CoSy: A Tool for Controller Synthesis using Deterministic Pushdown Automata as Specifications

CoSy: A Tool for Controller Synthesis using Deterministic Pushdown Automata as Specifications

Sven Schneider

August 16, 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 [4] and requires the synthesis of a least-restrictive satisfactory controller for a plant and a specification.

We present the Java-based prototype CoSy, which implements two concrete controller synthesis algorithms. Both algorithms are fixed-point algorithms and synthesize a DPDA controller when being provided with a DFA plant and a DPDA specification. The first synthesis algorithm from chapter 4|p.11 reduces controllability to nonblockingness on the level of DPDA, is formally verified in Isabelle, and does not terminate for some specifications that are not appropriate from a control theory perspective. The second synthesis algorithm from chapter 8|p.19 reduces controllability to nonblockingness on the level of LR(1)-CFG, is not formally verified, and is guaranteed to terminate as at least one production is removed in each iteration. 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 developers of CoSy 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, Packages, and Files

Our prototype CoSy is implemented in Java using Maven [5] for dependency and build management, log4j2 [1] for logging, and XML schema definitions [6] for input and output files.

The following figure visualizes the basic structure of the prototype where the part on the Java classes is refined in the subsequent figure.

Figure 2.1: «File System Structure (1/2)»

```
prototype
├── src
│   ├── main
│   │   ├── java ..... java classes discussed in more detail in Figure 2.2 p.4
│   │   └── resources
│   │       ├── cosy.xsd ..... schema definition for xml input/output files
│   │       └── log4j2.xml ..... logging configuration for main.Main
│   └── test
│       ├── java ..... java files containing unit tests (see chapter 5 p.13)
│       └── resources
│           ├── log4j2.xml ..... logging configuration for testing
│           └── unit_tests ..... xml files for unit tests
├── target
│   └── CoSy-1.0-jar-with-dependencies.jar ..... jar file for benchmarking
├── out ..... test outputs and log file
├── pom.xml ..... maven configuration file
└── benchmarking ..... scripts for benchmarking and the results
```

We consider in the following figure the package structure of our prototype in more detail. The packages of *datatypes* and *operations* contain the two algorithms with their various building blocks.

Figure 2.2: «File System Structure (2/2)»

```
java
├── main
│   ├── Main.java ..... main class for benchmarks (see chapter 7|p.17)
│   └── Config.java ..... toggles for selecting code alternatives
├── multithreading ..... support for multithreading (see chapter 6|p.15)
├── import_export ..... using xml files and cosy.xsd (see chapter 3|p.7)
├── datatypes
│   ├── basic_datatypes_of_algorithm .... datatypes used within the algorithm
│   └── formalisms ..... formalisms' data structures and semantics
│       ├── epda.java
│       ├── epda_configuration.java
│       ├── epdaH_semantics.java
│       ├── parser.java
│       ├── parser_configuration.java
│       ├── parserHF_semantics.java
│       ├── cfg.java
│       ├── cfg_configuration.java
│       ├── cfgLM_semantics.java
│       └── language_check ..... for comparing instances using semantics
└── operations ..... constructions of the two algorithms
    ├── ADDITIONAL_OPERATIONS ..... optimizations of verified constructions
    ├── p01__ENFORCE_BLOCKFREE ..... major building block 1
    ├── p02__DPDA_DFA_PRODUCT ..... major building block 2
    ├── p03__ENFORCE_CONTROLLABLE ..... major building block 3
    ├── p04__CONSTRUCT_CONTROLLER ..... 1st algorithm (see chapter 4|p.11)
    │   └── Algorithm_Definition__Construct_Controller.java
    └── p05__CFG_CONSTRUCT_CONTROLLER .... 2nd algorithm (see chapter 8|p.19)
        └── Algorithm_Definition__CFG_Construct_Controller.java
```

The main method in *Main.java* is used to apply our first concrete controller synthesis algorithm on one of the three benchmark examples presented in our thesis based arguments passed to this main method.

In *Config.java* we allow for the adaptation of building blocks by selecting alternative implementations (such as optimizations) for building blocks to be used. Moreover, we allow for the selection of additional intermediate steps to simplify instances of the formalisms between applications of building blocks (such as for the removal of inaccessible states and edges from an EPDA).

The further semantics of the three formalisms that we defined in our thesis and Isabelle are not required for the comparison of two instantiations of these

formalisms during testing regarding their generated marked and unmarked languages. Especially the linear semantics seem to have few benefits for these testing purposes.

The first concrete controller synthesis algorithm from chapter 4|p.11 uses the Java classes from the three building blocks of *po1__ENFORCE_BLOCKFREE*, *po2_-_DPDA_DFA_PRODUCT*, and *po3__ENFORCE_CONTROLLABLE* whereas the second concrete controller synthesis algorithm from chapter 8|p.19 only reuses operations from the first two building blocks and replaces the operations from the third building block by operations on LR(1)-CFG.

The implementations of the building blocks in the two packages *ADDITIONAL_-OPERATIONS* and *po5__CFG_CONSTRUCT_CONTROLLER* have not been verified in Isabelle whereas the implementations in the other packages are close to the formally verified definitions.

3

Import and Export of EPDA, CFGs, and Parsers

We are using an XML schema definition *cosy.xsd* to specify valid XML files for test cases and benchmarks that contain EPDA, CFGs, and Parsers. Moreover, these XML files can contain assertions to be tested on resulting instances of these formalisms in the form of event sequences that are either part of the marked language, part of the unmarked language, or not part of the unmarked language.

