).
- [120] Antonio Filieri, Henry Hoffmann, and Martina Maggio. "Automated design of self-adaptive software with control-theoretical formal guarantees". In: *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*. Ed. by Pankaj Jalote, Lionel C. Briand, and André van der Hoek. ACM, 2014, pp. 299–310. ISBN: 978-1-4503-2756-5. DOI: 10.1145/2568225.2568272 (cit. on p. 170).
- [121] Bernd Finkbeiner. "Synthesis of Reactive Systems". In: *Dependable Software Systems Engineering*. Ed. by Javier Esparza, Orna Grumberg, and Salomon Sickert. Vol. 45. NATO Science for Peace and Security Series - D: Information and Communication Security. IOS Press, 2016, pp. 72–98. ISBN: 978-1-61499-626-2. DOI: 10.3233/978-1-61499-627-9-72. URL: <https://doi.org/10.3233/978-1-61499-627-9-72> (cit. on pp. 1, 219).
- [122] Markus P. J. Fromherz, Lara S. Crawford, and Haitham A. Hindi. "Coordinated Control for Highly Reconfigurable Systems". In: *Hybrid Systems: Computation and Control, 8th International Workshop, HSCC 2005, Zurich, Switzerland, March 9-11, 2005, Proceedings*. Ed. by Manfred Morari and Lothar Thiele. Vol. 3414. Lecture Notes in Computer Science. Springer, 2005, pp. 1–24. DOI: 10.1007/978-3-540-31954-2_1 (cit. on p. 169).
- [123] Benoit Gaudin and Paddy Nixon. "Supervisory control for software runtime exception avoidance". In: *Fifth International C* Conference on Computer Science & Software Engineering, C3S2E '12, Montreal, QC, Canada, June 27-29, 2012*. Ed. by Bipin C. Desai, Emil Vassev, and Sudhir P. Mudur. ACM, 2012, pp. 109–112. ISBN: 978-1-4503-1084-0. DOI: 10.1145/2347583.2347598 (cit. on p. 170).

- [124] Benoit Gaudin, Emil Vassev, Patrick Nixon, and Michael G. Hinchey. "A control theory based approach for self-healing of un-handled runtime exceptions". In: *Proceedings of the 8th International Conference on Autonomic Computing, ICAC 2011, Karlsruhe, Germany, June 14-18, 2011*. Ed. by Hartmut Schmeck, Wolfgang Rosenstiel, Tarek F. Abdelzaher, and Joseph L. Hellerstein. ACM, 2011, pp. 217–220. DOI: 10.1145/1998582.1998633 (cit. on p. 170).
- [125] Viliam Geffert, Carlo Mereghetti, and Beatrice Palano. "More concise representation of regular languages by automata and regular expressions". In: *Information and Computation* 208.4 (2010), pp. 385–394. DOI: 10.1016/j.ic.2010.01.002 (cit. on p. 180).
- [126] Matthew M. Geller and Michael A. Harrison. "On LR(k) Grammars and Languages". In: *Theoretical Computer Science* 4.3 (1977), pp. 245–276. DOI: 10.1016/0304-3975(77)90013-5 (cit. on p. 157).
- [127] Matthew M. Geller, Harry B. Hunt III, Thomas G. Szymanski, and Jeffrey David Ullman. "Economy of Descriptions by Parsers, DPDA's, and PDA's". In: *16th Annual Symposium on Foundations of Computer Science, Berkeley, California, USA, October 13-15, 1975*. IEEE Computer Society, 1975, pp. 122–127. DOI: 10.1109/SFCS.1975.12 (cit. on p. 157).
- [128] Sona Ghahremani, Holger Giese, and Thomas Vogel. "Efficient Utility-Driven Self-Healing Employing Adaptation Rules for Large Dynamic Architectures". In: *2017 IEEE International Conference on Autonomic Computing, ICAC 2017, Columbus, OH, USA, July 17-21, 2017*. Ed. by Xiaorui Wang, Christopher Stewart, and Hui Lei. IEEE Computer Society, 2017, pp. 59–68. ISBN: 978-1-5386-1762-5. DOI: 10.1109/ICAC.2017.35 (cit. on pp. 146, 170).
- [129] Seymour Ginsburg and Sheila Adele Greibach. "Deterministic Context Free Languages". In: *Information and Control* 9.6 (1966), pp. 620–648. DOI: 10.1016/S0019-9958(66)80019-0 (cit. on pp. 18, 164).
- [130] Alessandro Giua. "Petri Nets as Discrete Event Models for Supervisory Control". PhD thesis. Rensselaer Polytechnique Institute, July 1992 (cit. on p. 178).
- [131] Alessandro Giua and Frank DiCesare. "Blocking and controllability of Petri nets in supervisory control". In: *IEEE Transactions on Automatic Control* 39.4 (Apr. 1994), pp. 818–823. DOI: 10.1109/9.286260 (cit. on p. 178).
- [132] Alessandro Giua and Frank DiCesare. "Decidability and closure properties of weak Petri net languages in supervisory control". In: *IEEE Transactions on Automatic Control* 40.5 (May 1995), pp. 906–910. DOI: 10.1109/9.384227 (cit. on p. 178).
- [133] Alessandro Giua and Frank DiCesare. "On the existence of Petri net supervisors". In: *[1992] Proceedings of the 31st IEEE Conference on Decision and Control*. 1992, pp. 3380–3385. DOI: 10.1109/CDC.1992.371011 (cit. on p. 178).
- [134] Alessandro Giua and Frank DiCesare. "Petri net structural analysis for supervisory control". In: *IEEE Transactions on Robotics and Automation* 10.2 (1994), pp. 185–195. DOI: 10.1109/70.282543 (cit. on p. 179).
- [135] Alessandro Giua and Frank DiCesare. "Supervisory design using Petri nets". In: *[1991] Proceedings of the 30th IEEE Conference on Decision and Control*. Dec. 1991, pp. 92–97. DOI: 10.1109/CDC.1991.261262 (cit. on pp. 177, 178).
- [136] Alessandro Giua and Frank DiCesare. "Weak Petri net languages in supervisory control". In: *Proceedings of 32nd IEEE Conference on Decision and Control*. Dec. 1993, pp. 229–234. DOI: 10.1109/CDC.1993.325158 (cit. on p. 178).
- [137] Alessandro Giua and Carla Seatzu. "Modeling and Supervisory Control of Railway Networks Using Petri Nets". In: *IEEE Transactions on Automation Science and Engineering* 5.3 (2008), pp. 431–445. DOI: 10.1109/TASE.2008.916925 (cit. on p. 179).
- [138] Alessandro Giua and Carla Seatzu. "Petri nets for the control of discrete event systems". In: *Software and System Modeling* 14.2 (2015), pp. 693–701. DOI: 10.1007/s10270-014-0425-1 (cit. on p. 179).
- [139] GNU. *time - time a simple command or give resource usage*. 2017. URL: <http://man7.org/linux/man-pages/man1/time.1.html> (cit. on p. 158).

- [140] C. H. Golaszewski and Peter Jeffrey Godwin Ramadge. “Mutual exclusion problems for discrete event systems with shared events”. In: *Proceedings of the 27th IEEE Conference on Decision and Control*. Dec. 1988, pp. 234–239. DOI: 10.1109/CDC.1988.194301 (cit. on p. 178).
- [141] Christopher Howard Griffin. “A note on deciding controllability in pushdown systems”. In: *IEEE Transactions on Automatic Control* 51.2 (2006), pp. 334–337. DOI: 10.1109/TAC.2005.863513 (cit. on pp. 165, 166, 181).
- [142] Christopher Howard Griffin. “A Note on the Properties of the Supremal Controllable Sublanguage in Pushdown Systems”. In: *IEEE Transactions on Automatic Control* 53.3 (2008), pp. 826–829. DOI: 10.1109/TAC.2008.919519 (cit. on p. 181).
- [143] Christopher Howard Griffin. “Decidability and Optimality in Pushdown Control Systems: A New Approach to Discrete Event Control”. PhD thesis. Penn State, 2007 (cit. on pp. 144–146, 150, 181, 215).
- [144] Christopher Howard Griffin. “Exception Handling Controllers: An application of pushdown systems to discrete event control”. In: *2008 American Control Conference*. June 2008, pp. 1722–1727. DOI: 10.1109/ACC.2008.4586740 (cit. on p. 181).
- [145] Christopher Howard Griffin. “On partial observability in discrete event control with pushdown systems”. In: *Proceedings of the 2010 American Control Conference*. June 2010, pp. 2619–2622. DOI: 10.1109/ACC.2010.5530531 (cit. on p. 181).
- [146] Christopher Howard Griffin. “On partial observability in discrete event control with pushdown systems”. In: *Proceedings of the 2010 American Control Conference*. June 2010, pp. 2619–2622. DOI: 10.1109/ACC.2010.5530531 (cit. on p. 181).
- [147] Irène Guessarian. “Pushdown Tree Automata”. In: *Mathematical Systems Theory* 16.4 (1983), pp. 237–263. DOI: 10.1007/BF01744582 (cit. on p. 216).
- [148] Michel Henri Théodore Hack. “Decidability questions for Petri nets”. PhD thesis. Massachusetts Institute of Technology. Dept. of Electrical Engineering and Computer Science, 1976 (cit. on p. 177).
- [149] Michel Henri Théodore Hack. “Petri Net Languages”. In: *Computation Structures Group - Memo No. 124* (June 1975) (cit. on p. 177).
- [150] Esfandiar Haghverdi and Hasan Ural. “Submodule construction from concurrent system specifications”. In: *Information & Software Technology* 41.8 (1999), pp. 499–506. DOI: 10.1016/S0950-5849(99)00014-2 (cit. on p. 171).
- [151] Chen Haoxun and Li Huifeng. “Maximally permissive state feedback logic for controlled time Petri nets”. In: *Proceedings of the 1997 American Control Conference (Cat. No.97CH36041)*. Vol. 4. June 1997, pp. 2359–2363. DOI: 10.1109/ACC.1997.609097 (cit. on p. 179).
- [152] David Harel. “Statecharts: A Visual Formalism for Complex Systems”. In: *Science of Computer Programming* 8.3 (1987), pp. 231–274. DOI: 10.1016/0167-6423(87)90035-9 (cit. on pp. 111, 164).
- [153] David Harel, Hillel Kugler, and Amir Pnueli. “Synthesis Revisited: Generating Statechart Models from Scenario-Based Requirements”. In: *Formal Methods in Software and Systems Modeling, Essays Dedicated to Hartmut Ehrig, on the Occasion of His 60th Birthday*. Ed. by Hans-Jörg Kreowski, Ugo Montanari, Fernando Orejas, Grzegorz Rozenberg, and Gabriele Taentzer. Vol. 3393. Lecture Notes in Computer Science. Springer, 2005, pp. 309–324. DOI: 10.1007/978-3-540-31847-7_18 (cit. on p. 167).
- [154] William R. Harris, Somesh Jha, and Thomas W. Reps. “Secure Programming via Visibly Pushdown Safety Games”. In: *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*. Ed. by Parthasarathy Madhusudan and Sanjit A. Seshia. Vol. 7358. Lecture Notes in Computer Science. Springer, 2012, pp. 581–598. DOI: 10.1007/978-3-642-31424-7_41 (cit. on p. 168).
- [155] William R. Harris, Somesh Jha, Thomas W. Reps, and Sanjit A. Seshia. “Program synthesis for interactive-security systems”. In: *Formal Methods in System Design* 51.2 (2017), pp. 362–394. DOI: 10.1007/s10703-017-0296-5 (cit. on p. 168).

