An Isabelle Formalization for Controller Synthesis using Deterministic Pushdown Automata as Specifications

An Isabelle Formalization for Controller Synthesis using Deterministic Pushdown Automata as Specifications

Sven Schneider

September 10, 2018

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

Contents

1

Introduction

The supervisory control problem was introduced in [3] and requires the synthesis of a least-restrictive satisfactory controller for a plant and a specification.

We present our Isabelle-based [2] formalization of an abstract controller synthesis algorithm and of a concrete controller synthesis algorithm that is an instantiation thereof. This formalization consists of definitions and proofs in Isabelle. The algorithms are both fixed-point algorithms. The abstract controller synthesis algorithm synthesizes a DES controller when being provided with a DES plant and a DES specification while the concrete controller synthesis algorithm synthesizes a DPDA controller when being provided with a DFA plant and a DPDA specification. Both algorithms reduce controllability to nonblockingness but do not terminate in general. While this is acceptable for DES that allow arbitrary languages, this is also true for the case of the concrete controller synthesis algorithm for some specifications that are not appropriate from a control theory perspective. The formal details of these algorithms are covered in the phd-thesis of the author.

In this document, we briefly discuss several aspects that can be helpful for future users and maintainers of this formalization to get started. That is, we merely mention entry points to the project and do not provide an in-depth discussion of all aspects.

Contents of this Document

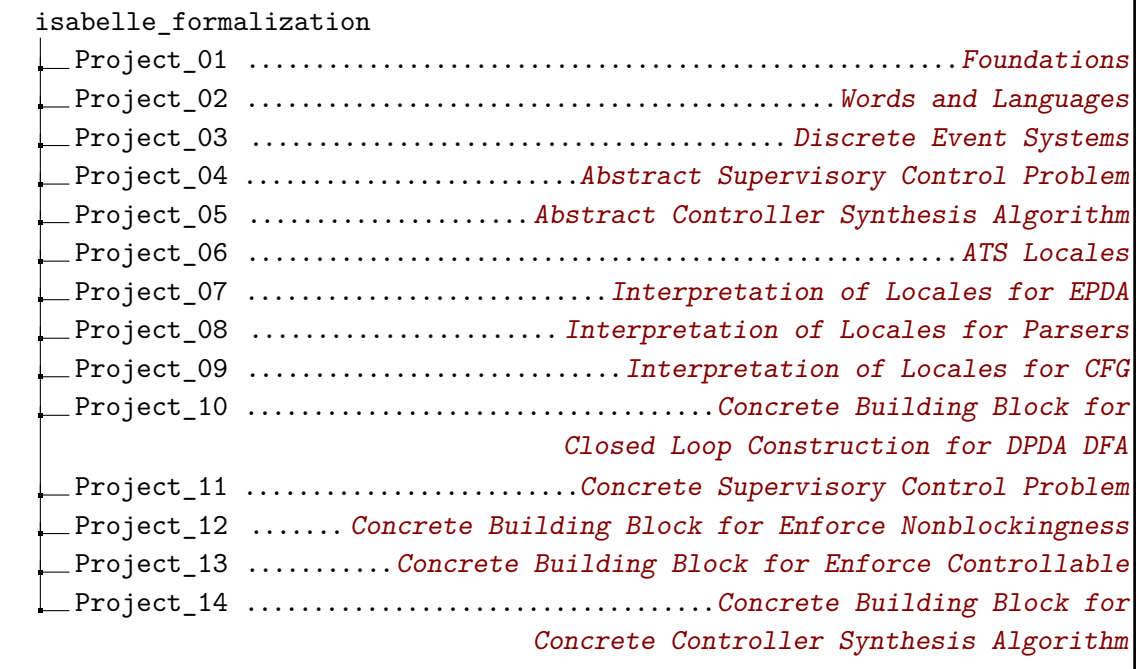
2

Overview of Structure and Files

We have formalized our abstract controller synthesis algorithm and our concrete controller synthesis algorithm in Isabelle [2]. We have used 82 projects to keep each project rather short. The project $i + 1$ (for $0 \leq i \leq 81$) depends on the project i and the first project depends on the built-in library *Main*.

The following figure visualizes the basic structure of the formalization. We grouped the 82 projects into 14 main projects of which some then have sub*projects.