We also make use of the DOT language and Graphviz [3] to visualize EPDA and CFGs using the automatic layouting algorithms of Graphviz. Similar visualizations have been used in our thesis.

We provide subsequently three examples of XML input files for a DPDA, a CFG, and a Parser that demonstrate the syntax to be used.

Example 3.1: «Input File Using Our Schema Definition»

A DPDA in XML:

```
<?xml version="1.0"?>
<root
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="../../../main/resources/cosy.xsd">

 <!-- symbols used in the DPDA
  note, the string attribute is used for debugging and outputs -->
 <string_symbol id="event_a" string="a"/>
 <string_symbol id="event_b" string="b"/>
 <string_symbol id="box"     string="□"/>
 <string_symbol id="bullet"  string="•"/>
 <string_symbol id="state_0" string="q0"/>
 <string_symbol id="state_1" string="q1"/>
 <string_symbol id="state_2" string="q2"/>
 <string_symbol id="state_3" string="q3"/>

 <!-- edges of the DPDA
  note, empty edge_event and edge_push components are omitted -->
 <epda_step_label id="edge_0" edge_src="state_0"
  edge_event="event_a" edge_pop="box"
  edge_push="bullet box" edge_trg="state_1"/>
 <epda_step_label id="edge_1" edge_src="state_1"
  edge_event="event_a" edge_pop="bullet"
  edge_push="bullet bullet" edge_trg="state_1"/>
 <epda_step_label id="edge_2" edge_src="state_1"
  edge_event="event_b" edge_pop="bullet"
  edge_trg="state_2"/>
 <epda_step_label id="edge_3" edge_src="state_2"
  edge_event="event_b" edge_pop="bullet"
  edge_trg="state_2"/>
 <epda_step_label id="edge_4" edge_src="state_2"
  edge_pop="box"
  edge_push="box" edge_trg="state_3"/>

<!-- the DPDA with its components -->
 <epda
   id           = "dpda_a2n_b2n"
   epda_states  = "state_0 state_1 state_2 state_3"
   epda_events  = "event_a event_b"
   epda_gamma   = "box bullet"
   epda_delta   = "edge_0 edge_1 edge_2 edge_3 edge_4"
   epda_initial = "state_0"
   epda_eos     = "box"
   epda_marking = "state_0 state_3"/>
</root>
```

A Parser in XML:

```
<?xml version="1.0"?>
<root
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="../../../main/resources/cosy.xsd">

 <!-- stack symbols and events used in the PARSER -->
 <string_symbol id="nonterminal_0"      string="S"/>
 <string_symbol id="nonterminal_1"      string="S1"/>
 <string_symbol id="nonterminal_2"      string="S2"/>
 <string_symbol id="nonterminal_3"      string="S3"/>
 <string_symbol id="nonterminal_box"    string="S□"/>
 <string_symbol id="nonterminal_bullet" string="S•"/>
 <string_symbol id="event_END"          string="$"/>
 <string_symbol id="event_a"            string="a"/>
 <string_symbol id="event_b"            string="b"/>

 <!-- rules of the Parser
  note, empty rule_rpop and rule_rpush components are omitted -->
 <parser_step_label id="rule_0"
  rule_lpop="nonterminal_0"
  rule_lpush="nonterminal_box nonterminal_0"/>
 <parser_step_label id="rule_1"
  rule_lpop="nonterminal_0" rule_rpop="event_a"
  rule_lpush="nonterminal_bullet nonterminal_1"/>
 <parser_step_label id="rule_2"
  rule_lpop="nonterminal_1" rule_rpop="event_a"
  rule_lpush="nonterminal_bullet nonterminal_1"/>
 <parser_step_label id="rule_3"
  rule_lpop="nonterminal_bullet nonterminal_1" rule_rpop="event_b"
  rule_lpush="nonterminal_2"/>
 <parser_step_label id="rule_4"
  rule_lpop="nonterminal_bullet nonterminal_2" rule_rpop="event_b"
  rule_lpush="nonterminal_2"/>
 <parser_step_label id="rule_5"
  rule_lpop="nonterminal_box nonterminal_2"
  rule_lpush="nonterminal_3"/>

 <!-- the Parser with its components -->
 <parser
   id                = "parser__a2n_b2n"
   parser_nonterms   = "nonterminal_0 nonterminal_1 nonterminal_2
                        nonterminal_3 nonterminal_box
                        nonterminal_bullet"
   parser_events     = "event_a event_b event_END"
   parser_initial    = "nonterminal_0"
   parser_marking    = "nonterminal_0 nonterminal_3"
   parser_step_labels = "rule_0 rule_1 rule_2 rule_3 rule_4 rule_5"
   parser_eoi        = "event_END"/>
</root>
```

A CFG in XML:

```
<?xml version="1.0"?>
<root
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="../../../main/resources/cosy.xsd">

 <!-- nonterminals and events used in the CFG -->
 <string_symbol id="nonterminal_0" string="S"/>
 <string_symbol id="event_a"       string="a"/>
 <string_symbol id="event_b"       string="b"/>

 <!-- wrapped nonterminals and events used in the right hand side of
     productions below -->
 <two_elements__teA id="beA_nonterminal_0" value="nonterminal_0"/>
 <two_elements__teB id="beB_event_a"       value="event_a"/>
 <two_elements__teB id="beB_event_b"       value="event_b"/>

 <!-- productions of the CFG
  note, empty prod_rhs components are omitted -->
 <cfg_step_label id="prod_0"
  prod_lhs="nonterminal_0"
  prod_rhs="beB_event_a beA_nonterminal_0 beB_event_b"/>
 <cfg_step_label id="prod_1"
  prod_lhs="nonterminal_0"/>

 <!-- the CFG with its components -->
 <cfg
   id               = "cfg__a2n_b2n"
   cfg_nonterminals = "nonterminal_0"
   cfg_events       = "event_a event_b"
   cfg_initial      = "nonterminal_0"
   cfg_step_labels  = "prod_0 prod_1"/>
</root>
```