- [156] Reiko Heckel and Mourad Chouikha. “Control Synthesis for discrete Event Systems: a Semantic Framework Based on Open Petri Nets”. In: *Transactions of the SDPS* 6.4 (2002), pp. 63–77. URL: <http://content.iospress.com/articles/journal-of-integrated-design-and-process-science/jid6-4-05> (cit. on p. 180).
- [157] Stephan Heilbrunner. “A Parsing Automata Approach to LR Theory”. In: *Theoretical Computer Science* 15 (1981), pp. 117–157. DOI: 10.1016/0304-3975(81)90067-0 (cit. on p. 157).
- [158] Steffen Helke and Florian Kammüller. *Formalizing Statecharts using Hierarchical Automata*. 2010. URL: <http://www.isa-afp.org/entries/Statecharts.shtml> (cit. on p. 187).
- [159] Joseph L. Hellerstein, Yixin Diao, Sujay S. Parekh, and Dawn M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004. ISBN: 047126637X (cit. on p. 170).
- [160] Kunihiro Hiraishi and Petr Kuvcera. “Application of DES Theory to Verification of Software Components”. In: *IEICE Transactions* 92-A.2 (2009), pp. 604–610. URL: http://search.ieice.org/bin/summary.php?id=e92-a_2_604&category=A&year=2009&lang=E&abst= (cit. on p. 180).
- [161] Charles Antony Richard Hoare. “An Axiomatic Basis for Computer Programming”. In: *Communications of the ACM* 12.10 (1969), pp. 576–580. DOI: 10.1145/363235.363259 (cit. on p. 48).
- [162] Charles Antony Richard Hoare. “Communicating Sequential Processes”. In: *Communications of the ACM* 21.8 (1978), pp. 666–677. DOI: 10.1145/359576.359585 (cit. on pp. 171, 209).
- [163] John Edward Hopcroft and Jeffrey David Ullman. *Introduction to Automata Theory, languages and computation*. Ed. by Michael A Harrison. Addison-Wesley Publishing company, 1979 (cit. on pp. 36, 64, 182, 183).
- [164] Lawrence E. Holloway and S. Chand. “Time templates for discrete event fault monitoring in manufacturing systems”. In: *American Control Conference, 1994*. Vol. 1. June 1994, pp. 701–706. DOI: 10.1109/ACC.1994.751830 (cit. on p. 173).
- [165] Lawrence E. Holloway and Bruce H. Krogh. “On closed-loop liveness of discrete-event systems under maximally permissive control”. In: *IEEE Transactions on Automatic Control* 37.5 (May 1992), pp. 692–697. DOI: 10.1109/9.135519 (cit. on p. 178).
- [166] Lawrence E. Holloway and Bruce H. Krogh. “Synthesis of feedback control logic for a class of controlled Petri nets”. In: *IEEE Transactions on Automatic Control* 35.5 (May 1990), pp. 514–523. DOI: 10.1109/9.53517 (cit. on p. 178).
- [167] Lawrence E. Holloway, Bruce H. Krogh, and Alessandro Giua. “A Survey of Petri Net Methods for Controlled Discrete Event Systems”. In: *Discrete Event Dynamic Systems* 7.2 (1997), pp. 151–190. DOI: 10.1023/A:1008271916548 (cit. on pp. 178, 179).
- [168] John Edward Hopcroft, Rajeev Motwani, and Jeffrey David Ullman. *Introduction to Automata Theory, Languages, and Computation*. 2nd ed. Addison-Wesley, 2001. ISBN: 0-201-44124-1 (cit. on p. 210).
- [169] David Albert Huffman. “A Method for the Construction of Minimum-Redundancy Codes”. In: 40 (Oct. 1952), pp. 1098–1101 (cit. on p. 214).
- [170] Atsunobu Ichikawa and Kunihiro Hiraishi. “Analysis and control of discrete event systems represented by petri nets”. In: *Discrete Event Systems: Models and Applications*. Ed. by Pravin Varaiya and Alexander B. Kurzhanski. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 115–134. ISBN: 978-3-540-48045-7 (cit. on p. 178).
- [171] Max Planck Institute for Informatics. *SPASS Workbench*. 2017. URL: <https://www.mpi-inf.mpg.de/departments/automation-of-logic/software/spass-workbench/> (cit. on p. 93).
- [172] Inria - Inventeurs du monde numérique. *The Coq Proof Assistant*. 2018. URL: <https://coq.inria.fr/> (cit. on p. 155).

- [173] Marian V. Iordache and Panos J. Antsaklis. "Concurrent program synthesis based on supervisory control". In: *Proceedings of the 2010 American Control Conference*. June 2010, pp. 3378–3383. DOI: 10.1109/ACC.2010.5530904 (cit. on pp. 168, 169, 179).
- [174] Marian V. Iordache and Panos J. Antsaklis. "Petri nets and programming: A survey". In: *2009 American Control Conference*. June 2009, pp. 4994–4999. DOI: 10.1109/ACC.2009.5159987 (cit. on pp. 169, 179).
- [175] Marian V. Iordache and Panos J. Antsaklis. "Supervision Based on Place Invariants: A Survey". In: *Discrete Event Dynamic Systems* 16.4 (2006), pp. 451–492. DOI: 10.1007/s10626-006-0021-9 (cit. on p. 179).
- [176] Marian V. Iordache and Panos J. Antsaklis. *Synthesis of Concurrent Programs Based on Supervisory Control*. Tech. rep. 2009 (cit. on pp. 168, 169, 179).
- [177] Marian V. Iordache, John O. Moody, and Panos J. Antsaklis. "Synthesis of deadlock prevention supervisors using Petri nets". In: *IEEE Transactions on Robotics and Automation* 18.1 (2002), pp. 59–68. DOI: 10.1109/70.988975 (cit. on p. 180).
- [178] Stefan Jacobi. "Controller Synthesis for Discrete Event Systems in the Setting of a Regular Plant and a Deterministic Context-Free Specification in libFAUDES". MA thesis. Technische Universität Berlin, Fachgebiet Regelungssysteme, 2013 (cit. on p. 156).
- [179] Matthias Jantzen. "Language Theory of Petri Nets". In: *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part I, Proceedings of an Advanced Course, Bad Honnef, 8.-19. September 1986*. Ed. by Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg. Vol. 254. Lecture Notes in Computer Science. Springer, 1986, pp. 397–412. DOI: 10.1007/BFb0046847 (cit. on p. 177).
- [180] MuDer Jeng and Frank DiCesare. "A review of synthesis techniques for Petri nets with applications to automated manufacturing systems". In: *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 23.1 (1993), pp. 301–312. DOI: 10.1109/21.214792 (cit. on p. 178).
- [181] Ting Jiao, Yongmei Gan, Guochun Xiao, and Walter Murray Wonham. "Exploiting symmetry of state tree structures for discrete-event systems with parallel components". In: *International Journal of Control* 90.8 (2017), pp. 1639–1651. DOI: 10.1080/00207179.2016.1216607 (cit. on pp. 175, 176).
- [182] Temesghen Kahsai and Marino Miculan. "Implementing Spi-Calculus Using Nominal Techniques". In: *Logic and Theory of Algorithms, 4th Conference on Computability in Europe, CiE 2008, Athens, Greece, June 15-20, 2008, Proceedings*. Ed. by Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe. Vol. 5028. Lecture Notes in Computer Science. Springer, 2008, pp. 294–305. ISBN: 978-3-540-69405-2. DOI: 10.1007/978-3-540-69407-6_33 (cit. on p. 187).
- [183] Timothy Kam, Tiziano Villa, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. *Synthesis of Finite State Machines: Functional Optimization*. Springer-Verlag US 1997, 1997. DOI: 10.1007/978-1-4757-2622-0 (cit. on p. 171).
- [184] Era Kasturia, Frank DiCesare, and Alan A. Desrochers. "Real time control of multilevel manufacturing systems using colored Petri nets". In: *Proceedings of the 1988 IEEE International Conference on Robotics and Automation, Philadelphia, Pennsylvania, USA, April 24-29, 1988*. IEEE, 1988, pp. 1114–1119. DOI: 10.1109/ROBOT.1988.12209 (cit. on p. 179).
- [185] Samir G. Kelekar and George W. Hart. "Synthesis of Protocols and Protocol Converters Using the Submodule Construction Approach". In: *Protocol Specification, Testing and Verification XIII, Proceedings of the IFIP TC6/WG6.1 Thirteenth International Symposium on Protocol Specification, Testing and Verification, Liège, Belgium, 25-28 May, 1993*. Ed. by André A. S. Danthine, Guy Leduc, and Pierre Wolper. Vol. C-16. IFIP Transactions. North-Holland, 1993, pp. 307–322. ISBN: 0-444-81648-8 (cit. on p. 171).
- [186] Jeffrey O. Kephart and David M. Chess. "The Vision of Autonomic Computing". In: *IEEE Computer* 36.1 (2003), pp. 41–50. DOI: 10.1109/MC.2003.1160055 (cit. on p. 169).