Figure 2.1: «Structure of Isabelle Formalization»



The names of the projects indicate nesting as in *PRJ_12_04_06_04* where in the project 12 there is subproject 4 with a subproject 6 with a subproject 4.

2.1. How to Check the Formalization Using Isabelle

To use our formalization we recommend the following steps and scripts.

2.1.1. Setup

1. Download and install Isabelle [2] (for Linux and Windows we assume that the archive is extracted to *~/Downloads*).
   - Linux: the *bash* is assumed in the following.
   - Windows: the *bash* can be started using *Isabelle2018/Cygwin-Terminal.bat*.
   - macOS: the *terminal* should be used.
2. Execute the Isabelle binary once as in
   - Linux: *~/Downloads/Isabelle2018/bin/isabelle jedit*
   - Windows: *~/Downloads/Isabelle2018/bin/isabelle jedit*
   - macOS: */Applications/Isabelle2018.app/Isabelle/bin/isabelle jedit*
3. Download the isabelle formalization into a directory of your choice.
4. Change directory in the shell to that directory.
5. (possibly necessary) Change the paths in *config/config.sh* if required
6. (possibly necessary) Make sure that *pdflatex* is available

2.1.2. Check All Subprojects

Execute *bash bin/create_image 82* to check each project, to build the document (a PDF containing definitions, theorems, and proofs), and the outline (a PDF containing definitions and theorems but no proofs) for each project (this may require some time). The PDF files are then in each the *output* subdirectory of each project. The subproject *PRJ_12_04_06_06_10* may require up to 30 min whereas the other subprojects are checked rather fast.

2.1.3. Inspect One Subprojects

Moreover, we provide support to open a certain subproject.

- *bash bin/open.sh 45* opens the 45th subproject.
- *bash bin/open.sh PRJ_12_04_06_06_05* opens the subproject *PRJ_12_04_06_-06_05*.

2.1.4. Generate Document and Outline PDF

Finally, use *bash bin/full_document_outline.sh* to create document and outline PDF files in the output directory once all subprojects have been checked. This script generates the files *output/document.pdf* and *output/outline.pdf*.

3

Discrete Event Systems and Control Theoretic Notions

The Isabelle type of Discrete Event Systems (DES) is introduced in *PRJ_03/discrete_event_systems.thy* together with the instantiation of the *complete_lattice* type class.

Nonblockingness and controllability are defined for languages in *PRJ_04/control_theory_on_languages.thy* including facts that these properties are preserved by arbitrary union. These definitions and results are extended in *PRJ_04/control_theory_on_discrete_event_systems.thy* to DES also covering facts about the preservation of other properties such as begin a valid DES and satisfaction of DES specification. Moreover, we define in this theory already the desired least restrictive satisfactory DES controllers and relate these definitions with alternative definitions as in [4]. This elaborate comparison of different characterizations underlines the suitability of the DES based abstract supervisory control problem. The theory *PRJ_04/control_theory_on_languages__alternative_characterization_of_controllable_language.thy* contains other notions of controllability (such as some rephrasings used by Christopher Griffin). However, they are either equivalent or we do not provide automata based implementations for them (as for the stronger controllability conditions that checks for sequences of uncontrollable events that are prohibited by the controller). In the theory *PRJ_04/control_theory_on_languages__characterization_of_controllable_sublanguage.thy* we then compare notions of the least restrictive controllable sublanguage based on these alternative characterizations. These additional alternative definitions and the corresponding comparisons have not been included in our thesis.

4

An Abstract Controller Synthesis Algorithm

In the project *PRJ_05*, we introduce our framework of fixed point iterators and apply it for our purposes to define an abstract controller synthesis algorithm by providing DES-based fixed-point iterators for enforcing the properties required for the desirable controllers.

In *PRJ_05/fixed_point_iterator.thy*, we introduce fixed point iterators, results on their composition, and their usage for computing greatest fixed points.

In *PRJ_05/compute_greatest_fixed_points.thy*, we introduce the operation *Compute_Fixed_Point* that applies a function to an argument until reaching a fixed point. This function is then applied to fixed point iterators and it is shown that the repeated application computes the greatest fixed point of the function under suitable further restrictions.

In *PRJ_05/operations_on_discrete_event_systems.thy*, we introduce the DES-based fixed-point iterators mentioned above by defining them and by verifying their fundamental properties.

In *PRJ_05/composition_of_fixed_point_iterators.thy*, we obtain our abstract controller synthesis algorithm as a composition of the previous basic fixed-point iterators.