4

First Concrete Controller Synthesis Algorithm

In Figure 2.2|p.4, we stated that *Algorithm_Definition__Construct_Controller.java* is the main class for our first concrete controller synthesis algorithm. We omit subsequently type parameters of most generic types and some keywords. The concrete controller synthesis algorithm is applied using the method

```
option__abstract<epda>
  F_DPDA_DFA_CC
  (epda S, epda P, List SigmaUC)
```

This method computes the initial controller candidate using the method

```
option__abstract<epda>
  F_DPDA_DFA_CC__fp_start
  (epda S, epda P)
```

and then obtains the resulting controller using the method

```
option__abstract<epda>
  F_DPDA_DFA_CC__fp
  (epda C, epda P, List SigmaUC)
```

This method applies the method

```
tuple2<option__abstract<epda>, Boolean>
  F_DPDA_DFA_CC__fp_one
  (epda C, epda P, List SigmaUC)
```

on the current controller candidate until a fixed point is obtained. The boolean in the return type is used to state whether the controller candidate C has been changed in the last application of *F_DPDA_DFA_CC__fp_one*.

These methods implement the corresponding construction from our Isabelle-based formalization. Moreover, *F_DPDA_DFA_CC__fp_start* and *F_DPDA_DFA_-CC__fp_one* rely on further methods for enforcing nonblockingness,

```
option__abstract<epda>
  Algorithm_Definition__Enforce_Blockfree.F_DPDA_EB_OPT
  (epda C)
```

for constructing the product automaton of a DPDA and a DFA,

```
epda
  Algorithm_Definition__DPDA_DFA_PRODUCT.F_DPDA_DFA_PRODUCT
  (epda S, epda P)
```

and for reducing controllability to nonblockingness.

```
tuple2<option__abstract<epda>, Boolean>
  Algorithm_Definition__Enforce_Controllable.F_DPDA_EC
  (epda C, epda P, List SigmaUC)
```

These methods are given in the three packages representing the major building blocks of our first concrete controller synthesis algorithm (see Figure 2.2|p.4).

With this discussion of the outermost methods of our first concrete controller synthesis algorithm, the interested reader should now be capable to inspect the mentioned further methods and classes to understand how these methods in turn rely on further methods from the respective major building blocks.

5

Unit Tests

In Figure 2.2|p.4, we specified the directory with the Java classes and XML files for our unit tests. We now describe the typical pattern of the unit tests in this directory. As a first step, these unit tests load some instances of DPDA, CFGs, and Parsers from an XML file specific to the unit test. As a second step, they apply one or more construction of our concrete controller synthesis algorithm and check after each application whether the marked or unmarked languages have been preserved between input and output. Also, many unit tests check whether the resulting EPDA or Parser is still deterministic. Finally, as a last step, we check the size of the resulting EPDA, CFGs, and Parsers.

For comparing the marked/unmarked language of input and output we use the package *language_check* from Figure 2.2|p.4. Using the contained classes and methods, together with the semantics of the formalisms, we generate randomly/in breadth first manner words of events from the (un)marked language of the input and output and then check for each of these words whether they can also be derived such that they are in the (un)marked language of the other instance.

For the generation of marked/unmarked words, we require the semantics to define the step-relation, the initial configuration, and methods for obtaining the current history from a configuration and for checking whether the configuration is a marking configuration.

6

Multithreading Support

In Figure 2.2|p.4 we specified the directory with the Java classes for our multithreading support, which is considered in the following figure in more detail.

Figure 6.1: «Classes for Multithreading Support»

```
multithreading
└─ general
   ├─ Local_Solver.java ..... central management component for threads
   ├─ Runnable_Solver.java ..... class for threads that process problems
   └─ abstract_components
      ├─ Abstract_Problem.java ..... a class for storing a single problem
      ├─ Abstract_Solution.java ..... a class for storing computed results
      ├─ Abstract_Solver.java ..... a class with the solving code
      └─ Callback.java ..... an interface for callbacks
```

The class *Local_Solver* is the central management component where threads, priorities, lists of open problems awaiting processing, and mappings from problems to solvers are stored. Also, the selection of the number of threads to be used as well as the shutdown of multithreading support is given in this class.

The number of threads to be used depends on the hardware architecture, we impose no restrictions but the automatic memory management of Java may not be ideal to support many threads and using more threads than executable in parallel is often not helpful. Each worker thread is an instance of the class *Runnable_Solver*.

For applications of this framework, each class in *abstract_components* must be implemented (possibly using multiple callbacks). Firstly, *Abstract_Problem* is for storing the arguments required for the computation to be executed. Secondly, *Abstract_Solution* is for storing the results of computation. Lastly, *Abstract_Solver*

contains a method solving a given *Abstract_Problem* resulting in an *Abstract_-Solution*. Implementations of the *Callback* interface allow for the handling of computed results at the end (when the execution is started by a thread and when a parallel task has been completed). The main purpose is that task is split into multiple problems that are then solved in parallel. The results are collected using the callbacks and once all problems have been solved and all results have been collected, further processing can be continued.

