Test Automation

Foundations and Applications of Model-based Testing

Lecture Notes

Jan Peleska and Wen-ling Huang
{peleska,huang}@uni-bremen.de

Issue 5.2
2021-11-03

Note. These lecture notes are still under development, so it is advisable to check for updates.

Change History

Issue	Date	Change description
5.2	2021-11-03	Add Lemma 3.2 about characterisation set. Change bars mark the differences with respect to Issue 5.0.
5.1	2021-02-09	Small fixes in Chapter 12 regarding compile options. New chapter 14 on code coverage analysis to support fuzz testing. Change bars mark the differences with respect to Issue 5.0.
5.0	2021-02-01	Preface extended regarding the scope of these lecture notes. More on test objectives and test types in Part I. New Part IV on fuzz testing. Change bars mark the differences with respect to Issue 4.0.
4.0	2019-07-01	Missing primes added in formula for γ in Section 7.7.4. New section on state counting principle.
3.9	2019-06-24	Fixed typo in Definition 7.2.
3.8	2019-05-27	Small improvements in the H-Method proof and related definitions.
3.7	2019-05-20	Removed exercise on W-method implementation in Java (Section 4.6). New Section 4.7 presenting the H-Method.
3.6	2019-05-06	Fixed proof of Theorem 4.4. Changed pre-requisites of Theorem 4.5. Added discussion of minimisation.
3.5	2019-04-29	New Section 4.4 on test oracles and complete testing assumption. Revised Section 4.5 on product complete testing theory based on product automata construction.
3.4	2019-04-12	Updated description of fsm-generator in Section B.3.
3.3	2019-04-08	New version of Exercise 2. This now requires to specify the input equivalence classes explicitly, and to choose random values from each class.

3.2	2017-04-26	Fixed error in the description of the P_k-table algorithm, Part B in Section 3.3.2. First version of the new Appendix B introducing the FSM Library. Preface extended w.r.t. Appendix B. Since Appendix B is new, it has not been marked by change bars. Otherwise, change bars mark the differences with respect to Issue 3.1.
3.1	2017-02-01	(1) Theorem 4.9 added showing that the Wp-Method never produces more test cases than the W-Method. (2) Removed the preliminary section 4.6.2 on FSM testing for reduction. This section will be included in a later revision. (3) Exchanged the term *propositional logic* against *first order logic* where appropriate. (4) Updated versions of chapters 8, 9, and 10, based on final version of [32]. (5) Improved statement and proof of Theorem 4.1. Change bars mark the differences with respect to Issue 2.11.
2.11	2017-01-09	(1) New Exercise. (2) Small improvements in Chapter 5. (3) Removed definition of synchronous product from Chapter 6, since this is no longer needed. (3) Small improvements in Chapter 7. Change bars mark the differences with respect to Issue 2.10.
2.10	2016-11-14	New proof using abstract test cases for Theorem 4.3 (completeness of the T-Method). Simplified proof of Lemma 4.2. Added explanation about test cases to Theorem. Change bars mark the differences with respect to Issue 2.9.
2.9	2016-11-02	New Exercise 4 in Section 4.3. Change bars mark the differences with respect to Issue 2.8.
2.8	2016-10-26	Re-worked the CSM description in Exercise 1. New Exercise 2 in Section 3.2. Change bars mark the differences with respect to Issue 2.7.
2.7	2016-10-24	Added abstract notion of FSM test cases in Chapter 4. Change bars mark the differences with respect to Issue 2.6.
2.6	2016-10-19	Added first exercise in Chapter 1 Change bars mark the differences with respect to Issue 2.5.
2.5	2016-10-17	(1) New Chapter 1 on basic definitions related to testing. (2) Change of test case definition in Section 2.4. Change bars mark the differences with respect to Issue 2.4.
2.4	2016-08-04	First version made available on the web. Copyright notice inserted. Otherwise no changes with respect to Issue 2.3.

2.3	2016-04-21	Fixed error in formula and improved presentation of IECP calculation in Section 7.7. Change bars mark the differences with respect to Issue 2.2.
2.2	2016-03-15	Updates in Section 7.7.6: the representation of Φ_f described in Issue 2.1 is only valid for deterministic RIOSTS. The new version is adequate for the nondeterministic case. Change bars mark the differences with respect to Issue 2.1.
2.1	2016-02-02	Final manuscript version for winter semester 2015/2016. (0) Completed Appendix A (1) Moved appendix behind bibliography. (3) Fixes in the table of Example 11. (4) Fixed $g_{0,2}$ in Example 14. (5) Use notation η_i instead if β_i in Examples 10 and 11, because β_i is used later on with another meaning. (6) Added a preface. Change bars mark the differences with respect to Issue 2.0.
2.0	2016-01-31	Pre-final manuscript version for winter semester 2015/2016. (1) Added example showing the tree generated by the minimal hitting set algorithm introduced in Appendix A. (2) New Section 7.7 describing an algorithm for constructing input equivalence classes and the model map from RIOSTS to FSMs. (3) Added example showing that the Wp-Method cannot be used for proving reduction added in Section4.8.1. (4) Explanation for Step 9 of the Wp-Method (calculation of W_p) added. Change bars mark the differences with respect to Issue 1.9.
1.9	2016-01-26	(1) Improvements in Section 3.7 on characterisation set and state identification sets for NFSMs. Change bars mark the differences with respect to Issue 1.8.
1.8	2016-01-25	(0) Section on DFSM characterisation sets moved to Section 3.4 in introductory FSM chapter. (1) New Section 3.7 on characterisation set and state identification sets for NFSMs. (2) New Appendix A describing simple algorithm for calculation of minimal hitting sets (needed for calculation of state identification sets). (3) Extended discussion about testing deterministic implementations against nondeterministic models. Change bars mark the differences with respect to Issue 1.7.

1.7	2016-01-18	(0) New Section on testing nondeterministic FSMs with respect to reduction. (1) New Section 3.5 on transformation of FSMs to observable ones. (2) New Section 3.6 on minimisation of NFSMs. (3) New part on equivalence class partition testing (no change bars here, since everything is new). (4) New Section 2.6 on translation of testing theories. Change bars mark the differences with respect to Issue 1.6.
1.6	2015-12-05	(0) Completed the description of DFSM minimisation in Section 3.3. (1) Definition of A-equivalence and k-equivalence added in Section 3.2. (2) New Section 4.8, introducing the nondeterministic version of the Wp-Method. (3) Exercise 3 (implementation of the W-Method) added. Change bars mark the differences with respect to Issue 1.5.
1.5	2015-11-16	(1) Added example for W-Method in Section 4.6. (2) Added Section 3.3 on minimisation of DFSMs. (3) Completed Section 3.4 on characterisation sets. Change bars mark the differences with respect to Issue 1.4.
1.4, 1.3	2015-11-09	(1) Added explanation about output alphabets in Section 3.2. (2) New Section 4.5 about testing theory derived from product automata. (3) New Section 4.6 describing the W-Method. Change bars mark the differences with respect to Issue 1.2.
1.2	2015-10-23	(1) Several typos were fixed in Chapter 4, paragraph 'Test cases'. (2) Proof of Theorem 4.3 added. Change bars mark the differences with respect to Issue 1.1.
1.1	2015-10-19	(0) Typos fixed in all sections. (1) Chapter 3: FSM homomorphism defined. Section 3.8: fault injection functions defined. (2) Chapter 4: FSM test cases introduced. State Cover and Transition Cover defined. T-Method described. Change bars mark the differences with respect to Issue 1.0.
1.0	2015-10-09	First edition

Preface

Test automation has many different facets, emphasising different aspects that should be automated during test campaigns. In the early years of test automation (applied in industry from about 1990 on), the focus was on automating the execution of test procedures, formerly written as textual "recipes", which were now realised as executable computer programs (*test scripts*). Since then, automation methods have been elaborated for every conceivable aspect of testing, from test case identification, test data calculation, test procedure generation to automated evaluation of observed against expected results (so-called *test oracles*) and automated compilation of *traceability data* relating test cases, procedures, and results to the requires they verify.

In these lecture notes, we focus on two different approaches to test automation, *model-based testing* and *fuzz testing*.

Model-based testing (MBT) is a testing methodology where the expected behaviour of the *system under test (SUT)* is represented by a reference model, so that "relevant" test cases can be derived from this model. MBT is suitable for testing software and systems consisting of (networks of) controllers with their embedded software and firmware. It is typically applied to verify whether a software or system under test fulfils the (mostly functional) requirements represented by the model.

Fuzz testing (or simply *fuzzing*) is currently one of the most important variants of random software testing. It uses random data, in particular "unexpected" or invalid data as inputs to the software under test and tries to provoke failures like crashes, memory boundary violations, memory leaks, or assertion failures. So, in contrast to MBT, fuzzing aims at exposing *vulnerabilities* of a software component. Therefore, fuzzing complements MBT, but is not an alternative to the model-based approach.

Systematic testing is based on *testing theories* asserting that specific kinds

of tests are capable of finding specific kinds of errors. Testing theories are of considerable interest, because the test suites associated with each theory have a well-defined test strength. As a consequence, they are more easily justified in the context of safety-critical systems, where the trustworthiness of test suites has to be demonstrated.

These lecture notes are structured into four main parts and several appendixes.

- In Part I, testing theories are introduced in general terms, independent on any concrete modelling formalisms.
- In Part II, well-known material about testing against finite state machine (FSM) models is presented.
- In Part III, new material on complete equivalence class testing theories is presented. It is based on variants of Kripke Structures as semantic models. It turns out that the complete FSM testing theories introduced in Part II induce complete equivalence class testing theories for a semantic sub-domain of Kripke Structures.
- In Part IV, the practical application of fuzzing in software testing is described.
- In Appendix B, an open source library is introduced, where many of the algorithms described in Part II have been implemented. The library is called the *FSM Library (fsmlib-cpp)*. The FSM Library also provides a simple test harness for executing test suites generated from FSMs against C-functions.