In *PRJ_05/compute_least_restrictive_satisfactory_controllers.thy*, we provide an example of nontermination from our thesis that shows that our abstract controller synthesis algorithm does not terminate for all DES specifications that use deterministic context free languages. Moreover, we verify that, upon termination, the operation *Compute_Fixed_Point* computes the desired controllers of the abstract supervisory control problem by applying the abstract controller synthesis algorithm in the form of a fixed-point iterator until reaching a fixed point. For these results we reuse the general results from *PRJ_05/compute_greatest_fixed_points.thy* and the our approach described in our thesis.

5

Framework of Semantics

In *PRJ_06_01* through *PRJ_06_07* we introduce our framework of abstract transition systems to capture later the semantics of EPDA, CFGs, and Parsers.

Figure 5.1: «File System Structure for Project 06»

```
isabelle
├─ Project_06_01 ..................................................Basic
├─ Project_06_02 ..............................................History
├─ Project_06_03 ............................................Scheduler
├─ Project_06_04 .........................................Determinism
├─ Project_06_05 ....................Nonblockingness Correspondence
├─ Project_06_06 ..........................................Translation
└─ Project_06_07 ..............................Interpretation Schemes
```

The first three subprojects introduce definitions to capture concrete transition systems. We introduce (initial) derivations in our framework, we determine variables occurring in configurations, and, partially based on these variables, we define the (un)marked languages generated. We also differentiate thereby between different kinds of semantics: linear semantics where the evolution of a derivation is guided by a schedule consumed during the steps of the derivation and branching semantics where the steps of the derivation are only restricted by, for example, the EPDA, the CFG, or the Parser at hand. Then, *PRJ_06_04* and *PRJ_06_05* focus on the semantical property of determinism and nonblockingness and compare different characterizations of it. In *PRJ_06_06*, we relate different abstract transition systems with each other while preserving certain semantical properties such as (un)marked languages, determinism, and nonblockingness. Finally, in *PRJ_06_07*, we provide listings of which parts of our framework are to be interpreted for the semantics of EPDA, CFGs, and Parsers. Subsequently, we consider each of these subprojects separately.

5.1. Project PRJ_06_01: Basic

Derivations are defined in *PRJ_06_01/PRJ_06_01/Derivations_Basics.thy* as maps from natural numbers to pairs of step labels and configurations where the pair for the natural number 0 has no step label. Thereby, derivations can be infinite. Also, we define some basic operations on derivations such as taking prefixes or suffixes of derivations as well as applying a map to every configuration of a derivation. Moreover, we provide operations for appending derivations. However, theorems about these operations are postponed to the next subproject where we have a step relation that can distinguish permitted from nonpermitted steps.

In the theory *PRJ_06_01/PRJ_06_01/L_ATS.thy*, we introduce our core locale that assumes, amongst others, the existence of a step relation and a set of configurations. Using these parameters, we define well formed derivations and initial derivations that start in initial configurations.

In *PRJ_06_01/PRJ_06_01/L_ATS_Languageo.thy*, *PRJ_06_01/PRJ_06_01/L_ATS_-Language.thy*, and *PRJ_06_01/PRJ_06_01/L_ATS_Language_by_Finite_Derivations.thy*, we define how the (un)marked language is defined and also cover the case where finite derivations are sufficient for these definitions.

In *PRJ_06_01/PRJ_06_01/L_ATS_Destinations.thy*, we introduce destinations that are all required to be accessible to let the structure that is used in an instantiation such as an EPDA, a CFG, or a Parser to be considered to be accessible. For example, these destinations are states and edges for EPDA.

In *PRJ_06_01/PRJ_06_01/L_ATS_List_Based_Effects.thy*, we refine our locale to use lists of events for (un)marked effects. The generated language thereby contain also these lists of events. Without this locale, we just assume that (initial) derivation generate (un)marked effects that are collected for the (un)marked languages.

In *PRJ_06_01/PRJ_06_01/L_ATS_Marking_Configurations.thy*, we describe the special case where marking configurations are used to determine when an initial derivation is a marking derivation from which marked effects are obtained. This special case is indeed satisfied by EPDA, CFGs, and Parsers.