- [187] Narges Khakpour, Farhad Arbab, and Éric Rutten. "Supervisory Controller Synthesis for Safe Software Adaptation". In: *12th International Workshop on Discrete Event Systems, WODES 2014, Cachan, France, May 14-16, 2014*. Ed. by Jean-Jacques Lesage, Jean-Marc Faure, José E. R. Cury, and Bengt Lennartson. International Federation of Automatic Control, 2014, pp. 39–45. ISBN: 978-3-902823-61-8. DOI: 10.3182/20140514-3-FR-4046.00035 (cit. on p. 169).
- [188] Eric Klavins. "A computation and control language for multi-vehicle systems". In: *42nd IEEE International Conference on Decision and Control (IEEE Cat. No.03CH37475)*. Vol. 4. Dec. 2003, pp. 4133–4139. DOI: 10.1109/CDC.2003.1271797 (cit. on p. 166).
- [189] Donald Ervin Knuth. "On the translation of languages from left to right". In: *Information and Control* 8.6 (1965), pp. 607–639. DOI: 10.1016/S0019-9958(65)90426-2 (cit. on pp. 18, 22, 27, 66, 67, 69–73, 76, 116, 121, 214).
- [190] Koichi Kobayashi and Kunihiro Hiraishi. "On opacity and diagnosability in discrete event systems modeled by pushdown automata". In: *2012 IEEE International Conference on Automation Science and Engineering, CASE 2012, Seoul, Korea (South), August 20-24, 2012*. IEEE, 2012, pp. 662–667. ISBN: 978-1-4673-0429-0. DOI: 10.1109/CoASE.2012.6386310 (cit. on p. 180).
- [191] Koichi Kobayashi and Kunihiro Hiraishi. "Verification of Opacity and Diagnosability for Pushdown Systems". In: *Journal of Applied Mathematics* 2013 (2013), pp. 1–10. DOI: 10.1155/2013/654059 (cit. on p. 180).
- [192] Jan Komenda, Tomás Masopust, and Jan Hendrik van Schuppen. "Distributed computation of maximally permissive supervisors in three-level relaxed coordination control of discrete-event systems". In: *55th IEEE Conference on Decision and Control, CDC 2016, Las Vegas, NV, USA, December 12-14, 2016*. IEEE, 2016, pp. 441–446. ISBN: 978-1-5090-1837-6. DOI: 10.1109/CDC.2016.7798308 (cit. on p. 175).
- [193] A. J. Korenjak. "A practical method for constructing LR(k) processors". In: *Communications of the ACM* 12.11 (1969), pp. 613–623. DOI: 10.1145/363269.363281 (cit. on p. 214).
- [194] Jeff Kramer and Jeff Magee. "Self-Managed Systems: an Architectural Challenge". In: *International Conference on Software Engineering, ISCE 2007, Workshop on the Future of Software Engineering, FOSE 2007, May 23-25, 2007, Minneapolis, MN, USA*. Ed. by Lionel C. Briand and Alexander L. Wolf. IEEE Computer Society, 2007, pp. 259–268. ISBN: 0-7695-2829-5. DOI: 10.1109/FOSE.2007.19 (cit. on p. 170).
- [195] Alexander Krauss and Tobias Nipkow. *Regular Sets and Expressions*. 2010. URL: <http://www.isa-afp.org/entries/Regular-Sets.shtml> (cit. on p. 186).
- [196] Stephan Kreutzer and Martin Lange. "Non-regular fixed-point logics and games". In: *Logic and Automata: History and Perspectives [in Honor of Wolfgang Thomas]*. Ed. by Jörg Flum, Erich Grädel, and Thomas Wilke. Vol. 2. Texts in Logic and Games. Amsterdam University Press, 2008, pp. 423–456. ISBN: 978-90-5356-576-6 (cit. on pp. 166, 168, 169).
- [197] Bruce H. Krogh. "Controlled Petri nets and maximally permissive feedback logic". In: (Sept. 1987), pp. 317–326 (cit. on p. 178).
- [198] Bruce H. Krogh and C. L. Beck. "Synthesis of Place Transition Nets for Simulation and Control of Manufacturing Systems". In: *IFAC Proceedings Volumes* 20.9 (1987). 4th IFAC/IFORS Symposium on Large Scale Systems: Theory and Applications 1986, Zurich, Switzerland, 26-29 August 1986, pp. 583–588. DOI: 10.1016/S1474-6670(17)55770-5 (cit. on p. 178).
- [199] Bruce H. Krogh and Lawrence E. Holloway. "Synthesis of feedback control logic for discrete manufacturing systems". In: *Automatica* 27.4 (1991), pp. 641–651. DOI: 10.1016/0005-1098(91)90055-7 (cit. on p. 178).
- [200] Ratnesh Kumar and Vijay Garg. "Optimal Supervisory Control of Discrete Event Dynamical Systems". In: *SIAM Journal on Control and Optimization (SICON)* 33.2 (1995), pp. 419–439. DOI: 10.1137/S0363012992235183 (cit. on p. 165).

- [201] Ratnesh Kumar, Vijay Garg, and Steven I. Marcus. "On controllability and normality of discrete event dynamical systems". In: *Systems & Control Letters* 17.3 (1991), pp. 157–168. doi: 10.1016/0167-6911(91)90061-I (cit. on pp. 59, 165).
- [202] Ratnesh Kumar, Vijay Garg, and Steven I. Marcus. "On supervisory control of sequential behaviors". In: *IEEE Transactions on Automatic Control* 37.12 (Dec. 1992), pp. 1978–1985. doi: 10.1109/9.182487 (cit. on p. 168).
- [203] Ratnesh Kumar and Lawrence E. Holloway. "Supervisory control of deterministic Petri nets with regular specification languages". In: *IEEE Transactions on Automatic Control* 41.2 (Feb. 1996), pp. 245–249. doi: 10.1109/9.481527 (cit. on p. 180).
- [204] Ratnesh Kumar and Lawrence E. Holloway. "Supervisory control of Petri net languages". In: *[1992] Proceedings of the 31st IEEE Conference on Decision and Control*. 1992, pp. 1190–1195. doi: 10.1109/CDC.1992.371529 (cit. on p. 178).
- [205] Dara Kusic, Jeffrey O. Kephart, James E. Hanson, Nagarajan Kandasamy, and Guofei Jiang. "Power and performance management of virtualized computing environments via lookahead control". In: *Cluster Computing* 12.1 (2009), pp. 1–15. doi: 10.1007/s10586-008-0070-y (cit. on p. 170).
- [206] Marta Zofia Kwiatkowska, Gethin Norman, Roberto Segala, and Jeremy Sproston. "Verifying Quantitative Properties of Continuous Probabilistic Timed Automata". In: *CONCUR 2000 - Concurrency Theory, 11th International Conference, University Park, PA, USA, August 22-25, 2000, Proceedings*. Ed. by Catuscia Palamidessi. Vol. 1877. Lecture Notes in Computer Science. Springer, 2000, pp. 123–137. doi: 10.1007/3-540-44618-4_11 (cit. on p. 216).
- [207] Marta Zofia Kwiatkowska, Gethin Norman, Jeremy Sproston, and Fuzhi Wang. "Symbolic model checking for probabilistic timed automata". In: *Information and Computation* 205.7 (2007), pp. 1027–1077. doi: 10.1016/j.ic.2007.01.004 (cit. on p. 217).
- [208] Stéphane Lafortune, Demosthenis Teneketzis, Meera Sampath, Raja Sengupta, and Kasim Sinnamohideen. "Failure diagnosis of dynamic systems: an approach based on discrete event systems". In: *Proceedings of the 2001 American Control Conference*. (Cat. No.01CH37148). Vol. 3. 2001, pp. 2058–2071. doi: 10.1109/ACC.2001.946047 (cit. on p. 173).
- [209] François Laroussinie. HCMC. 2005. URL: <http://www.control.toronto.edu/~wonham/Research.html> (cit. on p. 172).
- [210] François Laroussinie and Kim Guldstrand Larsen. "Compositional Model Checking of Real Time Systems". In: *CONCUR '95: Concurrency Theory, 6th International Conference, Philadelphia, PA, USA, August 21-24, 1995, Proceedings*. Ed. by Insup Lee and Scott A. Smolka. Vol. 962. Lecture Notes in Computer Science. Springer, 1995, pp. 27–41. doi: 10.1007/3-540-60218-6_3 (cit. on pp. 171, 174, 175).
- [211] Kim Guldstrand Larsen. "Context-dependent Bisimulation Between Processes". PhD thesis. Institute of Electronic Systems, Aalborg University Centre, Denmark, 1986 (cit. on p. 171).
- [212] Kim Guldstrand Larsen. "Proof Systems for Satisfiability in Hennessy-Milner Logic with Recursion". In: *Theoretical Computer Science* 72.2&3 (1990), pp. 265–288. doi: 10.1016/0304-3975(90)90038-J (cit. on p. 171).
- [213] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. "Compositional and Symbolic Model-Checking of Real-Time Systems". In: *16th IEEE Real-Time Systems Symposium, Palazzo dei Congressi, Via Matteotti, 1, Pisa, Italy, December 4-7, 1995, Proceedings*. IEEE Computer Society, 1995, pp. 76–87. ISBN: 0-8186-7337-0. doi: 10.1109/REAL.1995.495198 (cit. on pp. 171, 174, 175).
- [214] André Bittencourt Leal, Diogo L. L. da Cruz, and Marcelo da Silva Hounsell. "Supervisory Control Implementation into Programmable Logic Controllers". In: *Proceedings of 12th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2009, September 22-25, 2008, Palma de Mallorca, Spain*. IEEE, 2009, pp. 1–7. ISBN: 978-1-4244-2727-7. doi: 10.1109/ETFA.2009.5347090 (cit. on p. 172).