The package *multithreading* contains some packages for certain tasks supported by multithreading in the described sense. It is to be noted that the problems should require a reasonable amount of time to limit the multithreading overhead (memory management and result composition). However, too big chunks can decrease performance when not all worker threads are used at the end of the parallel computation.

The package *T03_02_EPDA_APPROXIMATE_INITIAL_ACCESSIBLE* is a simple first example where the state space of a EPDA is approximated with configurations with finite stack depth. A problem consists of the EPDA, a list of approximate configurations to be considered, and the maximal depth k of the stack. The solution is a resulting list of approximate configurations obtained. The solver checks each provided approximate configuration and derives a set of approximate configurations for each of them. The results for one problem and, moreover, the results of parallel threads are then merged (and duplicates are removed) before the remaining new approximate configurations are again split among the worker threads in the next iteration.

7

Execution of Benchmarks

In Figure 2.1|p.3, we specified the directory with scripts used for benchmarking and also containing the results of these benchmark executions.

Figure 7.1: «Files for Benchmarking»

```
benchmarking
├── results
│   ├── fb14srv5 ..................... directory with each test-run result
│   ├── results_fb14srv5.txt ......... aggregated results (averages etc.)
│   ├── fb14srv6 ..................................... as for the other server
│   └── results_fb14srv6.txt ......................... as for the other server
└── scripts
    ├── copy_to_remote.sh ....... script for copying jar etc.  to the servers
    ├── run_benchmarks.sh .... script executing CoSy with selected parameters
    └── copy_from_remote.sh ..... script for copying results from the servers
```

We have considered three examples from manufacturing our thesis. We split the last of these examples into three different versions requiring different computation times and memory leading altogether to five examples. We evaluated the efficiency of our prototype by applying it to these examples while measuring the required time and memory. For this purpose we use Maven to package CoSy into a jar file, use *copy_to_remote.sh* to copy this jar file with the required XML files for the benchmarks to remote servers where the benchmarks are to be executed, use *run_benchmarks.sh* to execute the benchmarks that are given in this script as well with their individual parameters, and use *copy_from_remote.sh* to retrieve the results of executed benchmarks from the servers. The benchmark results are stored in the *results/{server name}* directory and multiple runs for one parameter configuration (we used 10 executions for each set of parameters) are aggregated

by computing average runtime and memory consumption resulting in an entry in *results/results_{server name}.txt*.

We have added all benchmark results in Appendix A|p.25 but omitted most of them in our thesis as they provide not additional insights.

8

Second Concrete Controller Synthesis Algorithm (LR(1)-CFG)

In Figure 2.2|p.4, we stated that *Algorithm_Definition__CFG_Construct_Controller.java* is the main class for our second concrete controller synthesis algorithm. A test case for this concrete controller synthesis algorithm is given in *po5__CFG_CONSTRUCT_CONTROLLER__Algorithm_Definition__CFG_Construct_Controller* that makes use of the example presented in the thesis.

We omit subsequently type parameters of most generic types and some keywords. The concrete controller synthesis algorithm is applied using the method

```
option__abstract<epda>
  F_UNVERIFIED_CFG_DPDA_DFA_CC
  (epda S, epda P, List SigmaUC)
```

This method computes the initial controller candidate (and a map that contains for each plant state the set of events that can be executed from this plant state) using the method

```
option__abstract<tuple2<cfg,Map<plant-state,Set<event>>>>
  F_UNVERIFIED_DPDA_DFA_CC__fp_start
  (epda S, epda P)
```

and then obtains the resulting controller using the method

```
option__abstract<cfg>
  F_UNVERIFIED_CFG_DPDA_DFA_CC__fp
  (cfg C, epda P, List SigmaUC, Map<plant-state,Set<event>>
      plant_state_to_events)
```

This method applies the method

```
tuple2<option__abstract<cfg>, Boolean>
  F_UNVERIFIED_CFG_DPDA_DFA_CC__fp_one
  (cfg C, epda P, List SigmaUC, Map<plant-state,Set<event>>
      plant_state_to_events)
```

on the current controller candidate until a fixed point is obtained. The boolean in the return type is used to state whether the controller candidate C has been changed in the last application of *F_UNVERIFIED_DPDA_DFA_CC__fp_one*.

These methods have not been formalized or verified in Isabelle. The method *F_UNVERIFIED_DPDA_DFA_CC__fp_start* converts the initial controller candidate, which is computed using

```
epda
  Algorithm_Definition__DPDA_DFA_PRODUCT.F_DPDA_DFA_PRODUCT
  (epda S, epda P)
```

as the synchronous product of plant P and specification S, into an LR(1)-CFG, which satisfies the nonblockingness property. It then applies the method

```
option__abstract<tuple2<cfg,Map<plant-state,Set<event>>>>
  F_UNVERIFIED_CFG_DPDA_DFA_CC__fp_start2
  (cfg C, epda P, List SigmaUC)
```

for disambiguating the LR(1)-CFG using the method *FUN_CFG_PULL* discussed in our thesis, for the computation of the nonterminals with potential controllability problems and their parents, and for disambiguating a second time based on these parents using *split_cfg_weak*.