In *PRJ_06_01/PRJ_06_01/L_ATS_String_State_Modification.thy*, we already describe at an abstract level a variable (typically a schedule or a history variable) that is either consumed or extended during an initial derivation. This variable is then assumed to be extractable from a given configuration and the step relation is assumed to ensure that the variable is consumed/extended as expected. However, this locale is extended in the subsequent subprojects to cover the different special cases of variables, their role, and their interplay more closely.

5.2. Project PRJ_06_02: History

In *PRJ_06_02/PRJ_06_02/L_ATS_History.thy*, we introduce a locale extension that requires operations for getting and setting a history from an to a configuration.

Also, history values and their composition is given by operations and assumptions on the operations. This general framework does not rely on histories being lists of events as in EPDA, CFGs, and Parsers; it allows also for use for semantics of formalisms where this is not the case. The history of an EPDA configuration (in some of the semantics we introduce later on) is given by all events executed in the initial derivation at hand until reaching the configuration under inspection.

5.3. Project PRJ_06_03: Scheduler

In this subproject we introduce scheduler variables for use in linear semantics of EPDA and Parsers. However, there are three different scheduler variables: scheduler variables, unfixed scheduler variables, and fixed scheduler variables. The latter two of these are only used in semantics of Parsers but not in EPDA. In the linear semantics of EPDA and Parsers, schedulers are always given by a list of events to be executed and, in the case of Parsers, by a end of input symbol (we also call the schedule to be an input to be processed). In Parsers, we can split a scheduler into two parts: the fixed part and the unfixed part, which then determine the values of the two other scheduler variables.

Besides the separation into three kinds of variables, we also distinguish on how we obtain the variables. We can obtain these variables from configurations as for the history variables (state based) or from a given initial derivation (derivation based).

In *PRJ_06_03/PRJ_06_03/L_ATS_Scheduler_Fragment.thy*, we define parts of schedulers, which are merely events for EPDA and Parsers. In *PRJ_06_03/PRJ_06_03/L_ATS_Sched.thy*, we define how schedulers are obtained from configurations and derivations.

The theories *PRJ_06_03/PRJ_06_03/L_ATS_Sched.thy*, *PRJ_06_03/PRJ_06_03/L_ATS_SchedUF_Basic.thy*, and *PRJ_06_03/PRJ_06_03/L_ATS_SchedF_Basic.thy* capture the situations where schedulers, unfixed schedulers, and fixed schedulers can be obtained at all.

The theories *PRJ_06_03/PRJ_06_03/L_ATS_SchedF_DB.thy*, *PRJ_06_03/PRJ_06_03/L_ATS_SchedF_SB.thy*, *PRJ_06_03/PRJ_06_03/L_ATS_SchedUF_DB.thy*, and *PRJ_06_03/PRJ_06_03/L_ATS_SchedUF_SB.thy* determine the state based and derivation based access to the three kinds of scheduler variables.

In *PRJ_06_03/PRJ_06_03/L_ATS_Sched_Basic.thy*, we then assume that all three variables are available. In *PRJ_06_03/PRJ_06_03/L_ATS_Sched_SB.thy*, *PRJ_06_03/PRJ_06_03/L_ATS_Sched_DBo.thy*, and *PRJ_06_03/PRJ_06_03/L_ATS_Sched_DB.thy* we then built upon *PRJ_06_03/PRJ_06_03/L_ATS_Sched.thy* and the locales for state/derivation based access to fixed/unfixed scheduler variables from above and require that the scheduler variable is given by the expected composition.

The theories *PRJ_06_03/PRJ_06_03/L_ATS_Sched_SDB.thy*, *PRJ_06_03/PRJ_06_03/L_ATS_SchedF_SDB.thy*, and *PRJ_06_03/PRJ_06_03/L_ATS_SchedUF_SDB.thy* then extend the prior considerations by relating the two access methodologies

for the cases when they are both available and require that they coincide, that is, state based and derivations based access should obtain the same schedules in each case.

5.4. Project PRJ_06_04: Determinism

We consider two kinds of determinism: the availability of unique step label to be applied in the next step (edge determinism) and the weaker form of a unique resulting configuration when a given step label is applied (target determinism). Only the latter is usually called determinism.

In *PRJ_06_01/PRJ_06_01/L_ATS.thy*, we have defined these two notions already. In *PRJ_06_04/PRJ_06_04/L_ATS_determHIST_DB.thy*, *PRJ_06_04/PRJ_06_04/L_-ATS_determHIST_SB.thy*, and *PRJ_06_04/PRJ_06_04/L_ATS_determHIST_SDB.thy* we define these notions based on the adaptation of the history variable using state based and/or derivation based access.