- [215] Rogério de Lemos, David Garlan, Carlo Ghezzi, and Holger Giese, eds. *Software Engineering for Self-Adaptive Systems III. Assurances - International Seminar, Dagstuhl Castle, Germany, December 15-19, 2013, Revised Selected and Invited Papers*. Vol. 9640. Lecture Notes in Computer Science. Springer, 2017. DOI: 10.1007/978-3-319-74183-3 (cit. on p. 169).
- [216] Rogério de Lemos, Holger Giese, Hausi A. Müller, and Mary Shaw, eds. *Software Engineering for Self-Adaptive Systems II - International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*. Vol. 7475. Lecture Notes in Computer Science. Springer, 2013. DOI: 10.1007/978-3-642-35813-5 (cit. on p. 169).
- [217] Hing Leung and Detlef Wotschke. "On the size of parsers and LR(k)-grammars". In: *Theoretical Computer Science* 242.1-2 (2000), pp. 59–69. DOI: 10.1016/S0304-3975(98)00199-6 (cit. on p. 157).
- [218] Jean Pierre Lévy. "Automatic Correction of Syntax Errors in Programming Languages". PhD thesis. Cornell University, Dec. 1971 (cit. on p. 76).
- [219] Yang Li and Walter Murray Wonham. "On Supervisory control of real-time discrete-event systems". In: *Information Sciences* 46.3 (1988), pp. 159–183. DOI: 10.1016/0020-0255(88)90048-5 (cit. on pp. 173, 175, 217).
- [220] Feng Lin. "Diagnosability of discrete event systems and its applications". In: *Discrete Event Dynamic Systems* 4.2 (1994), pp. 197–212. DOI: 10.1007/BF01441211 (cit. on p. 173).
- [221] Feng Lin and Walter Murray Wonham. "Decentralized supervisory control of discrete-event systems". In: *Information Sciences* 44.3 (1988), pp. 199–224. DOI: 10.1016/0020-0255(88)90002-3 (cit. on p. 175).
- [222] Feng Lin and Walter Murray Wonham. "On observability of discrete-event systems". In: *Information Sciences* 44.3 (1988), pp. 173–198. DOI: 10.1016/0020-0255(88)90001-1 (cit. on pp. 173, 216).
- [223] Feng Lin and Walter Murray Wonham. "Supervisory control of timed discrete-event systems under partial observation". In: *IEEE Transactions on Automatic Control* 40.3 (Mar. 1995), pp. 558–562. DOI: 10.1109/9.376081 (cit. on pp. 173, 174).
- [224] Marin Litoiu, Mary Shaw, Gabriel Tamura, Norha M. Villegas, Hausi A. Müller, Holger Giese, Romain Rouvoy, and Éric Rutten. "What Can Control Theory Teach Us About Assurances in Self-Adaptive Software Systems?" In: *Software Engineering for Self-Adaptive Systems III. Assurances - International Seminar, Dagstuhl Castle, Germany, December 15-19, 2013, Revised Selected and Invited Papers*. Ed. by Rogério de Lemos, David Garlan, Carlo Ghezzi, and Holger Giese. Vol. 9640. Lecture Notes in Computer Science. Springer, 2013, pp. 90–134. DOI: 10.1007/978-3-319-74183-3_4 (cit. on pp. 169, 170).
- [225] Cong Liu, Alex Kondratyev, Yosinori Watanabe, Jörg Desel, and Alberto L. Sangiovanni-Vincentelli. "Schedulability Analysis of Petri Nets Based on Structural Properties". In: *Fundamenta Informaticae* 86.3 (2008), pp. 325–341. URL: <http://content.iospress.com/articles/fundamenta-informaticae/fi86-3-05> (cit. on p. 169).
- [226] Christof Löding, Parthasarathy Madhusudan, and Olivier Serre. "Visibly Pushdown Games". In: *FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science, 24th International Conference, Chennai, India, December 16-18, 2004, Proceedings*. Ed. by Kamal Lodaya and Meena Mahajan. Vol. 3328. Lecture Notes in Computer Science. Springer, 2004, pp. 408–420. DOI: 10.1007/978-3-540-30538-5_34 (cit. on p. 216).
- [227] Nancy Ann Lynch and Mark R. Tuttle. "Hierarchical Correctness Proofs for Distributed Algorithms". In: *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, Vancouver, British Columbia, Canada, August 10-12, 1987*. Ed. by Fred B. Schneider. ACM, 1987, pp. 137–151. ISBN: 0-89791-239-X. DOI: 10.1145/41840.41852 (cit. on pp. 33, 111, 164).
- [228] Chuan Ma and Walter Murray Wonham. "Nonblocking supervisory control of state tree structures". In: *IEEE Transactions on Automatic Control* 51.5 (2006), pp. 782–793. DOI: 10.1109/TAC.2006.875030 (cit. on pp. 164, 175, 176, 209).

- [229] Chuan Ma and Walter Murray Wonham. *STSLib*. 2014. URL: <https://github.com/chuanma/STSLib> (cit. on p. 172).
- [230] Ziyue Ma, Zhi Wu Li, and Alessandro Giua. “Characterization of Admissible Marking Sets in Petri Nets With Conflicts and Synchronizations”. In: *IEEE Transactions on Automatic Control* 62.3 (2017), pp. 1329–1341. DOI: 10.1109/TAC.2016.2585647 (cit. on p. 179).
- [231] Ziyue Ma, Zhi Wu Li, and Alessandro Giua. “Design of Optimal Petri Net Controllers for Disjunctive Generalized Mutual Exclusion Constraints”. In: *IEEE Transactions on Automatic Control* 60.7 (2015), pp. 1774–1785. DOI: 10.1109/TAC.2015.2389313 (cit. on p. 179).
- [232] Martina Maggio, Enrico Bini, Georgios C. Chasparis, and Karl-Erik Årzén. “A Game-Theoretic Resource Manager for RT Applications”. In: *25th Euromicro Conference on Real-Time Systems, ECRTS 2013, Paris, France, July 9-12, 2013*. IEEE Computer Society, 2013, pp. 57–66. ISBN: 978-0-7695-5054-1. DOI: 10.1109/ECRTS.2013.17 (cit. on p. 168).
- [233] Rupak Majumdar, Elaine Render, and Paulo Tabuada. “A theory of robust ω -regular software synthesis”. In: *ACM Transactions on Embedded Computing Systems (TECS)* 13.3 (2013), pp. 1–27. DOI: 10.1145/2539036.2539044 (cit. on p. 166).
- [234] Mbi Makungu, Michel Barbeau, and Richard St-Denis. “Synthesis of Controllers of Processes Modeled as Colored Petri Nets”. In: *Discrete Event Dynamic Systems* 9.2 (1999), pp. 147–169. DOI: 10.1023/A:1008371814442 (cit. on p. 180).
- [235] Oded Maler, Amir Pnueli, and Joseph Sifakis. “On the Synthesis of Discrete Controllers for Timed Systems (An Extended Abstract)”. In: *STACS*. 1995, pp. 229–242. DOI: 10.1007/3-540-59042-0_76 (cit. on pp. 171, 174).
- [236] Kaushik Mallik and Anne-Kathrin Schmuck. “Supervisory controller synthesis for decomposable deterministic context free specification languages”. In: *13th International Workshop on Discrete Event Systems, WODES 2016, Xi’an, China, May 30 - June 1, 2016*. Ed. by Christos G. Cassandras, Alessandro Giua, and Zhiwu Li. IEEE, 2016, pp. 22–27. ISBN: 978-1-5090-4190-9. DOI: 10.1109/WODES.2016.7497821 (cit. on p. 216).
- [237] Zohar Manna. “Knowledge and Reasoning in Program Synthesis”. In: *Programming Methodology, 4th Informatik Symposium, IBM Germany, Wildbad, September 25-27, 1974*. Ed. by Clemens Hackl. Vol. 23. Lecture Notes in Computer Science. Springer, 1974, pp. 236–277. ISBN: 3-540-07131-8. DOI: 10.1007/3-540-07131-8_29. URL: https://doi.org/10.1007/3-540-07131-8_29 (cit. on p. 1).
- [238] Zohar Manna and Richard Jay Waldinger. “A Deductive Approach to Program Synthesis”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 2.1 (1980), pp. 90–121. DOI: 10.1145/357084.357090 (cit. on p. 167).
- [239] Zohar Manna and Pierre Wolper. “Synthesis of Communicating Processes from Temporal Logic Specifications”. In: *Logics of Programs, Workshop, Yorktown Heights, New York, May 1981*. Ed. by Dexter Kozen. Vol. 131. Lecture Notes in Computer Science. Springer, 1981, pp. 253–281. DOI: 10.1007/BFb0025786 (cit. on p. 171).
- [240] Hervé Marchand, Patricia Bournai, Michel Le Borgne, and Paul Le Guernic. “Synthesis of Discrete-Event Controllers Based on the Signal Environment”. In: *Discrete Event Dynamic Systems* 10.4 (2000), pp. 325–346. DOI: 10.1023/A:1008311720696 (cit. on p. 167).
- [241] Hervé Marchand, Patricia Bournai, Michel Le Borgne, Paul Le Guernic, M. Samaan, and Éric Rutten. *Sigali*. 2000. URL: <http://www.irisa.fr/vertecs/Logiciels/sigali.html> (cit. on p. 172).
- [242] Jasen Markovski, Dirk A. van Beek, and Jos C. M. Baeten. “Partially-Supervised Plants: Embedding Control Requirements in Plant Components”. In: *Integrated Formal Methods - 9th International Conference, IFM 2012, Pisa, Italy, June 18-21, 2012. Proceedings*. Ed. by John Derrick, Stefania Gnesi, Diego Latella, and Helen Treharne. Vol. 7321. Lecture Notes in Computer Science. Springer, 2012, pp. 253–267. DOI: 10.1007/978-3-642-30729-4_18 (cit. on p. 164).

- [243] Tomás Masopust. “A note on controllability of deterministic context-free systems”. In: *Automatica* 48.8 (2012), pp. 1934–1937. DOI: 10.1016/j.automatica.2012.06.004 (cit. on p. 180).
- [244] Robert McNaughton. *Research in the Theory of Automata*. Tech. rep. Project MAC Report. MIT, 1965, pp. 58–63 (cit. on p. 1).
- [245] Philip M. Merlin and Gregor von Bochmann. “On the Construction of Submodule Specifications and Communication Protocols”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 5.1 (1983), pp. 1–25. DOI: 10.1145/357195.357196 (cit. on pp. 1, 146, 171).
- [246] Robin Milner. *A Calculus of Communicating Systems*. Vol. 92. Lecture Notes in Computer Science. Springer, 1980. DOI: 10.1007/3-540-10235-3 (cit. on p. 164).
- [247] Thomas Moor and Jennifer M. Davoren. “Robust Controller Synthesis for Hybrid Systems Using Modal Logic”. In: *Hybrid Systems: Computation and Control, 4th International Workshop, HSCC 2001, Rome, Italy, March 28-30, 2001, Proceedings*. Ed. by Maria Domenica Di Benedetto and Alberto L. Sangiovanni-Vincentelli. Vol. 2034. Lecture Notes in Computer Science. Springer, 2001, pp. 433–446. DOI: 10.1007/3-540-45351-2_35 (cit. on p. 176).
- [248] Thomas Moor, Jörg Raisch, and Siu O’Young. “Discrete Supervisory Control of Hybrid Systems Based on l-Complete Approximations”. In: *Discrete Event Dynamic Systems* 12.1 (2002), pp. 83–107. DOI: 10.1023/A:1013339920783 (cit. on p. 176).
- [249] Thomas Moor, Klaus Werner Schmidt, and Thomas Wittmann. “Abstraction-Based Control for Not Necessarily Closed Behaviours”. In: *IFAC Proceedings Volumes* 44.1 (2011). 18th IFAC World Congress, pp. 6988–6993. DOI: 10.3182/20110828-6-IT-1002.00480 (cit. on p. 168).
- [250] Soraia Moradi, Laurent Hardouin, and Jörg Raisch. “Optimal control of a class of timed discrete event systems with shared resources, an approach based on the hadamard product of series in dioids”. In: *56th IEEE Annual Conference on Decision and Control, CDC 2017, Melbourne, Australia, December 12-15, 2017*. IEEE, 2017, pp. 4831–4838. ISBN: 978-1-5090-2873-3. DOI: 10.1109/CDC.2017.8264373 (cit. on p. 179).
- [251] Christophe Morvan and Sophie Pinchinat. “Diagnosability of Pushdown Systems”. In: *Hardware and Software: Verification and Testing - 5th International Haifa Verification Conference, HVC 2009, Haifa, Israel, October 19-22, 2009, Revised Selected Papers*. Ed. by Kedar S. Namjoshi, Andreas Zeller, and Avi Ziv. Vol. 6405. Lecture Notes in Computer Science. Springer, 2009, pp. 21–33. DOI: 10.1007/978-3-642-19237-1_6 (cit. on p. 180).
- [252] Markus Müller-Olm. “A Modal Fixpoint Logic with Chop”. In: *STACS 99, 16th Annual Symposium on Theoretical Aspects of Computer Science, Trier, Germany, March 4-6, 1999, Proceedings*. Ed. by Christoph Meinel and Sophie Tison. Vol. 1563. Lecture Notes in Computer Science. Springer, 1999, pp. 510–520. DOI: 10.1007/3-540-49116-3_48 (cit. on p. 168).
- [253] Tadao Murata, N. Komoda, K. Matsumoto, and K. Haruna. “A Petri Net-Based Controller for Flexible and Maintainable Sequence Control and its Applications in Factory Automation”. In: *IEEE Transactions on Industrial Electronics* IE-33.1 (Feb. 1986), pp. 1–8. DOI: 10.1109/TIE.1986.351700 (cit. on p. 178).
- [254] Petter Nilsson, Omar Hussien, Ayca Balkan, Yuxiao Chen, Aaron D. Ames, Jessy W. Grizzle, Necmiye Ozay, Huei Peng, and Paulo Tabuada. “Correct-by-Construction Adaptive Cruise Control: Two Approaches”. In: *IEEE Transactions on Industrial Electronics* 24.4 (2016), pp. 1294–1307. DOI: 10.1109/TCST.2015.2501351 (cit. on p. 170).
- [255] Tobias Nipkow. *Functional Automata*. 2004. URL: <http://www.isa-afp.org/entries/Functional-Automata.shtml> (cit. on p. 187).
- [256] Cüneyt M. Özveren and Alan S. Willsky. “Observability of discrete event dynamic systems”. In: *IEEE Transactions on Automatic Control* 35.7 (July 1990), pp. 797–806. DOI: 10.1109/9.57018 (cit. on pp. 173, 216).