Contents

List of Figures

Part I

Introduction and Background

Chapter 1

Testing – Basic Definitions

1.1 Basic Terms

In this chapter, some basic notions about testing are summarised in glossary style; they will be discussed in more detail in the chapters to follow. A rather comprehensive set of test-related definitions can be found, for example, in [64], and in the international standard [35, 36, 37, 38].

Dynamic testing. These lecture notes are about *dynamic testing*: the *system under test (SUT)* is executed and receives inputs from the testing environment which then monitors the associated outputs of the SUT and checks them against specifications of the expected behaviour. Dynamic testing can be performed on *host computers* differing from the real operational environment of the SUT or on *target computers* with the real operational hardware, including real hardware interfaces between SUT and environment.

Static analysis. The term *static analysis* or *static testing* is used when the SUT code is analysed instead of executed. Just as in dynamic testing, static analysis has specific objectives to uncover certain types of software errors. Unlike dynamic testing on target computers, however, static analysis cannot uncover errors caused by inappropriate HW/SW integration, such as registers with insufficient length to perform the required calculations without overflows. [1]

[1] If trustworthy models of the underlying HW exist, static analysis becomes also capable of detecting HW/SW integration errors. This is currently an important research field, but

System under test (SUT). The system to be tested – software or hardware – is denoted by *system under test (SUT)*. Alternative terms that are also used in standards and in the testing literature are

- *unit under test (UUT)* if the SUT is a software unit like a function, procedure, or method,
- *implementation under test (IUT)*, synonymous to SUT, and
- *test item*, also synonymous to SUT.

Test case. Following [35], a *test case* is a *set of test case preconditions, inputs (including actions, where applicable), and expected results, developed to drive the execution of a test item to meet test objectives, including correct implementation, error identification, checking quality, and other valued information.*

Test suite. A collection of test cases to be executed against a given SUT is called a *test suite*.

Test strength. The *strength* of a test suite is its capability to uncover errors inside the SUT.

Fault injections (mutations). One approach to determine the strength of a test suite is to *inject faults* into a copy of the SUT. For software testing, this is fairly simple: the program statements are *mutated* in certain places. The test strength of the suite is then measured by the number of mutations uncovered in at least one test case execution divided by the overall number of created mutations. This fraction is called the *mutation score*.

For testing integrated HW/SW systems, the situation is more complex, since certain erroneous HW behaviour cannot be simulated by software residing on the HW. Therefore, corrupted versions of the HW need to be produced in order to be able to test whether the SUT can cope adequately with this behaviour, or the HW has to be corrupted during test execution. This variant of testing is called *intrusive* because it modifies the SUT hardware.

---

cannot be expected to become an accepted best practise in industry in the near future.

Test procedure. Following [70], a *test procedure* consists of *step-by-step instructions for how each test case is to be set up and executed, how the test results are evaluated, and the test environment to be used.*

For automated testing, test procedures are usually realised as executable programs, containing algorithms to establish the pre-conditions, create the test inputs, and check the expected results associated with one or more test cases.

Test oracles. In dynamic testing, the test environment components deciding whether the SUT reactions comply to the expected results associated with a test case are called *checkers* or *test oracles.* The *verdicts* stated by test oracles are on a per-test-case basis

- PASS – the SUT has reacted to the input data associated with a test case as expected
- FAIL – the SUT reactions have violated the expected results specification associated with the test case
- INCONCLUSIVE – the test procedure executing the test case has encountered a runtime error, or the precondition of the test case could not be realised during the test execution. Therefore the test case could not be properly tested – it's neither PASS nor FAIL.

Test harness. In automated software testing, the software component executing a test procedure is called *test harness*[2], because it "puts a harnesses" on the SUT by forcing specific test data to be provided to its input interfaces and checks every reaction on the SUT's output interfaces. Typically, a test harness provides the following services.

1. Activation of the SUT (function call, object instantiation and method call, thread activation, process activation, ...).
2. Provision of input data on the SUT's software interfaces (input parameters, shared memory, global variables, files, sockets, ...).
3. Monitoring of SUT reactions (data written to output parameters, return values, ...) and check against expected results.
4. Documentation of the test results.

[2]harness = *Zaumzeug* in German

Stubs and mock objects. In automated software testing, the SUT may call sub-functions `f` that are not considered to be a part of the SUT, but part of the test environment. For these functions, the test harness needs to specify the return values, output parameters and writes to global variable performed by these sub-functions, in order to check whether the SUT reacts correctly for any conceivable results of the sub-function. We say that the real sub-function `f` is replaced by a *stub*: another function with the same signature as `f` but with a behaviour which is controlled by the test harness.

In object-oriented software testing, the situation is more complex, since the SUT may call methods on objects that have been passed as parameters or reside in object attributes or static class attributes. As a consequence, the test harness needs to provide objects that can be controlled by the harness. These are called *mock objects*.

Model-based testing. Model-based testing (MBT) can be implemented using different approaches; this is also expressed in the current definition of MBT presented in Wikipedia[3].

> *Model-based testing is an application of model-based design for designing and optionally also executing artifacts to perform software testing or system testing. Models can be used to represent the desired behavior of a System Under Test (SUT), or to represent testing strategies and a test environment.*

In these lecture notes, we follow the variant where formal models represent the desired behaviour of the SUT, because this promises the maximal return of investment for the effort to be spent on test model development.

- Test cases can be automatically identified in the model.
- If the model contains links to the original requirements (this is systematically supported, for example, by the SysML modelling language [48]), test cases can be automatically traced back to the requirements they help to verify.
- Since the model is associated with a formal semantics, test cases can be represented by means of logical formulas representing reachability goals, and concrete test data can be calculated by means of constraint solvers.

[3] `https://en.wikipedia.org/wiki/Model-based_testing`, 2016-07-11

- Using model-to-text transformations, executable test procedures, including test oracles, can be generated in an automated way.
- Comprehensive traceability data linking test results, procedures, test cases, and requirements can be automatically compiled.

Requirements tracing. One of the main objectives of testing is to *cover* a set of given *requirements* specifying the expected SUT behaviour. It is therefore necessary to *trace* each test case back to the set of requirements it helps to verify. The relationship between requirements is $\mathtt{n} : \mathtt{m}$. One requirement may be tested by several test cases, and one test case may help to verify several requirements.

Exercise 1. The European Train Control System ETCS relies on the existence of an onboard controller in train engines, the *European Vital Computer EVC*. Its functionality and basic architectural features are described in the public ETCS system specification [17]. One functional category of the EVC covers aspects of speed and distance monitoring, to accomplish the *"...supervision of the speed of the train versus its position, in order to assure that the train remains within the given speed and distance limits."* [68, 3.13.1.1]. Speed and distance monitoring is decomposed into three sub-functions [68, 3.13.10.1.2], where only one out of these three is active at a point in time. *Ceiling speed monitoring (CSM)* supervises the observance of the maximal speed allowed according to the current most restrictive speed profile (MRSP). CSM is active while the train does not approach a target (train station, level crossing, or any other point that must be reached with predefined speed). The other variants of speed monitoring are not considered here; they are applied in situations when the train has to "brake to a target" or approaches the end of its movement authority.

In this exercise, a C++ class implementing the control functions of the CSM is developed and tested in an intuitive way. In subsequent chapters, we will come back to this class and test it more systematically, applying the methods which are introduced in these lecture notes.

The CSM manages the internal states `NORMAL`, `OVERSPEED`, `WARNING`, `SVC_BRAKE_INTERVENTION`, and `EMER_BRAKE_INTERVENTION` which are processed as follows.

1. State `NORMAL` is used while the actual train speed `v` is less or equal to the maximal speed `vMax` allowed. The status is indicated by value

OK on the display output d to the train engine driver. The brakes are released.

From NORMAL, transitions into the following internal states are performed.

- To OVERSPEED, when condition

$$\texttt{vMax} < \texttt{v} \leq \texttt{vMax} + \texttt{dW}(\texttt{vMax})$$

  holds, where the threshold function dW(vMax) is specified below.

- To WARNING, when condition

$$\texttt{vMax} + \texttt{dW}(\texttt{vMax}) < \texttt{v} \leq \texttt{vMax} + \texttt{dSI}(\texttt{vMax})$$

  holds, where the threshold function dSI(vMax) is specified below.

- To SVC_BRAKE_INTERVENTION, when condition

$$\texttt{vMax} + \texttt{dSI}(\texttt{vMax}) < \texttt{v} \leq \texttt{vMax} + \texttt{dEI}(\texttt{vMax})$$

  holds, where the threshold function dEI(vMax) is specified below.

- To EMER_BRAKE_INTERVENTION, when condition

$$\texttt{vMax} + \texttt{dEI}(\texttt{vMax}) < \texttt{v}$$

  holds.

2. State OVERSPEED is used to indicate a negligible violation of the speed limit. The status is indicated by value OVR on the display output d to the train engine driver. The brakes are released.

   From OVERSPEED, transitions into the following internal states are performed.

   - To NORMAL, when condition

$$\texttt{v} \leq \texttt{vMax}$$

     holds.

- To WARNING, when condition

$$\texttt{vMax} + \texttt{dW}(\texttt{vMax}) < \texttt{v} \leq \texttt{vMax} + \texttt{dSI}(\texttt{vMax})$$

  holds.

- To SVC_BRAKE_INTERVENTION, when condition

$$\texttt{vMax} + \texttt{dSI}(\texttt{vMax}) < \texttt{v} \leq \texttt{vMax} + \texttt{dEI}(\texttt{vMax})$$

  holds.

- To EMER_BRAKE_INTERVENTION, when condition

$$\texttt{vMax} + \texttt{dEI}(\texttt{vMax}) < \texttt{v}$$

  holds.

Note that the conditions to reach another state from OVERSPEED are the same as the jump conditions specified for NORMAL.

3. State WARNING is used when another warning level is needed, since the train is overspeeding even more. The status is indicated by value WRN on the display output d to the train engine driver. The brakes are released.

   From WARNING, transitions into the following internal states are performed.

   - To NORMAL, when condition

$$\texttt{v} \leq \texttt{vMax}$$

     holds.