In *PRJ_06_04/PRJ_06_04/L_ATS_HISTCE_DB.thy* and *PRJ_06_04/PRJ_06_04/L_-ATS_HISTCE_SB.thy*, we determine by locale assumptions the case where edge determinism corresponds to the previously introduced history-based notions. In *PRJ_06_04/PRJ_06_04/L_ATS_HISTCT_DB.thy* and *PRJ_06_04/PRJ_06_04/L_-ATS_HISTCT_SB.thy*, we determine by locale assumptions the case where target determinism corresponds to the previously introduced history-based notions.

The notions based on the histories are simpler in some applications compared to the definitions from *PRJ_06_01/PRJ_06_01/L_ATS.thy* because they only consider the history variables obtained.

5.5. Project PRJ_06_05: Nonblockingness Correspondence

We introduce locales and definitions for nonblockingness. There are various definitions because we cover branching and linear semantics, derivation based access and state based access to history variables, history based or regular determinism, the implication from operational nonblockingness to language based nonblockingness and vice versa, and whether the unfixed scheduler is still extendable. The different cases result in 26 different theory files (omitted here) of which some are then used for concrete EPDA and Parser semantics. However, we show at the abstract level already that the various notions coincide under suitable locale assumptions.

5.6. Project PRJ_06_06: Translation

We provide theories for the translation of semantical properties among two semantics. These theories differ in the results established and the assumptions made on the two semantics (the relationships are established using (weak) (bi)simulations on configurations/derivations, isomorphisms, and maps on derivations). We

also provide locales to relate linear and branching semantics to relate different semantics of one common structure. More details are given in our thesis.

5.7. Project PRJ_06_07: Interpretation Schemes

We determine locales that are then to be interpreted for semantics of EPDA, Parsers, and CFGs. The selected locales thereby describe certain semantical properties satisfied by these semantics and, for the interpretation, rely on concrete definitions for the assumed parameters of the selected locales. We provide lists for the branching and linear semantics of EPDA and Parsers and, moreover, also for the semantics of CFG that can be understood to be branching semantics in each case.

6

EPDA and EPDA Semantics

We have formalized DPDA, DFA, and various other automata based formalisms that are also restrictions of the umbrella formalism of EPDA. For more details refer to our thesis. EPDA are pushdown automata where the pop and push components of edges are arbitrary lists of stack symbols (with the restriction that if the end-of-stack symbol is popped, then it is also pushed to ensure that it remains precisely at the bottom of the stack throughout all initial derivations). We have defined the semantics of *epdaH* (configurations contain a history variable), *epdaHS* (configurations contain a history and a scheduler variable), and *epdaS* (configurations contain a scheduler variable) for EPDA using the locale hierarchy from the previous chapter. Using the theories containing locales for relation semantics we have verified that the three semantics can be used interchangeably w.r.t. semantical properties such as unmarked and marked language, determinism, nonblockingness. This also resulted in explicit translations among the initial derivations of these semantics. Throughout the formalization of our concrete controller synthesis algorithm we also developed various fundamental theorems on EPDA and their restrictions that are of general importance. That is, we believe that they can be used in other applications of (restrictions of) EPDA as well. We used primarily the branching semantic *epdaH* and the linear semantic *epdaS* and defined the semantic *epdaHS* merely to relate both of them. The advantages of branching and linear semantics compared to each other are discussed in the thesis.

7

Parsers and Parser Semantics

We have formalized Parsers in our thesis where we provided many further results. Parsers are very similar to pushdown automata but allow to test (without executing/removing) for the k next elements of the scheduler (in linear semantics), they can also determine that the end of this schedule has been reached, and they can execute multiple events at once. We have defined the semantics of *parserHF* (configurations contain a history and a fixed scheduler variable), *parserHFS* (configurations contain a history, a scheduler, and a fixed scheduler variable), *parserFS* (configurations contain a scheduler and a fixed scheduler variable), and *parserS* (configurations contain a scheduler variable) for Parsers using the locale hierarchy from the previous chapter. The fixed scheduler variable contains the part of the scheduler that has been topped by the Parser but not yet executed; this variable is not used in EPDA where it would always be empty. As for EPDA, we used the theories containing locales for relation semantics we have verified that the four semantics can be used interchangeably w.r.t. semantical properties such as unmarked and marked language, determinism, nonblockingness. This also resulted in explicit translations among the initial derivations of these semantics. Throughout the formalization of our concrete controller synthesis algorithm we also developed various fundamental theorems on Parsers and their restrictions that are of general importance. That is, we believe that they can be used in other applications of (restrictions of) Parsers as well. We used primarily the branching semantic *parserHF* and the linear semantic *parserS* and defined the semantics *parserHFS* and *parserFS* merely to relate both of them. As for EPDA, we are using branching and linear semantics in different situations as discussed in the thesis.