- [257] Andrea Paoli and Stéphane Lafortune. “Safe diagnosability for fault-tolerant supervision of discrete-event systems”. In: *Automatica* 41.8 (2005), pp. 1335–1347. DOI: 10.1016/j.automatica.2005.03.017 (cit. on p. 173).
- [258] Seong-Jin Park and Kwang-Hyun Cho. “Modular nonblocking state feedback control of discrete event systems and its application to dynamic oligopolistic markets”. In: *International Journal of Control* 84.12 (2011), pp. 2046–2057. DOI: 10.1080/00207179.2011.633231 (cit. on p. 175).
- [259] Seong-Jin Park and Jong-Tae Lim. “Robust and fault-tolerant supervisory control of discrete event systems with partial observation and model uncertainty”. In: *International Journal of Systems Science* 29.9 (1998), pp. 953–957. DOI: 10.1080/00207729808929587 (cit. on p. 166).
- [260] Seong-Jin Park and Jong-Tae Lim. “Robust and nonblocking supervisory control of nondeterministic discrete event systems using trajectory models”. In: *IEEE Transactions on Automatic Control* 47.4 (2002), pp. 655–658. DOI: 10.1109/9.995044 (cit. on p. 166).
- [261] Joachim Parrow. “Submodule Construction as Equation Solving in CCS”. In: *Theoretical Computer Science* 68.2 (1989), pp. 175–202. DOI: 10.1016/0304-3975(89)90128-X (cit. on p. 171).
- [262] K. M. Passino and Panos J. Antsaklis. “Near-Optimal Control of Discrete Event Systems”. In: (Sept. 1989), pp. 915–924 (cit. on p. 165).
- [263] K. M. Passino and Panos J. Antsaklis. “On the optimal control of discrete event systems”. In: *Proceedings of the 28th IEEE Conference on Decision and Control*, Dec. 1989, pp. 2713–2718. DOI: 10.1109/CDC.1989.70672 (cit. on p. 165).
- [264] Lawrence Charles Paulson. *Finite Automata in Hereditarily Finite Set Theory*. 2015. URL: http://www.isa-afp.org/entries/Finite_Automata_HF.shtml (cit. on p. 187).
- [265] Lawrence Charles Paulson and Tobias Nipkow. *Isabelle/HOL*. 2017. URL: <http://isabelle.in.tum.de> (cit. on pp. 91, 93, 154, 227).
- [266] Lawrence Charles Paulson and Tobias Nipkow. *Isabelle/HOL: Archive of Formal Proofs*. 2017. URL: <http://www.isa-afp.org/topics.shtml> (cit. on pp. 186, 187).
- [267] Elisabeth Pelz. “Infinitary languages of Petri nets and logical sentences”. In: *Advances in Petri Nets 1987, covers the 7th European Workshop on Applications and Theory of Petri Nets, Oxford, UK, June 1986*. Ed. by Grzegorz Rozenberg. Vol. 266. Lecture Notes in Computer Science. Springer, 1986, pp. 224–237. DOI: 10.1007/3-540-18086-9_28 (cit. on p. 177).
- [268] Hans-Jörg Peter and Bernd Finkbeiner. “The Complexity of Bounded Synthesis for Timed Control with Partial Observability”. In: *Formal Modeling and Analysis of Timed Systems - 10th International Conference, FORMATS 2012, London, UK, September 18-20, 2012. Proceedings*. Ed. by Marcin Jurdzinski and Dejan Nickovic. Vol. 7595. Lecture Notes in Computer Science. Springer, 2012, pp. 204–219. DOI: 10.1007/978-3-642-33365-1_15 (cit. on pp. 164, 173).
- [269] James Lyle Peterson. *Petri Net Theory and the Modeling of Systems*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1981. ISBN: 0136619835 (cit. on p. 177).
- [270] Alexandre Petrenko and Nina Yevtushenko. “Solving Asynchronous Equations”. In: *Formal Description Techniques and Protocol Specification, Testing and Verification, FORTE XI / PSTV XVIII’98, IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XI) and Protocol Specification, Testing and Verification (PSTV XVIII), 3-6 November, 1998, Paris, France*. Ed. by Stanislaw Budkowski, Ana R. Cavalli, and Elie Najm. Vol. 135. IFIP Conference Proceedings. Kluwer, 1998, pp. 231–247. ISBN: 0-412-84760-4 (cit. on p. 171).
- [271] Alexandre Petrenko, Nina Yevtushenko, Gregor von Bochmann, and Rachida Dssouli. “Testing in context: framework and test derivation”. In: *Computer Communications* 19.14 (1996), pp. 1236–1249. DOI: 10.1016/S0140-3664(96)01157-7 (cit. on p. 171).
- [272] Carl Adam Petri. “Kommunikation mit Automaten”. PhD thesis. Universität Hamburg, 1962 (cit. on pp. 111, 164, 176).

- [273] Amir Pnueli and Roni Rosner. “Distributed Reactive Systems Are Hard to Synthesize”. In: *31st Annual Symposium on Foundations of Computer Science, St. Louis, Missouri, USA, October 22-24, 1990, Volume II*. IEEE Computer Society, 1990, pp. 746–757. DOI: 10.1109/FSCS.1990.89597. URL: <https://doi.org/10.1109/FSCS.1990.89597> (cit. on p. 9).
- [274] Amir Pnueli and Roni Rosner. “On the Synthesis of a Reactive Module”. In: *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*. ACM Press, 1989, pp. 179–190. ISBN: 0-89791-294-2. DOI: 10.1145/75277.75293 (cit. on p. 168).
- [275] Huajun Qin and Philip Lewis. “Factorisation of Finite State Machines under Strong and Observational Equivalences”. In: *Formal Aspects of Computing* 3.3 (1991), pp. 284–307. DOI: 10.1007/BF01245634 (cit. on p. 171).
- [276] Huajun Qin and Philip Lewis. “Factorization of Finite State Machines under Observational Equivalence”. In: *CONCUR ’90, Theories of Concurrency: Unification and Extension, Amsterdam, The Netherlands, August 27-30, 1990, Proceedings*. Ed. by Jos C. M. Baeten and Jan Willem Klop. Vol. 458. Lecture Notes in Computer Science. Springer, 1990, pp. 427–441. DOI: 10.1007/BFb0039075 (cit. on p. 171).
- [277] Michael Oser Rabin. “Probabilistic Automata”. In: *Information and Control* 6.3 (1963), pp. 230–245. DOI: 10.1016/S0019-9958(63)90290-0 (cit. on p. 111).
- [278] Jörg Raisch and Thomas Moor. “Hierarchical Hybrid Control Synthesis and its Application to a Multiproduct Batch Plant”. In: *Control and Observer Design for Nonlinear Finite and Infinite Dimensional Systems*. Ed. by Thomas Meurer, Knut Graichen, and Ernst Dieter Gilles. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 199–216. DOI: 10.1007/11529798_13 (cit. on p. 176).
- [279] Peter Jeffrey Godwin Ramadge. “Observability of discrete event systems”. In: *1986 25th IEEE Conference on Decision and Control*. Dec. 1986, pp. 1108–1112. DOI: 10.1109/CDC.1986.267551 (cit. on pp. 173, 216).
- [280] Peter Jeffrey Godwin Ramadge and Walter Murray Wonham. “On the Supremal Controllable Sublanguage of a Given Language”. In: *SIAM Journal on Control and Optimization (SICON)* 25.3 (1987), pp. 637–659 (cit. on p. 58).
- [281] Peter Jeffrey Godwin Ramadge and Walter Murray Wonham. “On the Supremal Controllable Sublanguage of a given Language”. In: *Decision and Control, 1984. The 23rd IEEE Conference on*. Vol. 23. 1984, pp. 1073–1080. DOI: 10.1109/CDC.1984.272178 (cit. on pp. 31, 34, 45, 49, 56, 57, 171, 172, 179, 213, 220, 224).
- [282] Peter Jeffrey Godwin Ramadge and Walter Murray Wonham. “Supervisory Control of a Class of Discrete Event Processes”. English. In: *Analysis and Optimization of Systems*. Ed. by A. Bensoussan and J.L. Lions. Vol. 63. Lecture Notes in Control and Information Sciences. Springer Berlin Heidelberg, 1984, pp. 475–498. DOI: 10.1007/BFb0006306 (cit. on pp. 1, 2, 4, 5, 8, 9, 18, 31, 166, 167, 170, 172, 178, 219, 224).
- [283] Peter Jeffrey Godwin Ramadge and Walter Murray Wonham. “Supervisory Control of a Class of Discrete Event Processes”. In: *SIAM Journal on Control and Optimization (SICON)* 25.1 (Jan. 1987), pp. 206–230. DOI: 10.1137/0325013 (cit. on pp. 165, 180).
- [284] C. V. Ramamoorthy, Siyi Terry Dong, and Yutaka Usuda. “An Implementation of an Automated Protocol Synthesizer (APS) and Its Application to the X.21 Protocol”. In: *IEEE Transactions on Software Engineering* 11.9 (1985), pp. 886–908. DOI: 10.1109/TSE.1985.232547 (cit. on p. 171).
- [285] Stefan Rass, Bo An, Christopher Kiekintveld, Fei Fang, and Stefan Schauer, eds. *Decision and Game Theory for Security - 8th International Conference, GameSec 2017, Vienna, Austria, October 23-25, 2017, Proceedings*. Vol. 10575. Lecture Notes in Computer Science. Springer, 2017. DOI: 10.1007/978-3-319-68711-7 (cit. on p. 168).
- [286] Thomas Reinbacher, Matthias Függer, and Jörg Brauer. “Runtime verification of embedded real-time systems”. In: *Formal Methods in System Design* 44.3 (2014), pp. 203–239. DOI: 10.1007/s10703-013-0199-z (cit. on p. 173).