   - To SVC_BRAKE_INTERVENTION, when condition

$$\texttt{vMax} + \texttt{dSI}(\texttt{vMax}) < \texttt{v} \leq \texttt{vMax} + \texttt{dEI}(\texttt{vMax})$$

     holds.

   - To EMER_BRAKE_INTERVENTION, when condition

$$\texttt{vMax} + \texttt{dEI}(\texttt{vMax}) < \texttt{v}$$

     holds.

4. State SVC_BRAKE_INTERVENTION is used brake the train until normal speed is reached again, using the service brakes only. The status is indicated by value SI on the display output d to the train engine driver. The service brakes are triggered, the emergency brakes are released.

   From SVC_BRAKE_INTERVENTION, transitions into the following internal states are performed.

   - To NORMAL, when condition

     $$\mathtt{v} \leq \mathtt{vMax}$$

     holds.
   - To EMER_BRAKE_INTERVENTION, when condition

     $$\mathtt{vMax} + \mathtt{dEI}(\mathtt{vMax}) < \mathtt{v}$$

     holds.

5. State EMER_BRAKE_INTERVENTION is used to brake the train using both service brakes and emergency brakes. The status is indicated by value SI on the display output d to the train engine driver. Both service brakes and emergency brakes are triggered.

   From EMER_BRAKE_INTERVENTION, transitions into the following internal states are performed.

   - To NORMAL, when condition

     $$\mathtt{v} = 0 \vee (\mathtt{v} \leq \mathtt{vMax} \wedge \mathtt{c})$$

     holds. This means that
     - either the train has come to a standstill ($\mathtt{v} = 0$), or
     - $\mathtt{v} \leq \mathtt{vMax}$, and the country the train is travelling in allows for releasing the emergency and service brakes already if this holds. This situation is marked by a country flag c.

The threshold function dW(vMax), dSI(vMax), and dEI(vMax) used to guard the transitions between states are defined as follows [68, 3.13.9.2.3].

$$\mathtt{dW}(\mathtt{vMax}) = \begin{cases} 4 & \text{if } \mathtt{vMax} \leq 110 \\ \frac{1}{3} + \frac{1}{30} \cdot \mathtt{vMax} & \text{if } 110 < \mathtt{vMax} \leq 140 \\ 5 & \text{if } 140 < \mathtt{vMax} \end{cases} \tag{1.1}$$

$$\texttt{dSI}(\texttt{vMax}) = \begin{cases} 5.5 & \text{if} \quad \texttt{vMax} \leq 110 \\ 0.55 + 0.045 \cdot \texttt{vMax} & \text{if} \quad 110 < \texttt{vMax} \leq 210 \\ 10 & \text{if} \quad 210 < \texttt{vMax} \end{cases} \tag{1.2}$$

$$\texttt{dEI}(\texttt{vMax}) = \begin{cases} 7.5 & \text{if} \quad \texttt{vMax} \leq 110 \\ -0.75 + 0.075 \cdot \texttt{vMax} & \text{if} \quad 110 < \texttt{vMax} \leq 210 \\ 15 & \text{if} \quad 210 < \texttt{vMax} \end{cases} \tag{1.3}$$

The header file of the class to be programmed is given in the listing `CSM.hpp` below. The main control function of the CSM is public method `doCSM()`. It gets the following inputs.

1. Variable `v` contains the current train speed provided by the environment.

2. Variable `vMax` contains the MRSP value, that is, the maximal train speed currently allowed.

3. The country flag `c` is `true` if after having entered the emergency brake state, the CSM may release the brakes already if the current speed is again below the maximal speed allowed. Otherwise, if `c` evaluates to `false`, the train must come to a standstill ($\texttt{v} = 0$), before the brakes may be released again.

The control function `doCSM()` sets the following outputs.

1. Variable `svcBrake` triggers the service brake if set by the method to 1, otherwise the value is 0.

2. Variable `emerBrake` triggers the emergency brake if set by the method to 1, otherwise the value is 0.

3. Variable `d` sets the display for the train engine driver. While in state `NORMAL`, the display should show `OK`. When state `OVERSPEED` is entered, the display must be set to `OVR`. When entering state `WARNING`, the display must show `WRN`. When in states `SVC_BRAKE_INTERVENTION` or `EMER_BRAKE_INTERVENTION`, the display must show `IV`.

Your tasks for this exercise are as follows.

1. Program the class implementation in a file `CSM.cpp`.

2. Program a `main.cpp` file that tests the CSM implementation as follows.

   (a) You may create and destroy the CSM instances several times if suitable for your test suite.

   (b) Call the `doCSM()` method with selected values and check whether the associated outputs are correct. Do this by using the checking and logging function listed below.

   (c) Identify each call to `doCSM()` by a test case identifier.

   (d) Structure the text above into a list of atomic requirements.

   (e) Create a table which documents which test cases contribute to testing a given requirement (see examples in the `main.cpp` listing below).

□

```
1  //
2  //  CSM.hpp
3  //  CSM
4  //
5  //  Created by Jan Peleska on 2016-10-18.
6  //  Copyright (c) 2016 Jan Peleska. All rights reserved.
7  //
8
9  #ifndef CSM_hpp
10 #define CSM_hpp
11
12 #include <stdio.h>
13
14
15 /**
16  *  Class for the Ceiling Speed Monitor (CSM)
17  */
18 class CSM {
19
20 public:
21
22     typedef enum {
23         OK,
24         OVR,
25         WRN,
26         IV
```

```
27      } display_t;
28
29  private:
30
31      typedef enum {
32          NORMAL,
33          OVERSPEED,
34          WARNING,
35          SVC_BRAKE_INTERVENTION, /** service brake intervention */
36          EMER_BRAKE_INTERVENTION /** emergency brake intervention */
37      } csm_state_t;
38
39      csm_state_t state;
40
41      /** Calculate threshold value for transition into state WARNING */
42      float dW(float vMax);
43
44      /** Calculate threshold value for transition into state
45       * SVC_BRAKE_INTERVENTION
46       */
47      float dSI(float vMax);
48
49      /** Calculate threshold value for transition into state
50       * EMER_BRAKE_INTERVENTION
51       */
52      float dEI(float vMax);
53
54      /** Process inputs in state NORMAL */
55      csm_state_t processN(float v, float vMax, bool c,
56                           int& svcBrake, int& emerBrake, display_t& d);
57
58      /** Process inputs in state OVERSPEED */
59      csm_state_t processO(float v, float vMax, bool c,
60                           int& svcBrake, int& emerBrake, display_t& d);
61
62      /** Process inputs in state WARNING */
63      csm_state_t processW(float v, float vMax, bool c,
64                           int& svcBrake, int& emerBrake, display_t& d);
65
66      /** Process inputs in state SVC_BRAKE_INTERVENTION */
67      csm_state_t processS(float v, float vMax, bool c,
68                           int& svcBrake, int& emerBrake, display_t& d);
69
70      /** Process inputs in state EMER_BRAKE_INTERVENTION */
71      csm_state_t processE(float v, float vMax, bool c,
```

```
72                              int& svcBrake, int& emerBrake, display_t& d);
73
74  public:
75
76      /** Constructor */
77      CSM();
78
79      /**
80       *  Perform one step of the ceiling speed monitoring
81       * @param[in] v current train speed
82       * @param[in] vMax current maximal speed allowed
83       * @param[in] c country flag - see CSM description
84       * @param[out] svcBrake output to service brake in range
85       *             0 (do not trigger the service brake)
86       *             1 (trigger the service brake)
87       * @param[out] emerBrake output to emergency brake in range
88       *             0 (do not trigger the emergency brake)
89       *             1 (trigger the emergency brake)
90       * @param[out] d display command
91       */
92      void doCSM(float v, float vMax, bool c,
93                 int& svcBrake, int& emerBrake, display_t& d);
94
95      /** Reset CSM instance to its initial state NORMAL */
96      void reset();
97
98  };
99
100 #endif /* CSM_hpp */
```

```
1  //
2  //  main.cpp
3  //  CSM
4  //
5  //  Created by Jan Peleska on 2016-10-18.
6  //  Copyright (c) 2016 Jan Peleska. All rights reserved.
7  //
8
9  #include <iostream>
10 #include <string>
11
12 #include "CSM.hpp"
13
14 /*
15  Requirements specification
16
```

```
17   REQ-N-001 In state NORMAL the message OK is displayed on output d
18   REQ-N-002 In state NORMAL, a transition to OVERSPEED is performed,
19             if vMax < v <= vMax + dW(vMax)
20   REQ-N-003 In state NORMAL, a transition to WARNING is performed, if
21             vMax + dW(vMax) < v < vMax + dSI(vMax)
22   ...
23   */
24
25  /* Requirements tracing
26
27   REQ-N-001: TC-N-001, TC-N-002, ...
28   REQ-N-002: TC-N-003, ...
29   ...
30   */
31
32
33  void assertAndLog(bool cnd, std::string testCase,
34                    int svcBrake, int emerBrake, CSM::display_t d) {
35      std::string verdict;
36
37      verdict = (cnd) ? "PASS" : "FAIL";
38
39      std::cout << testCase << " : " << verdict
40      << " svcBrake = " << svcBrake << " emerBrake = " << emerBrake
41      << " d = " << d << std::endl;
42  }
43
44  int main(int argc, const char * argv[]) {
45
46      int svcBrake;
47      int emerBrake;
48      CSM::display_t d;
49
50      CSM csm;
51
52      // Run test cases
53      csm.doCSM(0.0,0.0,false,svcBrake,emerBrake,d);
54      assertAndLog(svcBrake == 0 and emerBrake == 0 and d == CSM::OK,
55                   "TC-N-001",svcBrake,emerBrake,d);
56
57      csm.doCSM(500.0,500.0,false,svcBrake,emerBrake,d);
58      assertAndLog(svcBrake == 0 and emerBrake == 0 and d == CSM::OK,
59                   "TC-N-002",svcBrake,emerBrake,d);
60
61      csm.doCSM(500.01,500.0,false,svcBrake,emerBrake,d);
```

```
62      assertAndLog(svcBrake == 0 and emerBrake == 0 and d == CSM::OVR,
63                   "TC-N-003",svcBrake,emerBrake,d);
64  ...
65  }
```

1.2 Variants of Test Purposes

Testing is usually performed with a specific purpose in mind. There are different purposes to perform tests, and these influence the underlying methods and techniques applied for generating test cases, test data, and for specifying expected results.