In *F_UNVERIFIED_DPDA_DFA_CC__fp_one* we apply the method

```
tuple2<option__abstract<cfg>, Boolean>
  F_UNVERIFIED_CFG_DFA_EC
  (cfg C, Map<plant-state,Set<event>> plant_state_to_events)
```

for enforcing controllability on the LR(1)-CFG controller candidate. If the controller candidate has been changed in this application, we trim the obtained LR(1)-CFG controller candidate to enforce the satisfaction of the property of nonblockingness.

The method *F_UNVERIFIED_CFG_DFA_EC* computes the LR(1)-Machine for the LR(1)-CFG C, identifies certain states of this LR(1)-Machine (that indicate controllability problems) and then removes the corresponding nonterminals from C.

A more in depth discussion of the constructions is given in our thesis.

9

Future Work

There are some aspects of the prototype that need to be worked on in the future.

- Further operations may benefit from using multithreading and operations such as the construction of the LR(1)-Machine that have multithreading support already may be improved in this aspect.

- We should attempt to avoid recalculation for unchanged inputs of methods throughout the algorithm using caches and, moreover, should attempt to focus controller synthesis (that is, the restriction of the given DPDA controller candidate) in later iterations to the parts that are potentially to be adapted (that is, only prefixes of words with controllability problems in iteration i can have a controllability problem in iteration $j > i$ and, hence, states and edges of a DPDA controller candidate that are not on a path from the initial state to a state with a controllability problem do not need to be inspected further).

- More unit tests are required for some operations. In particular, the methods for reducing controllability of DPDA to nonblockingness has been covered in our benchmarks as integration tests but unit tests are missing as of now. Other operations that need further testing (and eventually verification) are the constructions mentioned in the ongoing work section of our thesis. These are the recursive construction of the LR(1)-CFG (in the java class *ADDITIONAL_OPERATIONS.T10_SDPDA_TO_LR_OPT2.java*), the reduction of controllability to nonblockingness for LR(1)-CFG (in the java class *Algorithm_Definition__CFG_Construct_Controller.java*, and the analysis of how worst case execution times can be established via annotations for the runtime environment of the controller.

- The prototype supports the three use cases of controller construction and controller verification, but the validation of the inputs (by means of outputs that are instructive and diverse to detect modeling errors in the plant and specification to be computed upfront or during controller synthesis) is missing as of now.
- Finally, various enhancements and optimizations of the prototype are discussed in the relevant chapter of our thesis such as for the reduction of the size of the resulting controller.

However, as pointed out in our thesis as well, we will implement our algorithms in C++ by adapting the libFAUDES [2] plugin developed earlier because our benchmarking-based evaluation 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.

10

Conclusion

We discussed several basic aspects of our prototype CoSy in this documentation to guide potential users and developers. In conclusion, the prototype is reasonably efficient for a Java based implementation but a C++ implementation may be more efficient in the future. Then, both implementation can be compared using back-to-back testing. To the best of our knowledge, the prototype is fault-free, but more testing is required. However, improvements will also be related to further enhancements and improvements of the concrete controller synthesis algorithm implemented. Initial steps in this direction have been taken in alternative procedures (exchanging entire building blocks) and more optimal methods (replacing single building blocks). These modifications of our formally verified concrete controller synthesis algorithm should be verified in the future as well to ensure trustworthiness.

A

Benchmark Results

In the following figure we list all benchmarks that have been executed on the two servers fb14srv6 and fb14srv5 described in more detail in our thesis. We first start with a description of the format used and then continue with the two listing for the two servers where we present the successful benchmarks (where all 10 executions succeeded) for the five different considered examples.

Figure A.1: «Benchmark Results on fb14srv6 and fb14srv5»

Format: Each entry consists of 8 fields.

1. *A* represents fb14srv6 and *B* represents fb14srv5
2. *true* if the controller has been synthesized and *false* if only build up and tear down have been executed as a dry run
3. The maximal depth of a stack in approximate configurations when approximating the state space to identify inaccessible states/edges in an EPDA
4. The maximal (heap) memory size in GiB to be used by the Java runtime environment; specified using the Java parameter $-Xmx$
5. The actually used total memory in GiB of the process (which may exceed the previous limitation in theory).
6. The number of minutes required by the process.
7. The number of seconds required by the process.
8. The percentage of cpu the process got (it is often above 100% because we employ multithreading)

The number 10 at the end of the line indicates that all 10 runs succeeded.