- [287] Christine Röckl. “A First-Order Syntax for the π -Calculus in Isabelle/HOL using Permutations”. In: *Electronic Notes in Theoretical Computer Science* 58.1 (2001), pp. 1–17. DOI: 10.1016/S1571-0661(04)00276-2 (cit. on p. 187).
- [288] Christine Röckl and Daniel Hirschhoff. “A fully adequate shallow embedding of the π -Calculus in Isabelle/HOL with mechanized syntax analysis”. In: *Journal of Functional Programming* 13.2 (2003), pp. 415–451. DOI: 10.1017/S0956796802004653 (cit. on p. 187).
- [289] Christine Röckl, Daniel Hirschhoff, and Stefan Berghofer. “Higher-Order Abstract Syntax with Induction in Isabelle/HOL: Formalizing the π -Calculus and Mechanizing the Theory of Contexts”. In: *Foundations of Software Science and Computation Structures, 4th International Conference, FOSSACS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*. Ed. by Furio Honsell and Marino Miculan. Vol. 2030. Lecture Notes in Computer Science. Springer, 2001, pp. 364–378. ISBN: 3-540-41864-4. DOI: 10.1007/3-540-45315-6_24 (cit. on p. 187).
- [290] Roni Rosner. “Modular Synthesis of Reactive Systems”. PhD thesis. Rehovot, Israel: The Weizmann Institute of Science, 1992 (cit. on p. 168).
- [291] Yu Ru, Maria Paola Cabasino, Alessandro Giua, and Christoforos N. Hadjicostis. “Supervisor synthesis for discrete event systems under partial observation and arbitrary forbidden state specifications”. In: *Discrete Event Dynamic Systems* 24.3 (2014), pp. 275–307. DOI: 10.1007/s10626-012-0152-0 (cit. on p. 179).
- [292] Matthias Rungger and Paulo Tabuada. “A Notion of Robustness for Cyber-Physical Systems”. In: *IEEE Transactions on Automatic Control* 61.8 (2016), pp. 2108–2123. DOI: 10.1109/TAC.2015.2492438 (cit. on p. 166).
- [293] Éric Rutten, Nicolas Marchand, and Daniel Simon. “Feedback Control as MAPE-K Loop in Autonomic Computing”. In: *Software Engineering for Self-Adaptive Systems III. Assurances - International Seminar, Dagstuhl Castle, Germany, December 15-19, 2013, Revised Selected and Invited Papers*. Ed. by Rogério de Lemos, David Garlan, Carlo Ghezzi, and Holger Giese. Vol. 9640. Lecture Notes in Computer Science. Springer, 2013, pp. 349–373. DOI: 10.1007/978-3-319-74183-3_12 (cit. on p. 169).
- [294] Dorsa Sadigh, Eric S. Kim, Samuel Coogan, S. Shankar Sastry, and Sanjit A. Seshia. “A learning based approach to control synthesis of Markov decision processes for linear temporal logic specifications”. In: *53rd IEEE Conference on Decision and Control, CDC 2014, Los Angeles, CA, USA, December 15-17, 2014*. IEEE, 2014, pp. 1091–1096. ISBN: 978-1-4799-7746-8. DOI: 10.1109/CDC.2014.7039527. URL: <https://doi.org/10.1109/CDC.2014.7039527> (cit. on p. 216).
- [295] Neda Saeedloei and Gopal Gupta. “Verifying Complex Continuous Real-Time Systems with Coinductive CLP(R)”. In: *Language and Automata Theory and Applications, 4th International Conference, LATA 2010, Trier, Germany, May 24-28, 2010. Proceedings*. Ed. by Adrian-Horia Dediu, Henning Fernau, and Carlos Martín-Vide. Vol. 6031. Lecture Notes in Computer Science. Springer, 2010, pp. 536–548. DOI: 10.1007/978-3-642-13089-2_45 (cit. on p. 180).
- [296] Yoshisato Sakai. “A Tableau Construction Approach to Control Synthesis of FSMs Using Simulation Relations”. In: *IEICE Transactions* 90-A.4 (2007), pp. 836–846. DOI: 10.1093/ietfec/e90-a.4.836 (cit. on p. 171).
- [297] Antonia M. Sánchez and Francisco J. Montoya. “Safe Supervisory Control Under Observability Failure”. In: *Discrete Event Dynamic Systems* 16.4 (2006), pp. 493–525. DOI: 10.1007/s10626-006-0022-8 (cit. on p. 173).
- [298] Germano Schafaschek, Max Hering de Queiroz, and José Eduardo Ribeiro Cury. “Local Modular Supervisory Control of Timed Discrete-Event Systems”. In: *IEEE Transactions on Automatic Control* 62.2 (2017), pp. 934–940. DOI: 10.1109/TAC.2016.2566884 (cit. on pp. 174, 175).

- [299] Klaus Werner Schmidt. “Optimal Supervisory Control of Discrete Event Systems: Cyclicity and Interleaving of Tasks”. In: *SIAM Journal on Control and Optimization (SICON)* 53:3 (2015), pp. 1425–1439. DOI: 10.1137/120892908 (cit. on p. 165).
- [300] Anne-Kathrin Schmuck. “Building bridges in abstraction-based controller synthesis: advancing, combining, and comparing methods from computer and control”. PhD thesis. Berlin, Germany: Technische Universität Berlin, 2015. ISBN: 978-3-7375-7174-6. DOI: 10.14279/depositonce-4906 (cit. on pp. 147, 148).
- [301] Anne-Kathrin Schmuck, Sven Schneider, Jörg Raisch, and Uwe Nestmann. “Extending Supervisory Controller Synthesis to Deterministic Pushdown Automata—Enforcing Controllability Least Restrictively”. In: *12th International Workshop on Discrete Event Systems, WODES 2014, Cachan, France, May 14-16, 2014*. Ed. by Jean-Jacques Lesage, Jean-Marc Faure, José E. R. Cury, and Bengt Lennartson. International Federation of Automatic Control, 2014, pp. 286–293. ISBN: 978-3-902823-61-8. DOI: 10.3182/20140514-3-FR-4046.00058 (cit. on pp. 12, 82, 180, 181, 225).
- [302] Anne-Kathrin Schmuck, Sven Schneider, Jörg Raisch, and Uwe Nestmann. “Supervisory control synthesis for deterministic context free specification languages—Enforcing controllability least restrictively”. In: *Discrete Event Dynamic Systems* 26.1 (2016), pp. 5–32. DOI: 10.1007/s10626-015-0221-2 (cit. on pp. 12, 82, 147, 148, 181, 185, 225).
- [303] Sven Schneider. *An Isabelle Formalization for Controller Synthesis using Deterministic Pushdown Automata as Specifications*. doi: 10.5281/zenodo.1412369. 2018. URL: <https://github.com/ControllerSynthesis/Isabelle> (cit. on pp. 13, 89, 96, 140, 220).
- [304] Sven Schneider. *CoSy: A Tool for Controller Synthesis using Deterministic Pushdown Automata as Specifications*. doi: 10.5281/zenodo.1346258. 2018. URL: <https://github.com/ControllerSynthesis/CoSy> (cit. on pp. 13, 141, 154, 161, 220).
- [305] Sven Schneider, Leen Lambers, and Fernando Orejas. “Automated reasoning for attributed graph properties”. In: *International Journal on Software Tools for Technology Transfer* (June 2018). ISSN: 1433-2787. DOI: 10.1007/s10009-018-0496-3. URL: <https://doi.org/10.1007/s10009-018-0496-3> (cit. on p. 12).
- [306] Sven Schneider, Leen Lambers, and Fernando Orejas. “Symbolic Model Generation for Graph Properties”. In: *Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*. Ed. by Marieke Huisman and Julia Rubin. Vol. 10202. Lecture Notes in Computer Science. Springer, 2017, pp. 226–243. ISBN: 978-3-662-54493-8. DOI: 10.1007/978-3-662-54494-5_13. URL: https://doi.org/10.1007/978-3-662-54494-5_13 (cit. on p. 12).
- [307] Sven Schneider and Uwe Nestmann. “Enforcing Operational Properties including Block-freeness for Deterministic Pushdown Automata”. In: *CoRR* (2014). URL: <http://arxiv.org/abs/1403.5081> (cit. on pp. 12, 181).
- [308] Sven Schneider and Uwe Nestmann. “Rigorous Discretization of Hybrid Systems Using Process Calculi”. In: *Formal Modeling and Analysis of Timed Systems - 9th International Conference, FORMATS 2011, Aalborg, Denmark, September 21-23, 2011. Proceedings*. Ed. by Uli Fahrenberg and Stavros Tripakis. Vol. 6919. Lecture Notes in Computer Science. Springer, 2011, pp. 301–316. DOI: 10.1007/978-3-642-24310-3_21 (cit. on pp. 12, 216).
- [309] Sven Schneider and Anne-Kathrin Schmuck. *Supervisory Controller Synthesis for Deterministic Pushdown Automata Specifications*. Tech. rep. Technical University of Berlin, 2013. URL: https://www.eecs.tu-berlin.de/fileadmin/f4/TechReports/2013/tr_2013-09.pdf (cit. on pp. 12, 82, 180, 181).
- [310] Sven Schneider, Anne-Kathrin Schmuck, Uwe Nestmann, and Jörg Raisch. “Reducing an Operational Supervisory Control Problem by Decomposition for Deterministic Pushdown Automata”. In: *12th International Workshop on Discrete Event Systems, WODES 2014, Cachan, France, May 14-16, 2014*. Ed. by Jean-Jacques Lesage, Jean-Marc Faure, José E. R. Cury, and Bengt Lennartson. International Federation of Automatic Control, 2014, pp. 214–221.