Functional testing executes test cases to verify whether the required behaviour of the SUT has been implemented correctly.

Vulnerability testing is performed to uncover errors in the SUT, regardless of whether they reveal functional, structural, or non-functional discrepancies.

Robustness testing checks whether the SUT can cope with erroneous environment behaviour.

Stress testing is used to verify whether the SUT can cope with the maximal work load.

Avalanche testing is performed to check whether the SUT can recover properly after a temporary overload.

Error guessing uses test cases where the test engineers expect vulnerabilities in the SUT.

Scenario testing (end-to-end testing) verifies whether complete services expected by the end users are performed correctly by the SUT. Executing a scenario typically involves the execution of a chain of test cases of functional and robustness tests.

1.3 Test Levels

Testing is usually performed on different *levels* associated with different views on the SUT.

Unit testing "sees" the individual functions, procedures, methods, classes, objects of a SUT and tests each of those separately by using stubs and mock objects as explained above.

Software integration testing verifies the correct cooperation between *several* functions, procedures, methods and the correct interaction between objects of several classes, threads and processes.

This immediately suggests to perform software integration tests on different levels:

1. Interacting functions/procedures of the same library or methods of the same class.
2. Interacting libraries or objects instantiated from different classes.
3. Interacting threads.
4. Interacting processes.

Hardware/software integration testing verifies the correctness of one integrated HW/SW system (i.e. one controller with its software).

System integration testing verifies the correct cooperation between several integrated components that are part of the complete system.

System testing verifies the correctness of the complete system, typically executed in its real operational environment.

Chapter 2

Testing Theories

2.1 Model-based Testing

In *model-based testing (MBT)*, the behaviour of the *system under test (SUT)* is compared to that of a *reference model*, using a so-called *conformance relation*.

Informally speaking, a *testing theory* states that under certain hypotheses, test suites derived from a given reference model will uncover all failures of a certain type. Moreover, it is obvious that these test suites should never reject a system under test (SUT) that conforms to the reference model. These informal notions can be formalised as described in the following section. While this formalisation is rather abstract, it will be illustrated in the chapters to follow.

2.2 Programs are Models

The notion of *models* used in this document is quite general. We only require the existence of an abstract syntax (which models are well-formed?) and behavioural semantics (which state transitions can be performed, which sequences of inputs and outputs can be observed?). In particular, computer programs are also considered as models, because they represent abstractions of low-level actions – think, for example, of the microcode executed by a CPU.

2.3 Fault Models

Given a signature Sig of models, a *fault model* $\mathcal{F}(\mathsf{M}, \leq, \mathsf{Dom})$ specifies a *reference model* $\mathsf{M} \in \mathsf{Sig}$, a conformance relation $\leq \subseteq \mathsf{Sig} \times \mathsf{Sig}$ between models, and a *fault domain* $\mathsf{Dom} \subseteq \mathsf{Sig}$.

The notion of *signatures* comes from the field of model theory [11], where it has been observed that certain categories of objects, say, models fulfilling a certain specification, depend on the symbols used in this context, for example, the free variables referenced in the specification. A signature fixes the symbol set, and changing this set corresponds to translating models and specifications from one signature into another.

Our definition of fault models generalises the one originally presented in [56], where it had been introduced in the context of finite state machines.

2.4 Test Cases

Given a signature Sig, we introduce the set $\mathrm{TC}(\mathsf{Sig})$ of *test cases* for Sig in a very abstract way, by just requiring the existence of

- a total relation $\underline{\mathrm{pass}} \subseteq \mathsf{Sig} \times \mathrm{TC}(\mathsf{Sig})$ and
- a function $\# : \mathsf{tc}(\mathsf{Sig}) \to \mathbb{N} \cup \{\infty\}$

Intuitively speaking, a test case is just an object that lets us decide whether a model of the signature "passes the object" or not, and whether the test execution will be finite. In the chapters to follow we will see that test cases can be represented in different ways, for example, as sets of observations, or as executable entities interacting concurrently with the SUT.

For $(\mathsf{M}, \mathsf{U}) \in \underline{\mathrm{pass}}$, the infix notation $\mathsf{M}\ \underline{\mathrm{pass}}\ \mathsf{U}$ is used, and interpreted as *'Model* M *passes the test case* U*'*. The function $\#$ associates a *length* with each test case. For practical applications, $\#(\mathsf{U})$ is set to the maximal number of test steps to be executed when test U is run against some SUT. From a theoretical perspective, it is also interesting to consider test cases of infinite length ($\#(\mathsf{U}) = \infty$).

2.5 Test Suites and Complete Testing Theories

A *test suite* $\mathbf{TS} \subseteq \mathrm{TC}(\mathsf{Sig})$ denotes a set of test cases. A model M *passes the test suite* $\mathbf{TS}$, also written as M $\underline{\text{pass}}$ $\mathbf{TS}$, if and only if M $\underline{\text{pass}}$ U for all $\mathsf{U} \in \mathbf{TS}$. A test suite $\mathbf{TS}$ is *finite* if it contains finitely many test cases, and every test case $\mathsf{U} \in \mathbf{TS}$ is finite in the sense that $\#(\mathsf{U}) \in \mathbb{N}$.

Definition 2.1 *A test suite* $\mathbf{TS}$ *is called* complete *with respect to fault model* $\mathcal{F}(\mathsf{M}, \leq, \mathsf{Dom})$*, if and only if the following properties hold.*

1. *If a member of the fault domain conforms to the reference model, it passes the test suite, that is,*

$$\forall \mathsf{M}' \in \mathsf{Dom} : \mathsf{M}' \leq \mathsf{M} \Rightarrow (\forall \mathsf{U} \in \mathbf{TS} : \mathsf{M}' \ \underline{\text{pass}} \ \mathsf{U})$$

   This property is called soundness *of the test suite.*

2. *If a member of the fault domain passes the test suite, it conforms to the reference model, that is,*

$$\forall \mathsf{M}' \in \mathsf{Dom} : (\forall \mathsf{U} \in \mathbf{TS} : \mathsf{M}' \ \underline{\text{pass}} \ \mathsf{U}) \Rightarrow \mathsf{M}' \leq \mathsf{M}$$

   This property is called exhaustiveness.

□

A *complete testing theory* is a mapping $\mathbf{TS} : \mathsf{F} \rightarrow \mathbb{P}(\mathrm{TC}(\mathsf{Sig}))$ from a set F of fault models to test suites, such that the test suites $\mathbf{TS}(\mathcal{F})$ are complete for all fault models $\mathcal{F} = \mathcal{F}(\mathsf{M}, \leq, \mathsf{Dom}) \in \mathsf{F}$. A testing theory $\mathbf{TS} : \mathsf{F} \rightarrow \mathbb{P}(\mathrm{TC}(\mathsf{Sig}))$ is *finite* if and only of $\mathsf{F}(\mathcal{F})$ is a finite test suite for all $\mathcal{F} \in \mathsf{F}$.

2.6 Translation of Testing Theories

Let Sig_1 and Sig_2 be two signatures with conformance relations $\leq_1$ and $\leq_2$, and test case relations $\underline{\text{pass}}_1$ and $\underline{\text{pass}}_2$, respectively. A function $\mathsf{T} : \underline{\mathsf{Sig}_1} \rightarrow \mathsf{Sig}_2$ defined on a sub-domain $\underline{\mathsf{Sig}_1} \subseteq \mathsf{Sig}_1$ is called a *model map*, and a function $\mathsf{T}^* : \mathrm{TC}(\mathsf{Sig}_2) \rightarrow \mathrm{TC}(\underline{\mathsf{Sig}_1})$ is called a *test case map*. Note that models and test cases are mapped in opposite directions (see Fig. 2.1).

Definition 2.2 (Satisfaction Condition) *The pair* $(\mathsf{T}, \mathsf{T}^*)$ *fulfils the* satisfaction condition *if and only if the following conditions* **SC1** *and* **SC2** *are fulfilled.*

SC1 *The model map is compatible with the conformance relations under consideration, in the sense that*

$$\forall \mathcal{S}, \mathcal{S}' \in \underline{\mathsf{Sig}_1} : \mathcal{S}' \leq_1 \mathcal{S} \Leftrightarrow \mathsf{T}(\mathcal{S}') \leq_2 \mathsf{T}(\mathcal{S}),$$

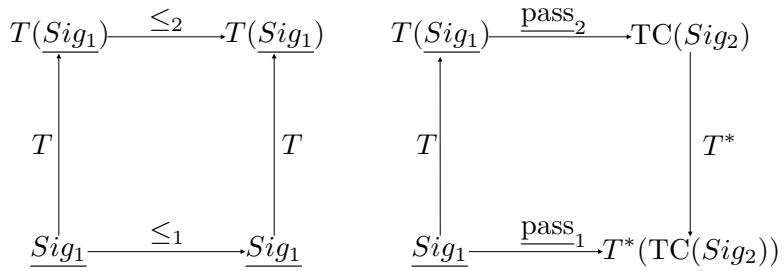
so the left-hand side diagram in Fig. 2.1 commutes due to the fact that $\mathsf{T}; \leq_2 = \leq_1; \mathsf{T}$.[1]

SC2 *Model map and test case map preserve the* $\underline{\text{pass}}$*-relationship in the sense that*

$$\forall \mathcal{S} \in \underline{\mathsf{Sig}_1}, \mathcal{U} \in TC(\mathsf{Sig}_2) : \mathsf{T}(\mathcal{S}) \ \underline{\text{pass}}_2 \ \mathcal{U} \Leftrightarrow \mathcal{S} \ \underline{\text{pass}}_1 \ \mathsf{T}^*(\mathcal{U}),$$

so the right-hand side diagram in Fig. 2.1 commutes, due to the fact that $\underline{\text{pass}}_1 = \mathsf{T}; \underline{\text{pass}}_2; \mathsf{T}^*$.