8

CFGs and CFG Semantics

We have formalized CFG as discussed in more detail in our thesis. We have defined the three semantics *cfgLM* (the left most nonterminal can be replaced in a step), *cfgRM* (the right most nonterminal can be replaced in a step), and *cfgSTD* (any nonterminal can be replaced in a step) for CFG using the locale hierarchy from the previous chapter. However, for CFG we used much fewer locales compared to EPDA and Parsers. Also, the correspondence between the semantics is more loose because only derivations that end in nonterminal-free configurations can be reordered (the sequence of step labels used) to obtained a derivation in *cfgRM* from a given derivation in one of the other two semantics (and similarly for *cfgLM* derivations to be obtained from *cfgRM* or *cfgSTD* derivations). However, these results already show that the three semantics are equivalent to some extend. In particular, the induced unmarked and marked language are identical. The partial translation for initial derivation not ending in non-terminal free derivations was also discussed in our thesis in the proof for showing that a DPDA can be translated into an LR(1)-CFG. Throughout the formalization of our concrete controller synthesis algorithm we also developed various fundamental theorems on CFG and LR(1)-CFG that are of general importance. That is, we believe that they can be used in other applications of CFG as well. The most important semantics for our thesis are *cfgLM* (the *cfgLM* derivations of the obtained LR(1)-CFG correspond closely to the *epdaH* derivations of the given DPDA) and *cfgRM* (the *cfgRM* derivations are used in the definition of the LR(1)-property to be verified in the above mentioned proof).

9

A Concrete Controller Synthesis Algorithm

For a detailed discussion of our concrete controller synthesis algorithm, we refer to our thesis. The basic structure is given as follows. Firstly, an initial controller candidate is computed from the provided DPDA specification and the provided DFA plant that is nonblocking by first applying the synchronous composition operation from *PRJ_10* and then the operation for enforcing nonblockingness for DPDA from *PRJ_12*. Secondly, as a fixed point computation, the current controller candidate is restricted by reducing the problem of controllability to the problem of nonblockingness in *PRJ_13* and by then enforcing nonblockingness for DPDA again from *PRJ_12*. The concrete supervisory control problem is presented in *PRJ_11*.

Figure 9.1: «File System Structure for Projects 10–14»

```
isabelle
├── Project_10 ..........................Construct Product of DPDA and DFA
│   ├── 10_01 ..................................................Construction
│   └── 10_02 ...............................................Theorems and Proofs
├── Project_11 .......................Concrete Supervisory Control Problem
├── Project_12 .Enforce Nonblockingness (more details in Figure 9.2|p.26)
├── Project_13 ..................Reduce Controllability to Nonblockingness
│   ├── 13_01 ..................................................Construction
│   ├── 13_02 ....................................Observe Top Stack of DPDA
│   ├── 13_03 ........................Enforce Unique Late Marking for SDPDA
│   ├── 13_04 ..........................................Enforce Accessibility
│   ├── 13_05 ..............................Restrict to Controllable States
│   └── 13_06 ...................................Composed Reduction Operation
└── Project_14 ...................Concrete Controller Synthesis Algorithm
    └── 14_01 ..................................................Construction
```

```
    14_02 ..............................One Step of Fixed Point Algorithm
    14_03 ..............................Iteration in Fixed Point Algorithm
    14_04 ..............................Composed Fixed Point Algorithm
```

The operations in *PRJ_13* are as follows (this construction was provided by Anne-Kathrin Schmuck but we determined the following decomposition to clarify and to verify the individual steps separately and to also rely on already established construction more clearly). In *PRJ_13_02* the given DPDA is unfolded by checking in each step for the current top stack. In *PRJ_13_03* we reuse the operation from *PRJ_12_04_05* (see below) to ensure that each marked word is marked by a certain unique initial marking derivation of the given DPDA. In *PRJ_13_04* we reuse the operation from *PRJ_12_08* (see below) to ensure that the DPDA has no inaccessible states. In *PRJ_13_05* we identify and remove states that have controllability problems (i.e., those states of the DPDA controller candidate that prevent an uncontrollable event that the DFA plant may want to execute). These steps are combined in *PRJ_13_06* where also an additional application of *PRJ_10* is used to align states of the DPDA controller candidate and the DFA plant.