- ISBN: 978-3-902823-61-8. DOI: 10.3182/20140514-3-FR-4046.00057 (cit. on pp. 12, 17, 35, 96, 108, 165, 181, 225).
- [311] Melanie Schuh and Jan Lunze. “Tracking control of deterministic I/O automata”. In: *13th International Workshop on Discrete Event Systems, WODES 2016, Xi'an, China, May 30 - June 1, 2016*. Ed. by Christos G. Cassandras, Alessandro Giua, and Zhiwu Li. IEEE, 2016, pp. 325–331. ISBN: 978-1-5090-4190-9. DOI: 10.1109/WODES.2016.7497867. URL: <https://doi.org/10.1109/WODES.2016.7497867> (cit. on p. 171).
 - [312] Stephan Schulz. *The E Theorem Prover*. 2017. URL: <http://www.lehre.dhbw-stuttgart.de/~ssschulz/E/E.html> (cit. on p. 93).
 - [313] Raja Sengupta. “Optimal control of discrete event systems”. PhD thesis. University of Michigan, 1995 (cit. on p. 165).
 - [314] Raja Sengupta and Stéphane Lafortune. “A deterministic optimal control theory for discrete event systems”. In: *Proceedings of 32nd IEEE Conference on Decision and Control*. Dec. 1993, pp. 1182–1187. DOI: 10.1109/CDC.1993.325369 (cit. on p. 165).
 - [315] Raja Sengupta and Stéphane Lafortune. “A graph-theoretic optimal control problem for terminating discrete event processes”. In: *Discrete Event Dynamic Systems 2.2* (1992), pp. 139–172. DOI: 10.1007/BF01797725 (cit. on p. 165).
 - [316] Raja Sengupta and Stéphane Lafortune. “An optimal control theory for discrete event systems”. In: *SIAM Journal on Control and Optimization (SICON)* 36.2 (1998), pp. 488–541. ISSN: 0363-0129 (cit. on p. 165).
 - [317] Raja Sengupta and Stéphane Lafortune. “Optimal Control of a Class of Discrete Event Systems”. In: *IFAC Proceedings Volumes 24.5* (1991). IFAC Symposium on Distributed Intelligence Systems, Arlington, VA, USA, 13-15 August 1991, pp. 29–34. DOI: 10.1016/S1474-6670(17)51219-7 (cit. on p. 165).
 - [318] Géraud Sénizergues. “The Equivalence Problem for Deterministic Pushdown Automata is Decidable”. In: *Automata, Languages and Programming, 24th International Colloquium, ICALP'97, Bologna, Italy, 7-11 July 1997, Proceedings*. Ed. by Pierpaolo Degano, Roberto Gorrieri, and Alberto Marchetti-Spaccamela. Vol. 1256. Lecture Notes in Computer Science. Springer, 1997, pp. 671–681. DOI: 10.1007/3-540-63165-8_221 (cit. on p. 213).
 - [319] M. W. Shields. “Implicit System Specification and the Interface Equation”. In: *The Computer Journal* 32.5 (1989), pp. 399–412. DOI: 10.1093/comjnl/32.5.399 (cit. on p. 171).
 - [320] Mehrdad Showkatbakhsh, Paulo Tabuada, and Suhas N. Diggavi. “System identification in the presence of adversarial outputs”. In: *55th IEEE Conference on Decision and Control, CDC 2016, Las Vegas, NV, USA, December 12-14, 2016*. IEEE, 2016, pp. 7177–7182. ISBN: 978-1-5090-1837-6. DOI: 10.1109/CDC.2016.7799376 (cit. on p. 165).
 - [321] Deepinder P. Sidhu and Juan Aristizabal. “Constructing Submodule Specifications and Network Protocols”. In: *IEEE Transactions on Software Engineering* 14.11 (1988), pp. 1565–1577. DOI: 10.1109/32.9045 (cit. on p. 171).
 - [322] Celso F. Silva, Camilo Quintáns, Antonio Colmenar, Manuel Castro, and Enrique Mandado. “A Method Based on Petri Nets and a Matrix Model to Implement Reconfigurable Logic Controllers”. In: *IEEE Transactions on Industrial Electronics* 57.10 (2010), pp. 3544–3556. DOI: 10.1109/TIE.2009.2038946 (cit. on p. 172).
 - [323] M. Silva and S. Velilla. “Programmable Logic Controllers and Petri Nets: A Comparative Study”. In: *IFAC Proceedings Volumes 15.7* (1982). 3rd IFAC/IFIP Symposium on Software for Computer Control 1982, Madrid, Spain, 5-8 October, pp. 83–88. DOI: 10.1016/S1474-6670(17)62804-0 (cit. on p. 172).
 - [324] Vladimir Sinyakov and Antoine Girard. “Formal Controller Synthesis from Hybrid Programs”. In: *Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (part of CPS Week), HSCC 2018, Porto, Portugal, April 11-13, 2018*. ACM, 2018, pp. 271–272. DOI: 10.1145/3178126.3186998 (cit. on p. 168).

- [325] Seppo Sippo and Eljas Soisalon-Soininen. *Parsing Theory*. Ed. by Wilfried Brauer, Grzegorz Rozenberg, and Arto Salomaa. 2nd ed. Vol. I(Languages and Parsing),II(LR(k) and LL(k) Parsing). EATCS Monographs on Theoretical Computer Science. Heidelberg: Springer, 1990 (cit. on pp. 22, 24, 27, 29, 66, 67, 69, 72, 73, 76, 116, 121, 124, 125, 157, 214).
- [326] IEEE Control Systems Society. *Tools for Discrete Event Controller Synthesis*. 2018. URL: <http://discrete-event-systems.ieeecss.org/tc-discrete/resources> (cit. on p. 171).
- [327] Ramavarapu S. Sreenivas. "A note on deciding the controllability of a language K with respect to a language L". In: *IEEE Transactions on Automatic Control* 38.4 (Apr. 1993), pp. 658–662. DOI: 10.1109/9.250543 (cit. on p. 178).
- [328] Ramavarapu S. Sreenivas. "On a weaker notion of controllability of a language K with respect to a language L". In: *IEEE Transactions on Automatic Control* 38.9 (Nov. 1993), pp. 1446–1447. DOI: 10.1109/9.237665 (cit. on p. 180).
- [329] Ramavarapu S. Sreenivas and Bruce H. Krogh. "On Petri net models of infinite state supervisors". In: *IEEE Transactions on Automatic Control* 37.2 (Feb. 1992), pp. 274–277. DOI: 10.1109/9.121634 (cit. on p. 178).
- [330] Colin Stirling. "An Introduction to Decidability of DPDA Equivalence". In: *FST TCS 2001: Foundations of Software Technology and Theoretical Computer Science, 21st Conference, Bangalore, India, December 13–15, 2001, Proceedings*. Ed. by Ramesh Hariharan, Madhavan Mukund, and V. Vinay. Vol. 2245. Lecture Notes in Computer Science. Springer, 2001, pp. 42–56. DOI: 10.1007/3-540-45294-X_4 (cit. on p. 213).
- [331] Colin Stirling. "Decidability of DPDA equivalence". In: *Theoretical Computer Science* 255.1-2 (2001), pp. 1–31. DOI: 10.1016/S0304-3975(00)00389-3 (cit. on p. 213).
- [332] Colin Stirling and David Walker. "Local Model Checking in the Modal μ -Calculus". In: *TAPSOFT'89: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Barcelona, Spain, March 13–17, 1989, Volume 1: Advanced Seminar on Foundations of Innovative Software Development I and Colloquium on Trees in Algebra and Programming (CAAP'89)*. Ed. by Josep Díaz and Fernando Orejas. Vol. 351. Lecture Notes in Computer Science. Springer, 1989, pp. 369–383. DOI: 10.1007/3-540-50939-9_144 (cit. on p. 169).
- [333] Ichiro Suzuki and Tadao Murata. "A Method for Stepwise Refinement and Abstraction of Petri Nets". In: *Journal of Computer and System Sciences* 27.1 (1983), pp. 51–76. DOI: 10.1016/0022-0000(83)90029-6 (cit. on p. 178).
- [334] Daniel Sykes, Domenico Corapi, Jeff Magee, Jeff Kramer, Alessandra Russo, and Katsumi Inoue. "Learning revised models for planning in adaptive systems". In: *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18–26, 2013*. Ed. by David Notkin, Betty H. C. Cheng, and Klaus Pohl. IEEE Computer Society, 2013, pp. 63–71. ISBN: 978-1-4673-3076-3. DOI: 10.1109/ICSE.2013.6606552 (cit. on p. 170).
- [335] IAR Systems. *VisualState*. 2018. URL: <https://www.iar.com/iar-embedded-workbench/add-ons-and-integrations/visualstate/> (cit. on p. 172).
- [336] Shigemasa Takai. "Synthesis of maximally permissive and robust supervisors for prefix-closed language specifications". In: *IEEE Transactions on Automatic Control* 47.1 (2002), pp. 132–136. DOI: 10.1109/9.981732 (cit. on p. 166).
- [337] Zhongping Tao, Gregor von Bochmann, and Rachida Dssouli. "A Formal Method for Synthesizing Optimized Protocol Converters and Its Application to Mobile Data Networks". In: *Mobile Networks and Applications (MONET)* 2.3 (1997), pp. 259–269. DOI: 10.1023/A:1013692902855 (cit. on p. 171).
- [338] Alfred Tarski. "A lattice-theoretical fixpoint theorem and its applications". In: *Pacific Journal of Mathematics* 5.2 (1955), pp. 285–309. URL: <http://projecteuclid.org/euclid.pjm/1103044538> (cit. on p. 47).