□



$$T; \leq_2 = \leq_1; T \qquad \underline{\text{pass}}_1 = T; \underline{\text{pass}}_2; T^*$$

Figure 2.1: Commuting diagrams reflecting the satisfaction condition.

Given a pair $(\mathsf{T}, \mathsf{T}^*)$ fulfilling the satisfaction condition, this allows to translate complete testing theories existing in Sig_2 to likewise complete testing theories in (sub-domains of) Sig_1.

[1] Operator ";" denotes the relational composition defined for functions or relations $\mathsf{f} \subseteq \mathsf{A} \times \mathsf{B}$, $\mathsf{g} \subseteq \mathsf{B} \times \mathsf{C}$ by $\mathsf{f}; \mathsf{g} = \{(\mathsf{a}, \mathsf{c}) \in \mathsf{A} \times \mathsf{C} \mid \exists \mathsf{b} \in \mathsf{B} : (\mathsf{a}, \mathsf{b}) \in \mathsf{f} \wedge (\mathsf{b}, \mathsf{c}) \in \mathsf{g}\}$. Note that $\mathsf{f}; \mathsf{g}$ is evaluated from left to right (like composition of code fragments), as opposed to right-to-left evaluation which is usually denoted by $\mathsf{g} \circ \mathsf{f}$.

Theorem 2.1 *Suppose that* $\mathbf{TS}_2 : \mathsf{F}_2 \to \mathbb{P}(\mathsf{TC}(\mathsf{Sig}_2))$ *is a sound (exhaustive, complete) testing theory. Define*

$$\begin{aligned}\mathsf{F}_1 = \{(\mathcal{S}, \leq_1, \mathsf{Dom}_1) \in \mathsf{F}(\underline{\mathsf{Sig}_1}, \leq_1) \mid \\ \exists \mathsf{Dom}_2 \subseteq \mathsf{Sig}_2 : \mathsf{T}(\mathsf{Dom}_1) \subseteq \mathsf{Dom}_2 \wedge (\mathsf{T}(\mathcal{S}), \leq_2, \mathsf{Dom}_2) \in \mathsf{F}_2\}.\end{aligned}$$

Then $\mathbf{TS}_1 : \mathsf{F}_1 \to \mathbb{P}(\mathsf{TC}(\mathsf{Sig}_1))$ *defined by* $\mathbf{TS}_1(\mathcal{S}, \leq_1, \mathsf{Dom}_1) = \mathsf{T}^*(\mathbf{TS}_2(\mathsf{T}(\mathcal{S}), \leq_2, \mathsf{Dom}_2))$*, such that* $\mathsf{T}(\mathsf{Dom}_1) \subseteq \mathsf{Dom}_2$*, is a sound (exhaustive, complete) testing theory.*

Proof. Suppose $\mathbf{TS}_2$ is sound (exhaustive). Let $\mathcal{F}_1 = (\mathcal{S}, \leq_1, \mathsf{Dom}_1) \in \mathsf{F}_1$ and $\mathcal{F}_2 = (\mathsf{T}(\mathcal{S}), \leq_2, \mathsf{Dom}_2) \in \mathsf{F}_2$ be any fault models in F_1 and F_2 respectively, satisfying $\mathbf{TS}_1(\mathcal{F}_1) = \mathsf{T}^*(\mathbf{TS}_2(\mathcal{F}_2))$. Let $\mathcal{S}' \in \mathsf{Dom}_1$. Then $\mathsf{T}(\mathcal{S}') \in \mathsf{Dom}_2$, and

$$\begin{array}{lcll}
\mathcal{S}' \leq_1 \mathcal{S} & \Leftrightarrow & \mathsf{T}(\mathcal{S}') \leq_2 \mathsf{T}(\mathcal{S}) & [\text{satisfaction condition } \mathbf{SC1}] \\
& \Rightarrow & \forall \mathsf{U} \in \mathbf{TS}_2(\mathcal{F}_2) : \mathsf{T}(\mathcal{S}') \underline{\text{pass}}_2 \mathsf{U} & [\mathbf{TS}_2 \text{ is sound}] \\
& (\Leftarrow) & & [\mathbf{TS}_2 \text{ is exhaustive}] \\
& \Leftrightarrow & \forall \mathsf{U} \in \mathbf{TS}_2(\mathcal{F}_2) : \mathcal{S}' \underline{\text{pass}}_1 \mathsf{T}^*(\mathsf{U}) & [\text{satisfaction condition } \mathbf{SC2}] \\
& \Leftrightarrow & \mathcal{S}' \underline{\text{pass}}_1 \mathsf{T}^*(\mathbf{TS}_2(\mathcal{F}_2)) & \\
& \Leftrightarrow & \mathcal{S}' \underline{\text{pass}}_1 \mathbf{TS}_1(\mathcal{F}_1) & [\mathbf{TS}_1(\mathcal{F}_1) = \mathsf{T}^*(\mathbf{TS}_2(\mathcal{F}_2))]
\end{array}$$

Hence $\mathbf{TS}_1(\mathcal{F}_1)$ is a sound (exhaustive) test suite for any fault model $\mathcal{F}_1 \in \mathsf{F}_1$. Consequently, $\mathbf{TS}_1$ is sound (exhaustive). Completeness is the combination of soundness and exhaustiveness, so this proves the theorem. □

Summarising, the existence of a model map and a test case map $(\mathsf{T}, \mathsf{T}^*)$ fulfilling the satisfaction condition allows us to translate every complete, sound, or exhaustive testing theory elaborated for Sig_2 to a complete, sound or exhaustive testing theory in $\underline{\mathsf{Sig}_1}$ by simply translating complete Sig_2-test suites via T^* to Sig_1-test suites. The crucial point is the construction of $(\mathsf{T}, \mathsf{T}^*)$ for the signatures under consideration, and the proof of the satisfaction condition. We will apply this strategy in the subsequent sections to signatures of FSMs playing the role of Sig_2, which allows us to translate existing FSM testing theories to signatures $\underline{\mathsf{Sig}_1}$ representing variants of RIOSTSs.

2.7 Testability Hypothesis

The *testability hypothesis* states that the true behaviour of the SUT is equivalent to one of the models in the fault domain. The validity of this assumption ensures that complete test suites will really uncover all SUT errors of a given type. In particular, the hypothesis states that the modelling formalism used is adequate to represent all "interesting" behaviours of the SUT. For example, if the SUT can only be considered as correct if it also meets some timing constraints, then a modelling formalism abstracting from time is inadequate, and the true behaviour of the SUT will never be inside the fault domain.

When performing white-box software tests where the complete program code can be analysed, the testability hypothesis can often be checked by means of static analysis. When performing black-box tests, in particular, against integrated hardware/software systems, however, it is often impossible to ensure its validity. This motivates why the verification of safety-critical systems can never be completed by black-box testing methods only.

2.8 Uniformity Hypothesis and Regularity Hypothesis

From a practical perspective, test suites should be finite, and each test case should terminate after a finite number of steps. Therefore testing theories are often established by proving the validity of two further hypotheses.

The *uniformity hypothesis* deals with the problem that data domains – for example, the inputs to the SUT – are physically or conceptually infinite, so that it is impossible to enumerate all possible values in a test suite. The hypothesis assumes the possibility to *partition* the domains under consideration into finitely many sub-domains D, so that it suffices to test a single representative from each sub-domain in order to conclude that the SUT would pass the infinite number of tests exercising *all* elements of the complete domain. The name of the hypothesis has been chosen because it assumes that the SUT behaves *uniformly* for all elements of a sub-domain D.

The *regularity hypothesis* deals with the problem of test cases with infinite lengths. It assumes that passing all cases of a maximal finite length (this length has to be specified by the testing theory) implies that *all* test cases – whether finite or infinite – will be passed.

The uniformity and regularity hypothesis have been originally introduced in [20].

Part II

Testing Finite State Machines

Chapter 3

Finite State Machines

In this chapter, we introduce definitions and notations for finite state machines (FSMs) that have been introduced in contributions on FSM testing, such as [54, 55, 59, 26]. A very comprehensive description of FSMs with many interesting results can be found in [66].

3.1 FSM Definition

Definition 3.1 *A* Finite State Machine (FSM) *is a tuple* $M = (Q, \underline{q}, \Sigma_I, \Sigma_O, h)$ *with state space* Q*, input alphabet* Σ_I*, output alphabet* Σ_O*, where* Q, Σ_I, Σ_O *are finite and nonempty sets.* $\underline{q} \in Q$ *denotes the initial state.* $h \subseteq Q \times \Sigma_I \times \Sigma_O \times Q$ *is the transition relation. If* $(q, x, y, q') \in h$*, we say that there is a transition from* q *to* q' *with input* x *and output* y*. The set of state machines over a given input alphabet* Σ_I *and output alphabet* Σ_O *is called an* FSM signature *and denoted by* $\mathrm{FSM}(\Sigma_I, \Sigma_O)$*.* □

Observe that there are two variants how FSMs can be defined:

1. The definition given above introduces the FSM variant known as *Mealy Automata* and is typically used for modelling reactive control systems that accept inputs and – depending on the current state and the input value – produce outputs and transit to a target state.

2. In the context of language theory, FSMs usually have a single alphabet, where each element corresponds to a symbol accepted or rejected in the given state. Termination states are typically explicitly defined as a subset of Q.

If a Mealy automaton $M = (Q, \underline{q}, \Sigma_I, \Sigma_O, h)$ according to Definition 3.1 is given, an FSM M' conforming to the second variant can be constructed by means of alphabet abstraction:

1. The alphabet of M' is $\Sigma = \Sigma_I \times \Sigma_O$, that is, every input/output pair is considered as a single label which is contained in the alphabet Σ.