The removal of violations of the nonblockingness property is much more involved compared to this reduction of controllability.

Figure 9.2: «File System Structure for Projects 12»

```
isabelle
└─Project_12 .Enforce Nonblockingness (more details in Figure 9.2|p.26)
  ├─12_01 ..................................................Construction
  ├─12_02 ..................................Generation of Fresh Elements
  ├─12_03 ..................Rename Elements in EPDA, Parsers, and CFGs
  ├─12_04 ....................................Convert DPDA to LR(1)-CFG
  │ ├─12_04_01 ................Restrict EPDA to Sets of States/Edges
  │ ├─12_04_02 ..............Approximate Accessibility of States/Edges
  │ ├─12_04_03 ......................Restrict EPDA using PRJ_12_04_02
  │ ├─12_04_04 ....................................Convert DPDA to SDPDA
  │ ├─12_04_05 ................Enforce Unique Early Marking for SDPDA
  │ └─12_04_06 ........Convert SDPDA (with Unique Marking) to LR(1)-CFG
  ├─12_05 ......Structurally Determine First Events Derivable From Word
  ├─12_06 ....................Convert LR(1)-CFG to Almost-EDPDA Parser
  ├─12_07 ..........................Convert Almost-EDPDA Parser to DPDA
  ├─12_08 ..............................Enforce Accessibility for DPDA
  ├─12_09 ................Composed Operation Enforcing Nonblockingness
  └─12_10 .......Optimized Composed Operation Enforcing Nonblockingness
```

The process is to convert the given DPDA into an SDPDA (verified in *PRJ_12_04_04* and *PRJ_12_04_05*), to convert this SDPDA into an LR(1)-CFG (verified in *PRJ_12_04_06*), to convert this LR(1)-CFG into an LR(1)-Parser (verified in *PRJ_12_06*), to convert this LR(1)-Parser into an EDPDA (verified in *PRJ_12_06* and *PRJ_12_07*), and to convert this EDPDA into the final DPDA (verified in *PRJ_12_*-

07 and *PRJ_12_08*). The violations can easily be removed one the LR(1)-CFG and the established nonblockingness is then preserved by the subsequent operations while the marked language of the DPDA is preserved in all of these steps. Also note that lifelocks are removed along the way. Finally, in *PRJ_12_08* we reuse the conversion from DPDA to LR(1)-CFG to identify states and edges that are inaccessible and remove them from the obtained DPDA that already satisfies the nonblockingness property. The usage of accessibility approximations using *PRJ_12_04_01*, *PRJ_12_04_02*, and *PRJ_12_04_03* results in an optimized conversion from SDPDA to LR(1)-CFG that has a reduced runtime. The usage of this optimized conversion ultimately results in the optimized overall construction in *PRJ_12_10* that is more efficient compared to the original construction verified in *PRJ_12_09*. Note, in ongoing work we determined a prototypical implementation of an even more efficient conversion as discussed in our thesis. The most involved verification step is given by the proof that the conversion from SDPDA to LR(1)-CFG indeed establishes the LR(1)-property; a property that was conjectured in [1] but that has not been carried out formally until now.

10

Conclusion

We have formalized an abstract controller synthesis algorithm and a concrete controller synthesis algorithm by formalizing EPDA, Parsers, and CFG. The resulting general Isabelle framework was also discussed in our thesis and is a sensible enhancement to the existing Isabelle frameworks as it supports a wider range of formalisms and various properties that are also relevant for other applications. We only sketched the basic structure of our formalization to provide an initial starting point to potential users of our framework.

A

Bibliography

[1] 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 p. 27).

[2] Lawrence Charles Paulson and Tobias Nipkow. *Isabelle/HOL*. 2017. URL: http://isabelle.in.tum.de (cit. on pp. 1, 5, 6).

[3] 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 p. 1).

[4] 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 p. 9).