- [339] John G. Thistle. “On control of systems modelled as deterministic Rabin automata”. In: *Discrete Event Dynamic Systems* 5.4 (1995), pp. 357–381. doi: 10.1007/BF01439153 (cit. on pp. 168, 171).
- [340] John G. Thistle and Hichem M. Lamouchi. “Effective Control Synthesis for Partially Observed Discrete-Event Systems”. In: *SIAM Journal on Control and Optimization (SICON)* 48.3 (2009), pp. 1858–1887. doi: 10.1137/060673862 (cit. on p. 168).
- [341] John G. Thistle and R. P. Malhamé. “Control of ω -automata under state fairness assumptions”. In: *Systems & Control Letters* 33.4 (1998), pp. 265–274. doi: 10.1016/S0167-6911(97)00106-0 (cit. on p. 168).
- [342] John G. Thistle and Walter Murray Wonham. “Control of ω -Automata, Church’s Problem, and the Emptiness Problem for Tree ω -Automata”. In: *Computer Science Logic, 5th Workshop, CSL ’91, Berne, Switzerland, October 7-11, 1991, Proceedings*. Ed. by Egon Börger, Gerhard Jäger, Hans Kleine Büning, and Michael M. Richter. Vol. 626. Lecture Notes in Computer Science. Springer, 1991, pp. 367–382. doi: 10.1007/BFb0023782 (cit. on p. 168).
- [343] John G. Thistle and Walter Murray Wonham. “Control of Infinite Behavior of Finite Automata”. In: *SIAM Journal on Control and Optimization (SICON)* 32.4 (1994), pp. 1075–1097. doi: 10.1137/S0363012991217536 (cit. on p. 168).
- [344] John G. Thistle and Walter Murray Wonham. “Supervision of Infinite Behavior of Discrete-Event Systems”. In: *SIAM Journal on Control and Optimization (SICON)* 32.4 (July 1994), pp. 1098–1113. doi: 10.1137/S0363012991217524 (cit. on p. 168).
- [345] Dmitriy Traytel. *A Codatatype of Formal Languages*. 2013. URL: http://www.isa-afp.org/entries/Coinductive_Languages.shtml (cit. on p. 187).
- [346] Alan Mathison Turing. “On computable numbers, with an application to the Entscheidungsproblem.” English. In: *Proceedings of the London Mathematical Society* (2) 42 (1936), pp. 230–265. doi: 10.1112/plms/s2-42.1.230 (cit. on p. 111).
- [347] Esko Ukkonen. “Upper Bounds on the Size of LR(k) Parsers”. In: *Information Processing Letters* 20.2 (1985), pp. 99–103. doi: 10.1016/0020-0190(85)90072-9 (cit. on p. 157).
- [348] Alphan Ulusoy, Tichakorn Wongpiromsarn, and Calin Belta. “Incremental controller synthesis in probabilistic environments with temporal logic constraints”. In: *International Journal of Robotics Research (IJRR)* 33.8 (2014), pp. 1130–1144. doi: 10.1177/0278364913519000 (cit. on p. 166).
- [349] Vladimir Ulyantsev, Igor Buzhinsky, and Anatoly Shalyto. “Exact finite-state machine identification from scenarios and temporal properties”. In: *STTT* 20.1 (2018), pp. 35–55. doi: 10.1007/s10009-016-0442-1. URL: <https://doi.org/10.1007/s10009-016-0442-1> (cit. on p. 215).
- [350] R. Valette, M. Courvoisier, J.M. Bigou, and J. Albukerque. “A Petri net based programmable logic controller”. In: *IFIP First International Conference on Computer Applications in Production and Engineering CAPE 83, Amsterdam (Apr. 1983)*. Ed. by E. A. Warman, pp. 103–116 (cit. on p. 178).
- [351] Moshe Ya’akov Vardi. “An Automata-Theoretic Approach to Fair Realizability and Synthesis”. In: *Computer Aided Verification, 7th International Conference, Liège, Belgium, July, 3-5, 1995, Proceedings*. Ed. by Pierre Wolper. Vol. 939. Lecture Notes in Computer Science. Springer, 1995, pp. 267–278. doi: 10.1007/3-540-60045-0_56 (cit. on p. 168).
- [352] László Zsolt Varga. “Game Theory Models for the Verification of the Collective Behaviour of Autonomous Cars”. In: *Proceedings First Workshop on Formal Verification of Autonomous Vehicles, FVAV@iFM 2017, Turin, Italy, 19th September 2017*. Ed. by Lukas Bulwahn, Maryam Kamali, and Sven Linker. EPTCS. 2017, pp. 27–34. doi: 10.4204/EPTCS.257.4. URL: <http://arxiv.org/abs/1709.02126> (cit. on p. 168).
- [353] Andrei Voronkov, Alexandre Riazanov, Krystof Hoder, Laura Kovacs, and Ioan Drăgan. *Vampire’s Home Page*. 2017. URL: <http://www.vprover.org/> (cit. on p. 94).

- [354] David A. Wagner and Drew Dean. "Intrusion Detection via Static Analysis". In: *2001 IEEE Symposium on Security and Privacy, Oakland, California, USA May 14-16, 2001*. IEEE Computer Society, 2001, pp. 156–168. ISBN: 0-7695-1046-9. DOI: 10.1109/SECPR1.2001.924296 (cit. on p. 180).
- [355] Masashi Wakaiki, Paulo Tabuada, and João Pedro Hespanha. "Supervisory Control of Discrete-Event Systems under Attacks". In: *CoRR* (2017). URL: <http://arxiv.org/abs/1701.00881> (cit. on p. 166).
- [356] Richard Jay Waldinger and Zohar Manna. "Knowledge and Reasoning in Program Synthesis". In: *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence, Tbilisi, Georgia, USSR, September 3-8, 1975*. 1975, pp. 288–295. URL: <http://ijcai.org/Proceedings/75/Papers/041.pdf> (cit. on p. 1).
- [357] Charles R. Wallace, Paul Jensen, and Nandit Soparkar. "Supervisory control of workflow scheduling". In: *Advanced Transaction Models & Architectures Workshop (ATMA)*. 1996. DOI: 10.1109/CDC.1984.272178 (cit. on p. 169).
- [358] Xinyu Wang, Isil Dillig, and Rishabh Singh. "Program synthesis using abstraction refinement". In: *Proceedings of the ACM on Programming Languages (PACMPL)* 2.POPL (2018), pp. 1–30. DOI: 10.1145/3158151 (cit. on p. 167).
- [359] Yin Wang, Hyoun Kyu Cho, Hongwei Liao, Ahmed Nazeem, Terence Kelly, Stéphane Lafortune, Scott A. Mahlke, and Spyros A. Reveliotis. "Supervisory control of software execution for failure avoidance: Experience from the Gadara project". In: *10th International Workshop on Discrete Event Systems, WODES 2010, Berlin, Germany, August 30 - September 01, 2010*. Ed. by Jörg Raisch, Alessandro Giua, Stéphane Lafortune, and Thomas Moor. International Federation of Automatic Control, 2010, pp. 259–266. ISBN: 978-3-902661-79-1. DOI: 10.3182/20100830-3-DE-4013.00044 (cit. on p. 170).
- [360] Yin Wang, Stéphane Lafortune, Terence Kelly, Manjunath Kudlur, and Scott A. Mahlke. "The theory of deadlock avoidance via discrete control". In: *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*. Ed. by Zhong Shao and Benjamin C. Pierce. ACM, 2009, pp. 252–263. ISBN: 978-1-60558-379-2. DOI: 10.1145/1480881.1480913 (cit. on p. 170).
- [361] Simon Wimmer. *Timed Automata*. 2016. URL: http://www.isa-afp.org/entries/Timed_Automata.shtml (cit. on p. 187).
- [362] Kai C. Wong and Walter Murray Wonham. "Hierarchical control of discrete-event systems". In: *Discrete Event Dynamic Systems* 6.3 (1996), pp. 241–273. DOI: 10.1007/BF01797154 (cit. on p. 176).
- [363] Kai C. Wong and Walter Murray Wonham. "Hierarchical control of timed discrete-event systems". In: *Discrete Event Dynamic Systems* 6.3 (1996), pp. 275–306. DOI: 10.1007/BF01797155 (cit. on pp. 174, 176).
- [364] Kai C. Wong and Walter Murray Wonham. "Modular Control and Coordination of Discrete-Event Systems". In: *Discrete Event Dynamic Systems* 8.3 (1998), pp. 247–297. DOI: 10.1023/A:1008210519960 (cit. on p. 175).
- [365] Walter Murray Wonham and Peter Jeffrey Godwin Ramadge. "Modular supervisory control of discrete-event systems". In: *Mathematics of Control, Signals, and Systems (MCSS)* 1.1 (1988), pp. 13–30. DOI: 10.1007/BF02551233 (cit. on p. 175).
- [366] Chunhan Wu, Xingyuan Zhang, and Christian Urban. *The Myhill-Nerode Theorem Based on Regular Expressions*. 2011. URL: <http://www.isa-afp.org/entries/Myhill-Nerode.shtml> (cit. on p. 186).
- [367] Guanci Yang. "Game Theory-Inspired Evolutionary Algorithm for Global Optimization". In: *Algorithms* 10.4 (2017), pp. 111–126. DOI: 10.3390/a10040111 (cit. on p. 168).

- [368] Nina Yevtushenko, Tiziano Villa, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli Alexandre Petrenko. *Sequential Synthesis by Language Equation solving*. Tech. rep. UCB/ERL Mo3/9. EECS Department, University of California, Berkeley, 2003. URL: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2003/4063.html> (cit. on p. 171).
- [369] Renyuan Zhang, Kai Cai, Yongmei Gan, and Walter Murray Wonham. “Delay-robustness in distributed control of timed discrete-event systems based on supervisor localisation”. In: *International Journal of Control* 89.10 (2016), pp. 2055–2072. DOI: 10.1080/00207179.2016.1147606 (cit. on pp. 166, 174, 175).
- [370] Hao Zhong and Walter Murray Wonham. “On the consistency of hierarchical supervision in discrete-event systems”. In: *IEEE Transactions on Automatic Control* 35.10 (Oct. 1990), pp. 1125–1134. DOI: 10.1109/9.58555 (cit. on p. 176).
- [371] Changyan Zhou, Ratnesh Kumar, and Ramavarapu S. Sreenivas. “Decentralized modular control of concurrent discrete event systems”. In: *46th IEEE Conference on Decision and Control, CDC 2007, New Orleans, LA, USA, December 12-14, 2007*. IEEE, 2007, pp. 5918–5923. DOI: 10.1109/CDC.2007.4434489 (cit. on p. 175).
- [372] Meng Chu Zhou and Frank DiCesare. *Petri Net Synthesis for Discrete Event Control of Manufacturing Systems*. The Springer International Series in Engineering and Computer Science. Springer US, 2012. ISBN: 9781461531265 (cit. on p. 178).
- [373] Meng Chu Zhou, Frank DiCesare, and Alan A. Desrochers. “Hybrid methodology for the synthesis of Petri nets models for manufacturing systems”. In: *IEEE Transactions on Robotics and Automation* 8.3 (1992), pp. 350–361 (cit. on p. 178).
- [374] Meng Chu Zhou, Frank DiCesare, and Daryl L. Rudolph. “Design and implementation of a petri net based supervisor for a flexible manufacturing system”. In: *Automatica* 28.6 (1992), pp. 1199–1208. DOI: 10.1016/0005-1098(92)90061-J (cit. on p. 178).