2. The transition relation $h' \subseteq Q \times \Sigma \times Q$ of M' is defined by

$$\forall q, q' \in Q, z \in \Sigma : \big((q, z, q') \in h' \Leftrightarrow \\ (\exists a \in \Sigma_I, b \in \Sigma_O : z = (a, b) \wedge (q, a, b, q') \in h)\big)$$

3. The set of termination states is identified explicitly by

$$Q_{Term} = \{q \in Q \mid \forall a \in \Sigma_I, b \in \Sigma_O, q' \in Q : (q, a, b, q') \notin h\}$$

3.2 Basic Properties of FSMs

We use both set notation $(q, x, y, q') \in h$ and Boolean notation $h(q, x, y, q')$ for specifying that (q, x, y, q') is a transition in h. We call x a *defined* input in state q, if there is a transition from q with input x. If every input of Σ_I is defined in every state, M is *completely specified*. FSM M is called a *deterministic FSM (DFSM)*, if for any state q and defined input x, $h(q, x, y, q') \wedge h(q, x, y', q'')$ implies $(y, q') = (y, q'')$. Intuitively speaking, a specific input applied to a specific state uniquely determines both post-state and associated output. If M is not deterministic, it is called a *nondeterministic FSM (NFSM)*. If there is no emanating transition for $q \in Q$, this state is called a *deadlock state*, and M *terminates in* q. The set of deadlock states is denoted by $\mathit{deadlock}(Q) \subseteq Q$. The set of states that do not deadlock is denoted by $\mathrm{DF}(Q) = \{q \in Q \mid \exists (q', x, y, q'') \in h : q' = q\}$.

The transition relation h can be extended in a natural way to input traces: let $\overline{x}$ be an input trace and $\overline{y}$ an output trace. Then $(q, \overline{x}, \overline{y}, q') \in h$, if and only if there is a transition sequence from q to q' with input trace $\overline{x}$ and output trace $\overline{y}$. If q is the initial state $\underline{q}$, such a transition sequence is called an *execution* of M. Executions are written in the notation

$$q_0 \xrightarrow{x_1/y_1} q_1 \xrightarrow{x_2/y_2} \ldots \xrightarrow{x_k/y_k} q_k$$

with $q_0 = \underline{q}$, $h(q_{i-1}, x_i, y_i, q_i)$ for $i = 1, \ldots, k$, and $\overline{x} = x_1 \ldots x_k$ and $\overline{y} = y_1 \ldots y_k$.

We use notation $q_2 \in q_1\text{-}\underline{\text{after}}\text{-}\overline{x}$ if there exists an output trace $\overline{y}$, such that $(q_1, \overline{x}, \overline{y}, q_2) \in h$. Notation $q_2 \in q_1\text{-}\underline{\text{after}}\text{-}\overline{x}/\overline{y}$ indicates that $(q_1, \overline{x}, \overline{y}, q_2) \in h$. If $q_1\text{-}\underline{\text{after}}\text{-}\overline{x}$ contains only a single element, the set notation is dropped, and we write $q_2 = q_1\text{-}\underline{\text{after}}\text{-}\overline{x}$. Analogously, notation $q_2 = q_1\text{-}\underline{\text{after}}\text{-}\overline{x}/\overline{y}$ is used.

The empty trace is denoted by ε, and $(q, \varepsilon, \varepsilon, q) \in h$, for any state q. A *language* of an FSM M is the set consisting of all possible input/output traces in M; we use notation $L_M(q) = \{\overline{x}/\overline{y} \mid \exists q' \in Q : h(q, \overline{x}, \overline{y}, q')\}$ for $q \in Q$, and $L(M) = L_M(\underline{q})$. By $\text{FSM}(\Sigma_I, \Sigma_O)$ we denote the signature of all FSMs with input alphabet Σ_I and output alphabet Σ_O.

If $M_1 \in \text{FSM}(\Sigma_I, \Sigma_{O_1})$, it is not guaranteed that every output $y \in \Sigma_{O_1}$ will be used in some transition. Therefore, if $M_2 \in \text{FSM}(\Sigma_I, \Sigma_{O_2})$, we can still compare their behaviour by setting $\Sigma_O = \Sigma_{O_1} \cup \Sigma_{O_2}$ and noting that $M_1, M_2 \in \text{FSM}(\Sigma_I, \Sigma_O)$. Intuitively speaking, we can always assume that the FSMs under consideration operate on the same output alphabet.

Exercise 2. Consider again Exercise 1. Since the ceiling speed monitor CSM described there operates on floating point values as inputs, it will be infeasible to test every possible input vector from every possible internal state.

A typical intuitive way to reduce the number of inputs to be considered is to build *input equivalence classes*. Such a class is a subset of the input domain where it can be assumed that the SUT will process all inputs from this class in the same way, so that "one input of this class is just as good as any other from the same class", when it comes to detecting faults. Therefore it suffices to select one or a small number of representatives from each class, to be used as test input data in a test suite.

In this exercise, this form of "discretisation" of a system with input domains that are too large is applied in an intuitive, heuristic way. We will see in Part III of these lecture notes that input equivalence classes can be determined in a systematic way, so that even complete test suites can be generated for fault models with practically relevant fault domains.

The objective for this exercise is to create a discrete DFSM model $M = (S, \underline{q}, \Sigma_I, \Sigma_O, h)$ for the CSM, where the input alphabet contains representatives from input equivalence classes that have been intuitively identified.

The state space S of this DFSM is $S = \{0, 1, 2, 3, 4\}$, where 0 corresponds to internal state `NORMAL`, 1 to `OVERSPEED` and so on, such tat 4 corresponds to `EMER_BRAKE_INTERVENTION`. The initial state is $\underline{q} = 0$.

The output alphabet is of the form $\Sigma_O = \{0, 1, 2, 3, 4\}$, where these integer codes correspond to the following output vectors.

output code	**svcBrake**	**emerBrake**	**d**
0	0	0	OK
1	0	0	OVR
2	0	0	WRN
3	1	0	SI
4	1	1	SI

The input alphabet $\Sigma_I = \{0, 1, 2, 3, \dots\}$ needs to be constructed in such a way that that each $x \in \Sigma_I$ corresponds to an input equivalence class. Each class restricts the values for $v, vMax$, and c. For example,

$$\begin{aligned} \text{class}_i \;=\; & \{(v, vMax, c) \mid v \in (vMax + dW(vMax), vMax + dSI(vMax)] \wedge \\ & vMax \in (110, 140] \wedge c\} \end{aligned}$$

could be such an equivalence class, but you need to justify for each class why it is appropriate. Identify at least 5 classes $\text{class}_0, \dots, \text{class}_4$, so that the input alphabet corresponds to classes as follows.

input code	**class**
0	class_0
1	class_1
2	class_2
3	class_3
4	class_4
...	...

With these alphabet definitions, the DFSM's transition relation can be represented in tabular form.

- The table's first column lists the input codes.
- The table's first row lists the internal states.
- A table entry in the field for input n and state q contains the expression y/q', if and only if the CSM, when in internal state q and when getting input n, transits to state q' while producing output y.

The resulting transition table should look like this, depending on your input equivalence class definitions.

input \ state	0	1	2	3	4
0	0/0	0/0	0/0	0/0	0/0
2	1/1	1/1	2/2	3/3	4/4
...	...	...	...	...	...
7	0/0	0/0	0/0	0/0	4/4
...	...	...	...	...	...

For the test execution, change the test harness (main program of the test) in such a way that each time an input represented by some class class_I is to be selected, the test harness selects a random value from class_I. □

Conformance Relations Two FSM M_1, M_2 are *I/O-equivalent (*$M_1 \sim M_2$*)* if and only if their languages coincide, i.e. $L(M_1) = L(M_2)$. FSM M_1 is a *reduction of* M_2 *(*$M_1 \preceq M_2$*)*, if and only if $L(M_1) \subseteq L(M_2)$. I/O-equivalence is also called *trace equivalence* by some authors, see, e.g. [45].

For a set of input traces $A \subseteq \Sigma_I^*$, two states s_1, s_2 are *A-equivalent* (written $s_1 \overset{A}{\sim} s_2$), if all input sequences from A, when applied to s_1 and s_2, lead to the same sets of outputs. More formally,

$$s_1 \overset{A}{\sim} s_2 \equiv \left(\forall \overline{x} \in A, \overline{y} \in \Sigma_O^* : \overline{x}/\overline{y} \in L(s_1) \Leftrightarrow \overline{x}/\overline{y} \in L(s_2)\right)$$

Let Σ_I^k denote all input traces of length $k \geq 0$. For $k = 0$, $\Sigma_I^0 = \{\varepsilon\}$, where ε denotes the empty trace. It is easy to see that for $k > 0$, $s_1 \overset{\Sigma_I^k}{\sim} s_2$ implies $s_1 \overset{\Sigma_I^{k-1}}{\sim} s_2$. We abbreviate $s_1 \overset{\Sigma_I^k}{\sim} s_2$ with $s_1 \overset{k}{\sim} s_2$ and call this k-*equivalence*. I/O-equivalence obviously coincides with Σ^*-equivalence.

Isomorphisms FSM homomorphisms map states to states and alphabets to alphabets, such that these mappings respect the transition relations. In the special case of two FSMs $M_i = (Q_i, \underline{q_i}, \Sigma_I, \Sigma_O, h_i), i = 1, 2$ over the same input/output alphabets, a bijective function $f : Q_1 \to Q_2$ induces an (FSM) *isomorphism* $M_1 \to M_2$, if and only if

1. $f(\underline{q_1}) = \underline{q_2}$
2. $\forall q, q' \in Q_1, x \in \Sigma_I, y \in \Sigma_O : h_1(q, x, y, q') \Leftrightarrow h_2(f(q), x, y, f(q'))$

It is easy to see that the existence of an isomorphism implies I/O-equivalence between M_1 and M_2, since the stronger result $\forall q \in Q_1 : L(q) = L(f(q))$ holds.