Results on fb14srv6:

```
automated_fabrication_scenario_A
\BENCHMARKentry{A}{false}{0}{230}{ 0}{ 0}{ 1}{279}% 10
\BENCHMARKentry{A}{ true}{1}{128}{ 4}{ 0}{15}{362}% 10
\BENCHMARKentry{A}{ true}{1}{ 16}{ 2}{ 0}{13}{386}% 10
\BENCHMARKentry{A}{ true}{1}{  1}{ 1}{ 0}{12}{433}% 10
\BENCHMARKentry{A}{ true}{1}{230}{ 4}{ 0}{14}{364}% 10
\BENCHMARKentry{A}{ true}{1}{  2}{ 1}{ 0}{12}{417}% 10
\BENCHMARKentry{A}{ true}{1}{ 32}{ 4}{ 0}{14}{363}% 10
\BENCHMARKentry{A}{ true}{1}{  3}{ 1}{ 0}{16}{330}% 10
\BENCHMARKentry{A}{ true}{1}{  4}{ 2}{ 0}{12}{403}% 10
\BENCHMARKentry{A}{ true}{1}{  5}{ 2}{ 0}{16}{314}% 10
\BENCHMARKentry{A}{ true}{1}{ 64}{ 4}{ 0}{15}{351}% 10
\BENCHMARKentry{A}{ true}{1}{  8}{ 2}{ 0}{13}{383}% 10
\BENCHMARKentry{A}{ true}{2}{128}{ 4}{ 0}{15}{369}% 10
\BENCHMARKentry{A}{ true}{2}{ 16}{ 3}{ 0}{13}{393}% 10
\BENCHMARKentry{A}{ true}{2}{  1}{ 1}{ 0}{13}{442}% 10
\BENCHMARKentry{A}{ true}{2}{230}{ 4}{ 0}{15}{364}% 10
\BENCHMARKentry{A}{ true}{2}{  2}{ 1}{ 0}{13}{422}% 10
\BENCHMARKentry{A}{ true}{2}{ 32}{ 4}{ 0}{15}{367}% 10
\BENCHMARKentry{A}{ true}{2}{  3}{ 1}{ 0}{16}{328}% 10
\BENCHMARKentry{A}{ true}{2}{  4}{ 2}{ 0}{13}{403}% 10
\BENCHMARKentry{A}{ true}{2}{  5}{ 2}{ 0}{16}{320}% 10
\BENCHMARKentry{A}{ true}{2}{ 64}{ 4}{ 0}{15}{358}% 10
\BENCHMARKentry{A}{ true}{2}{  8}{ 3}{ 0}{13}{393}% 10
\BENCHMARKentry{A}{ true}{3}{230}{ 4}{ 0}{20}{289}% 10
\BENCHMARKentry{A}{ true}{3}{  3}{ 1}{ 0}{16}{339}% 10
\BENCHMARKentry{A}{ true}{3}{  5}{ 2}{ 0}{17}{324}% 10
automated_fabrication_scenario_B
\BENCHMARKentry{A}{ true}{0}{128}{ 3}{ 0}{15}{346}% 10
\BENCHMARKentry{A}{ true}{0}{ 16}{ 3}{ 0}{13}{366}% 10
\BENCHMARKentry{A}{ true}{0}{  1}{ 1}{ 0}{14}{486}% 10
\BENCHMARKentry{A}{false}{0}{230}{ 0}{ 0}{ 1}{291}% 10
\BENCHMARKentry{A}{ true}{0}{230}{ 3}{ 0}{15}{339}% 10
\BENCHMARKentry{A}{ true}{0}{  2}{ 1}{ 0}{13}{436}% 10
\BENCHMARKentry{A}{ true}{0}{ 32}{ 3}{ 0}{15}{342}% 10
\BENCHMARKentry{A}{ true}{0}{  3}{ 2}{ 0}{22}{450}% 10
\BENCHMARKentry{A}{ true}{0}{  4}{ 2}{ 0}{13}{392}% 10
\BENCHMARKentry{A}{ true}{0}{ 64}{ 3}{ 0}{15}{337}% 10
\BENCHMARKentry{A}{ true}{0}{  7}{ 3}{ 0}{20}{341}% 10
\BENCHMARKentry{A}{ true}{0}{  8}{ 3}{ 0}{13}{364}% 10
\BENCHMARKentry{A}{ true}{1}{128}{ 0}{ 0}{ 1}{469}% 10
\BENCHMARKentry{A}{ true}{1}{ 16}{ 0}{ 0}{ 1}{467}% 10
\BENCHMARKentry{A}{ true}{1}{  1}{ 0}{ 0}{ 1}{499}% 10
\BENCHMARKentry{A}{ true}{1}{230}{ 0}{ 0}{ 1}{455}% 10
\BENCHMARKentry{A}{ true}{1}{  2}{ 0}{ 0}{ 1}{479}% 10
```

```
\BENCHMARKentry{A}{ true}{1}{ 32}{ 0}{ 0}{ 1}{457}% 10
\BENCHMARKentry{A}{ true}{1}{  4}{ 0}{ 0}{ 1}{478}% 10
\BENCHMARKentry{A}{ true}{1}{ 64}{ 0}{ 0}{ 1}{459}% 10
\BENCHMARKentry{A}{ true}{1}{  8}{ 0}{ 0}{ 1}{470}% 10
\BENCHMARKentry{A}{ true}{2}{128}{ 0}{ 0}{ 1}{476}% 10
\BENCHMARKentry{A}{ true}{2}{ 16}{ 0}{ 0}{ 1}{501}% 10
\BENCHMARKentry{A}{ true}{2}{  1}{ 0}{ 0}{ 1}{492}% 10
\BENCHMARKentry{A}{ true}{2}{230}{ 0}{ 0}{ 1}{485}% 10
\BENCHMARKentry{A}{ true}{2}{  2}{ 0}{ 0}{ 1}{494}% 10
\BENCHMARKentry{A}{ true}{2}{ 32}{ 0}{ 0}{ 1}{477}% 10
\BENCHMARKentry{A}{ true}{2}{  4}{ 0}{ 0}{ 1}{491}% 10
\BENCHMARKentry{A}{ true}{2}{ 64}{ 0}{ 0}{ 1}{487}% 10
\BENCHMARKentry{A}{ true}{2}{  8}{ 0}{ 0}{ 1}{497}% 10
automated_fabrication_scenario_CV3
\BENCHMARKentry{A}{false}{0}{230}{ 0}{ 0}{ 1}{297}% 10
\BENCHMARKentry{A}{ true}{1}{128}{45}{ 4}{51}{549}% 10
\BENCHMARKentry{A}{ true}{1}{ 16}{ 7}{ 3}{44}{588}% 10
\BENCHMARKentry{A}{ true}{1}{230}{73}{ 4}{57}{537}% 10
\BENCHMARKentry{A}{ true}{1}{ 32}{13}{ 4}{23}{609}% 10
\BENCHMARKentry{A}{ true}{1}{  4}{ 4}{ 3}{37}{616}% 10
\BENCHMARKentry{A}{ true}{1}{ 64}{24}{ 4}{39}{574}% 10
\BENCHMARKentry{A}{ true}{1}{  8}{ 5}{ 3}{37}{612}% 10
automated_fabrication_scenario_CV2
\BENCHMARKentry{A}{false}{0}{230}{ 0}{ 0}{ 1}{337}% 10
\BENCHMARKentry{A}{ true}{1}{128}{133}{767}{ 2}{982}% 10
\BENCHMARKentry{A}{false}{1}{230}{ 0}{ 0}{ 1}{359}% 10
\BENCHMARKentry{A}{ true}{1}{230}{198}{751}{30}{949}% 10
\BENCHMARKentry{A}{ true}{2}{230}{174}{928}{27}{410}% 10
\BENCHMARKentry{A}{ true}{3}{230}{182}{933}{15}{409}% 10
automated_fabrication_scenario_CV1
```