Homomorphisms More generally, a *homomorphism* between FSMs $M = (Q, \underline{q}, \Sigma_I, \Sigma_O, h)$ and $M' = (Q', \underline{q}', \Sigma_I', \Sigma_O', h')$ is a triple (ζ, ξ, η) of functions $\zeta : Q \to Q'$, $\xi : \Sigma_I \to \Sigma_I'$, and $\eta : \Sigma_O \to \Sigma_O'$, such that these functions respect the transition relations h, h' in the following sense.

$$\begin{aligned}
&\forall q \in Q, x \in \Sigma_I : \\
&\quad \{(y', w') \in \Sigma_O' \times Q' \mid h'(\zeta(z), \xi(x), y'w')\} = \\
&\qquad \{(y', w') \in \Sigma_O' \times Q' \mid \exists y \in \Sigma_O, w \in Q : \\
&\qquad\qquad h(z, x, y, w) \wedge \eta(y) = y' \wedge \zeta(w) = w'\}
\end{aligned}$$

Parallel Composition by Intersection FSM can be composed in parallel by synchronising over common input/output events: let FSMs $M_i = (Q_i, \underline{q_i}, \Sigma_I, \Sigma_O, h_i), i = 1, 2$ be defined over the same input/output alphabets. Then the *intersection of* M_1, M_2 (also called the *product of* M_1, M_2) is specified by

$$M_1 \cap M_2 = (Q_1 \times Q_2, (\underline{q_1}, \underline{q_2}), \Sigma_I, \Sigma_O, h)$$

where the transition relation is specified by

$$h((q_1, q_2), x, y, (q_1', q_2')) \Leftrightarrow h_1(q_1, x, y, q_1') \wedge h_2(q_2, x, y, q_2')$$

By construction, $L(M_1 \cap M_2) = L(M_1) \cap L(M_2)$. Every execution

$$(\underline{q_1}, \underline{q_2}) \xrightarrow{x_1/y_1} (q_1^1, q_2^1) \xrightarrow{x_2/y_2} \ldots \xrightarrow{x_k/y_k} (q_1^k, q_2^k)$$

of $M_1 \cap M_2$ is composed of executions

$$\underline{q_1} \xrightarrow{x_1/y_1} q_1^1 \xrightarrow{x_2/y_2} \ldots \xrightarrow{x_k/y_k} q_1^k \text{ of } M_1 \text{ and } \underline{q_2} \xrightarrow{x_1/y_1} q_2^1 \xrightarrow{x_2/y_2} \ldots \xrightarrow{x_k/y_k} q_2^k \text{ of } M_2$$

Minimality and Observability An FSM is *minimal*, if all of its states are reachable and each pair of different states q, q' produce different languages, that is, $L(q) = L(q') \Rightarrow q = q'$. An FSM M is called *observable*, if for any states q, q_1, q_2, input $x \in \Sigma_I$, and output $y \in \Sigma_O$,

$(q, x, y, q_1), (q, x, y, q_2) \in h$ implies $q_1 = q_2$. This means that in case of nondeterministic transitions being triggered by input x in some state q, the transition that has actually been taken can be determined by means of the output observed. Deterministic FSMs are observable. Every FSM is equivalent to an observable FSM [66]. An algorithm for transforming general NFSM into equivalent observable ones is given in [45, Appendix II]. For any $(q, \overline{x}, \overline{y}, q') \in h$, since observability guarantees that q' is uniquely determined by q, $\overline{x}$, and $\overline{y}$, we denote the state q' also by q-after-$\overline{x}/\overline{y}$. An I/O-trace is a finite sequence $\overline{x}/\overline{y} = (x_1, y_1) \dots (x_k, y_k)$ of input-output pairs with $\overline{x} = x_1 \dots x_k \in \Sigma_I^*$ and $\overline{y} = y_1 \dots y_k \in \Sigma_O^*$. The length of an I/O-trace $\overline{x}/\overline{y}$, is the length of its input part and equals the length of its output part: $|\overline{x}/\overline{y}| = |\overline{x}| = |\overline{y}|$. We assume that all states are reachable from the initial state $\underline{q}$ (*initial connected*) and that there is a well-defined reset operation allowing to restart the machine from its initial state. The state spaces of two equivalent minimal nondeterministic FSMs may have different sizes (see Example 1 below). However, equivalent *observable*, minimal FSMs are isomorphic, and therefore have the same number of states.

Lemma 3.1 *Let* $\overline{M} = (Q, \underline{q}, \Sigma_I, \Sigma_O, h), \overline{M}' = (Q', \underline{q}', \Sigma_I, \Sigma_O, h')$ *be two observable, minimal, nondeterministic FSMs over the same input alphabet and output alphabet. If* $\overline{M}, \overline{M}'$ *are equivalent, i.e.,* $L(\overline{M}) = L(\overline{M}')$ *then* $\overline{M}$ *and* $\overline{M}'$ *have the same number of states. Furthermore,* $\overline{M}$ *and* $\overline{M}'$ *are isomorphic.*

Proof. Let Q, Q' be the state space of $\overline{M}, \overline{M}'$, respectively. Since $\overline{M}, \overline{M}'$ are minimal, Q and Q' contain only reachable states, and for any $q_1, q_2 \in Q, q_1', q_2' \in Q'$, $L(q_1) = L(q_2)$ implies $q_1 = q_2$ and $L(q_1') = L(q_2')$ implies $q_1' = q_2'$.

For any $q \in Q$, there exists $\overline{x}/\overline{y} \in L(\underline{q})$ such that $\underline{q}$-after-$\overline{x}/\overline{y} = q$. Since $L(\overline{M}) = L(\overline{M}')$, $L(\underline{q}) = L(\underline{q}')$, $\overline{x}/\overline{y} \in L(\underline{q}')$ and $L(q') = L(q)$, where $\underline{q}'$-after-$\overline{x}/\overline{y} = q'$. Since $\overline{M}'$ is minimal, q' is the unique state of Q' with $L(q') = L(q)$. Similarly, for any $q' \in Q'$ there exists a unique $q \in Q$ such that $L(q) = L(q')$. Hence $f : Q \to Q' : q \mapsto q'$ with $L(q') = L(q)$ is a bijection and Q, Q' have the same number of states.

Since $L(\underline{q}) = L(\overline{M}) = L(\overline{M}') = L(\underline{q}')$, we get $f(\underline{q}) = \underline{q}'$. Then for any $(q_1, x, y, q_2) \in h$, $x/y \in L(q_1) = L(f(q_1))$, there is a unique $q' \in Q'$ such that $(f(q_1), x, y, q') \in h'$, since $\overline{M}'$ is observable. From $L(q_2) = \{\overline{x}/\overline{y} \in \Sigma_I^*/\Sigma_O^* \mid x.\overline{x}/y.\overline{y} \in L(q_1)\} = \{\overline{x}/\overline{y} \in \Sigma_I^*/\Sigma_O^* \mid x.\overline{x}/y.\overline{y} \in L(f(q_1))\} = L(q')$,

$f(q_2) = q'$ and $(f(q_1), x, y, f(q_2)) = (f(q_1), x, y, q') \in h'$ hold. Hence $\overline{M}$ and $\overline{M}'$ are isomorphic. □

Example 1. Let $M = (Q = \{\underline{q}, q_1, q_2, q_3\}, \underline{q}, I = \{a, b\}, O = \{c, d\}, h)$,$M' = (Q' = \{\underline{q}', q_1', q_2'\}, \underline{q}', I, O, h')$ as given in Fig. 3.1. M, M' are minimal and I/O-equivalent, but M is not observable, M' is observable, and $|Q| > |Q'|$. □

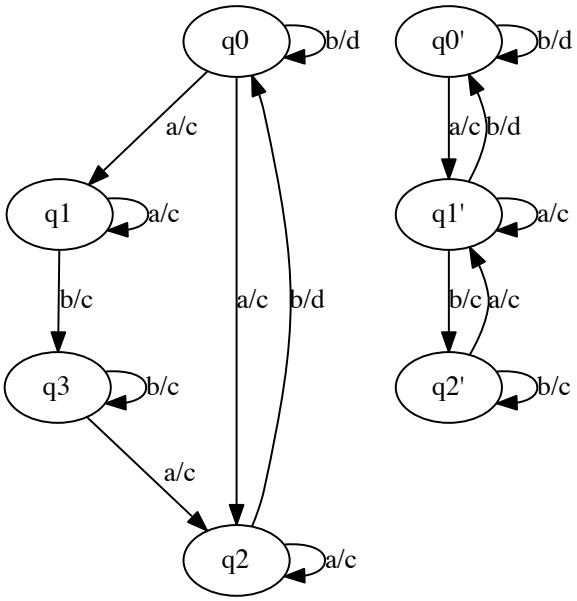


Figure 3.1: Equivalent minimal FSMs from Example 1 with $|Q| > |Q'|$.

We adopt the following definition introduced in [45]: given an FSM M, its *prime machine* is the minimal, observable FSM prime(M) satisfying $L(\text{prime}(M)) = L(M)$. As discussed above, the prime machine associated with an arbitrary FSM is uniquely determined up to isomorphism. For DFSMs, the associated prime machines are simply their minimised equivalents.

3.3 Minimisation of DFSMs

The minimisation algorithm for deterministic FSMs described in this section goes back to [21, pp. 60].

3.3.1 Transition Table Representation of DFSMs

In [21, p. 50], the following tabular representation of FSMs is introduced.

	Inputs → Outputs (I2O)			**Inputs → Post-state q' (I2P)**		
q	x_1	x_2	...	x_1	x_2	...
$\underline{q}$	y_{01}	y_{02}	...	q_{01}	q_{01}	...
q_1	y_{11}	y_{12}	...	q_{11}	q_{11}	...
q_2	y_{21}	y_{22}	...	q_{21}	q_{21}	...
...	...	...	...	...	...	...
q_k	y_{k1}	y_{k2}	...	q_{k1}	q_{k1}	...

The DFSM states $\underline{q}, q_1, \ldots, q_k$ are listed in the first column of the table. The other table columns are structured into an **Inputs → Outputs** section and an **Inputs → Post-state q'** section. In each of these sections, the columns are labelled with the inputs $x_1, x_2 \ldots$ of the input alphabet Σ_I.

In the **Inputs → Outputs** section (called *I2O-table* in the following), the table entry identified by state q_i and input x_j contains the associated output $y_{ij} \in \Sigma_O$, which is produced when input x_j is applied when residing in q_i.