Results on fb14srv5:

```
automated_fabrication_scenario_A
\BENCHMARKentry{B}{false}{0}{355}{ 0}{ 0}{ 1}{248}% 10
\BENCHMARKentry{B}{ true}{1}{128}{ 4}{ 0}{18}{314}% 10
\BENCHMARKentry{B}{ true}{1}{ 16}{ 2}{ 0}{16}{332}% 10
\BENCHMARKentry{B}{ true}{1}{  1}{ 1}{ 0}{14}{381}% 10
\BENCHMARKentry{B}{ true}{1}{256}{ 4}{ 0}{17}{320}% 10
\BENCHMARKentry{B}{ true}{1}{  2}{ 1}{ 0}{14}{358}% 10
\BENCHMARKentry{B}{ true}{1}{ 32}{ 4}{ 0}{18}{312}% 10
\BENCHMARKentry{B}{ true}{1}{355}{ 4}{ 0}{17}{317}% 10
\BENCHMARKentry{B}{ true}{1}{  3}{ 1}{ 0}{21}{284}% 10
\BENCHMARKentry{B}{ true}{1}{  4}{ 2}{ 0}{15}{334}% 10
\BENCHMARKentry{B}{ true}{1}{ 64}{ 4}{ 0}{18}{315}% 10
\BENCHMARKentry{B}{ true}{1}{  6}{ 2}{ 0}{21}{273}% 10
\BENCHMARKentry{B}{ true}{1}{  8}{ 2}{ 0}{16}{337}% 10
```

```
\BENCHMARKentry{B}{ true}{2}{128}{ 4}{ 0}{19}{322}% 10
\BENCHMARKentry{B}{ true}{2}{ 16}{ 3}{ 0}{16}{338}% 10
\BENCHMARKentry{B}{ true}{2}{  1}{ 1}{ 0}{15}{390}% 10
\BENCHMARKentry{B}{ true}{2}{256}{ 4}{ 0}{19}{335}% 10
\BENCHMARKentry{B}{ true}{2}{  2}{ 1}{ 0}{15}{372}% 10
\BENCHMARKentry{B}{ true}{2}{ 32}{ 4}{ 0}{19}{321}% 10
\BENCHMARKentry{B}{ true}{2}{355}{ 4}{ 0}{18}{336}% 10
\BENCHMARKentry{B}{ true}{2}{  3}{ 1}{ 0}{20}{291}% 10
\BENCHMARKentry{B}{ true}{2}{  4}{ 2}{ 0}{16}{352}% 10
\BENCHMARKentry{B}{ true}{2}{ 64}{ 4}{ 0}{19}{323}% 10
\BENCHMARKentry{B}{ true}{2}{  6}{ 2}{ 0}{21}{278}% 10
\BENCHMARKentry{B}{ true}{2}{  8}{ 3}{ 0}{17}{335}% 10
\BENCHMARKentry{B}{ true}{3}{355}{ 4}{ 0}{24}{264}% 10
\BENCHMARKentry{B}{ true}{3}{  3}{ 1}{ 0}{20}{296}% 10
\BENCHMARKentry{B}{ true}{3}{  5}{ 2}{ 0}{21}{289}% 10
automated_fabrication_scenario_B
\BENCHMARKentry{B}{ true}{0}{128}{ 3}{ 0}{17}{339}% 10
\BENCHMARKentry{B}{ true}{0}{ 16}{ 3}{ 0}{16}{353}% 10
\BENCHMARKentry{B}{ true}{0}{  1}{ 1}{ 0}{16}{451}% 10
\BENCHMARKentry{B}{ true}{0}{256}{ 3}{ 0}{16}{327}% 10
\BENCHMARKentry{B}{ true}{0}{  2}{ 1}{ 0}{15}{405}% 10
\BENCHMARKentry{B}{ true}{0}{ 32}{ 3}{ 0}{17}{337}% 10
\BENCHMARKentry{B}{false}{0}{355}{ 0}{ 0}{ 1}{258}% 10
\BENCHMARKentry{B}{ true}{0}{355}{ 3}{ 0}{16}{332}% 10
\BENCHMARKentry{B}{ true}{0}{  3}{ 2}{ 0}{26}{405}% 10
\BENCHMARKentry{B}{ true}{0}{  4}{ 2}{ 0}{16}{382}% 10
\BENCHMARKentry{B}{ true}{0}{ 64}{ 3}{ 0}{17}{333}% 10
\BENCHMARKentry{B}{ true}{0}{  7}{ 3}{ 0}{25}{331}% 10
\BENCHMARKentry{B}{ true}{0}{  8}{ 3}{ 0}{16}{357}% 10
\BENCHMARKentry{B}{ true}{1}{128}{ 0}{ 0}{ 2}{418}% 10
\BENCHMARKentry{B}{ true}{1}{ 16}{ 0}{ 0}{ 2}{444}% 10
\BENCHMARKentry{B}{ true}{1}{  1}{ 0}{ 0}{ 2}{439}% 10
\BENCHMARKentry{B}{ true}{1}{256}{ 0}{ 0}{ 2}{423}% 10
\BENCHMARKentry{B}{ true}{1}{  2}{ 0}{ 0}{ 2}{440}% 10
\BENCHMARKentry{B}{ true}{1}{ 32}{ 0}{ 0}{ 2}{412}% 10
\BENCHMARKentry{B}{ true}{1}{355}{ 0}{ 0}{ 2}{430}% 10
\BENCHMARKentry{B}{ true}{1}{  4}{ 0}{ 0}{ 2}{433}% 10
\BENCHMARKentry{B}{ true}{1}{ 64}{ 0}{ 0}{ 2}{412}% 10
\BENCHMARKentry{B}{ true}{1}{  8}{ 0}{ 0}{ 2}{438}% 10
\BENCHMARKentry{B}{ true}{2}{128}{ 0}{ 0}{ 2}{441}% 10
\BENCHMARKentry{B}{ true}{2}{ 16}{ 0}{ 0}{ 2}{452}% 10
\BENCHMARKentry{B}{ true}{2}{  1}{ 0}{ 0}{ 2}{454}% 10
\BENCHMARKentry{B}{ true}{2}{256}{ 0}{ 0}{ 2}{429}% 10
\BENCHMARKentry{B}{ true}{2}{  2}{ 0}{ 0}{ 2}{456}% 10
\BENCHMARKentry{B}{ true}{2}{ 32}{ 0}{ 0}{ 2}{437}% 10
\BENCHMARKentry{B}{ true}{2}{355}{ 0}{ 0}{ 2}{445}% 10
```

```
\BENCHMARKentry{B}{ true}{2}{  4}{ 0}{ 0}{ 2}{455}% 10
\BENCHMARKentry{B}{ true}{2}{ 64}{ 0}{ 0}{ 2}{440}% 10
\BENCHMARKentry{B}{ true}{2}{  8}{ 0}{ 0}{ 2}{455}% 10
automated_fabrication_scenario_CV3
\BENCHMARKentry{B}{false}{0}{355}{ 0}{ 0}{ 1}{267}% 10
\BENCHMARKentry{B}{ true}{1}{128}{45}{ 6}{20}{496}% 10
\BENCHMARKentry{B}{ true}{1}{ 16}{ 7}{ 5}{ 2}{566}% 10
\BENCHMARKentry{B}{ true}{1}{256}{84}{ 6}{36}{489}% 10
\BENCHMARKentry{B}{ true}{1}{ 32}{13}{ 5}{40}{558}% 10
\BENCHMARKentry{B}{ true}{1}{355}{94}{ 6}{27}{487}% 10
\BENCHMARKentry{B}{ true}{1}{  4}{ 4}{ 4}{51}{588}% 10
\BENCHMARKentry{B}{ true}{1}{ 64}{24}{ 5}{58}{529}% 10
\BENCHMARKentry{B}{ true}{1}{  8}{ 5}{ 4}{54}{583}% 10
automated_fabrication_scenario_CV2
\BENCHMARKentry{B}{false}{0}{355}{ 0}{ 0}{ 1}{303}% 10
\BENCHMARKentry{B}{ true}{1}{128}{133}{950}{ 5}{920}% 10
\BENCHMARKentry{B}{ true}{1}{256}{202}{921}{12}{895}% 10
\BENCHMARKentry{B}{false}{1}{355}{ 0}{ 0}{ 1}{318}% 10
\BENCHMARKentry{B}{ true}{1}{355}{225}{922}{37}{888}% 10
\BENCHMARKentry{B}{ true}{2}{355}{222}{1249}{44}{439}% 10
\BENCHMARKentry{B}{ true}{3}{355}{225}{1259}{36}{444}% 10
automated_fabrication_scenario_CV1
```

These benchmarks show that the largest benchmark does not terminate in 24 h and that different maximal memory values also impact the runtime of the concrete controller synthesis algorithm.

B

Bibliography

[1] Apache logging services. *Apache Log4j2*. 2018. URL: https://logging.apache.org/log4j/2.x/ (cit. on p. 3).

[2] 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 Pelaic, 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 p. 22).

[3] John Ellson and Emden Gansner. *Graphviz—Graph Visualization Software*. 2018. URL: https://www.graphviz.org/ (cit. on p. 7).

[4] 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).

[5] The Apache Software Foundation. *Apache Maven Project*. 2018. URL: https://maven.apache.org/ (cit. on p. 3).

[6] W3C. *W3C XML Schema Definition Language (XSD) 1.1*. 2018. URL: https://www.w3.org/TR/xmlschema11-1/ (cit. on p. 3).