In the **Inputs → Post-state q'** section (called *I2P-table* in the subsequent sections), the table entry identified by state q_i and input x_j contains the associated post state $q_{ij} \in \Sigma_O$, which is reached when input x_j is applied to the DFSM when residing in state q_i.

While this representation is obviously well-suited for DFSMs, it is unsuitable for nondeterministic FSMs, because there input x_j could lead to different outputs and post-states, when applied in some pre-state q_i. This would lead to many redundant table entries, because *all* inputs have to be handled in each line.

Fig. 3.2 shows a DFSM in the usual graph representation; its associated transition table is shown in Table 3.1.

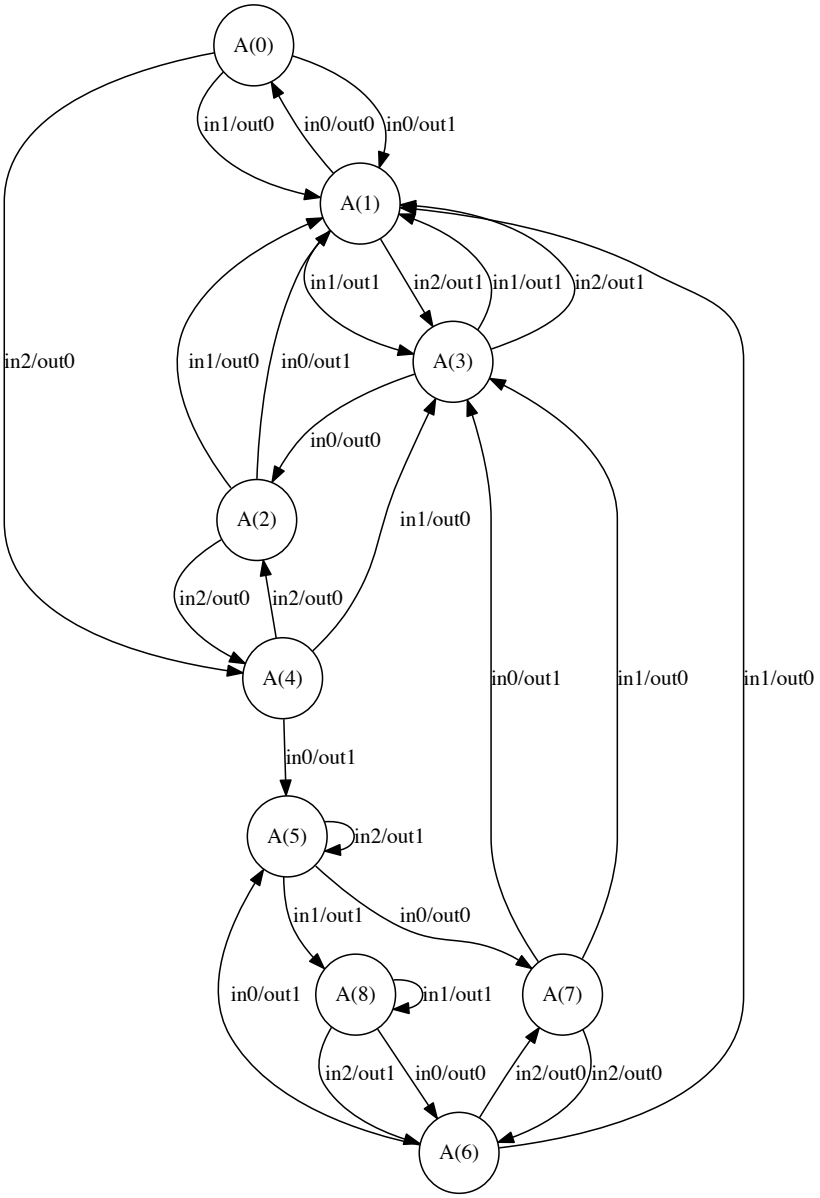


Figure 3.2: DFSM to be minimised (example based on [21, pp. 55]).

	I2O			**I2P**		
q	0	1	2	0	1	2
0	1	0	0	1	1	4
1	0	1	1	0	3	3
2	1	0	0	1	1	4
3	0	1	1	2	1	1
4	1	0	0	5	3	2
5	0	1	1	7	8	5
6	1	0	0	5	1	7
7	1	0	0	3	3	6
8	0	1	1	6	8	6

Table 3.1: Transition table of the DFSM from Fig. 3.2.

3.3.2 DFSM Minimisation With P_k Tables

Following [21, pp. 60], a P_k table extends the **Inputs** $\rightarrow$ **Post-state** q' (I2P-table) section of a transition table by an additional left-hand side column denoting the k-equivalence class (see definition in Section 3.2) every original state resides in.

The Transition table can be regarded as the "P_0-table", where all states reside in the same equivalence class $[\mathsf{q}] = 0$, because they cannot be distinguished by an input trace of length zero. The transition table from Fig. 3.1 would look like this in the P_0-table notation.

$\mathbf{P_0}$

		I2O			**I2P**		
[q]	**q**	0	1	2	0	1	2
0	0	1	0	0	1	1	4
0	1	0	1	1	0	3	3
0	2	1	0	0	1	1	4
0	3	0	1	1	2	1	1
0	4	1	0	0	5	3	2
0	5	0	1	1	7	8	5
0	6	1	0	0	5	1	7
0	7	1	0	0	3	3	6
0	8	0	1	1	6	8	6

For $k = 0, 1, \ldots,\ k < |Q|$, the P_{k+1}-table is created from the P_k-table as shown in the next paragraphs. Each P_k-table assigns all k-equivalent states to the same state equivalence class. The algorithm terminates, as soon as the transformation of the P_k-table into the P_{k+1}-table does not yield any additional state equivalence class. Since a DFSM with $|Q|$ states has at most $|Q|$ state equivalence classes (the number of classes is $|Q|$ if and only if the DFSM is minimised), the algorithm terminates after at most $(|Q| - 1)$ steps. The last P_k-table identifies the state equivalence classes of the minimised DFSM.

Algorithm – Part A. Create P_1-table from the transition table P_0.

- **Input** to the algorithm: DFSM transition table P_0
- **Output** of the algorithm: table P_1
- Create initial version of P_1 as copy of P_0, and reset all state equivalence classes [q] in P_1 to -1, meaning 'undefined'
- Assign all states whose outgoing transitions have identical I/O-labels (that is, whose I2O-rows coincide) into the same equivalence classes, using the following algorithm
  1. Set class counter c to 0
  2. While class counter $c < |Q|$ do

(a) Find smallest row r in P_1-table with $r \geq c$ and $[r] = -1$ (i.e. $[r]$ is undefined). If no such r can be found terminate by returning P_1

(b) Set $[r] = c$

(c) Set $u = r + 1$

(d) While $u < |Q|$ do

i. If $[u] = -1$ (i.e. row u is not yet associated with a state class) and the I2O-table shows the same outputs in row u as for row r, then set $[u] = c$, that is, state u is inserted into the same 1-equivalence class as row r.

ii. Increment u by 1

3. Increment c by 1

According to this algorithm, the P_1-table associates all 1-equivalent states in the same $[q]$-class. For the P_0 table shown above, the P_1-table generated by this algorithm looks as follows.

P_1

[q]	**q**	**I2P**		
		0	1	2
0	0	1	1	4
1	1	0	3	3
0	2	1	1	4
1	3	2	1	1
0	4	5	3	2
1	5	7	8	5
0	6	5	1	7
0	7	3	3	6
1	8	6	8	6

Algorithm – Part B. Create P_{k+1}-table from the P_k-table for $i \geq 1$.

- **Input** to the algorithm: DFSM transition table P_k
- **Output** of the algorithm: table P_{k+1}
- Create initial version of P_{k+1} as copy of P_k, and reset all state equivalence classes $[q]$ in P_{k+1} to -1, meaning 'undefined'

- Assign all P_k-equivalent states whose outgoing transitions have P_k-equivalent post-states into the same P_{k+1}-equivalence classes, using the following algorithm

  1. Set class counter c to 0
  2. While at least one row r in P_{k+1} is still marked with $[r] = -1$ do
     (a) Find smallest row r in P_{k+1}-table with $r \geq c$ and $[r] = -1$ (i.e. $[r]$ is undefined).
     (b) Set $[r] = c$
     (c) Set $u = r + 1$
     (d) While $u < |Q|$ do
        i. If $[u] = -1$ (i.e. row u is not yet associated with a state class), r and u are P_k-equivalent, and the I2P-table shows P_k-equivalent post-states in row u and row r, then set $[u] = c$, that is, state u is inserted into the same $P_{(k+1)}$-equivalence class with number c as row r.
        This condition is formalised as

$$[u] = -1 \wedge (P_k.[q])[r] = (P_k.[q])[u] \wedge$$
$$\big(\forall x \in \Sigma_I : (P_k.[\mathbf{q}])[\mathbf{I2P}[r][x]] = (P_k.[\mathbf{q}])[\mathbf{I2P}[u][x]]\big)$$

        ii. Increment u by 1
     (e) Increment c by 1
  3. Terminate by returning P_{k+1}

The following table show the application of this algorithm to the P_1-table and its successors. Fig. 3.3 shows the minimised state machine version of the DFSM in Fig. 3.2. The minimised machine is derived from table P_4.

$$\mathbf{P}_2$$

[q]	q	I2P		
		0	1	2
0	0	1	1	4
1	1	0	3	3
0	2	1	1	4
1	3	2	1	1
0	4	5	3	2
1	5	7	8	5
0	6	5	1	7
0	7	3	3	6
2	8	6	8	6

$\mathbf{P}_3$

[q]	q	I2P		
		0	1	2
0	0	1	1	4
1	1	0	3	3
0	2	1	1	4
1	3	2	1	1
0	4	5	3	2
3	5	7	8	5
0	6	5	1	7
0	7	3	3	6
2	8	6	8	6

$\mathbf{P}_4$

[q]	q	I2P		
		0	1	2
0	0	1	1	4
1	1	0	3	3
0	2	1	1	4
1	3	2	1	1
4	4	5	3	2
3	5	7	8	5
4	6	5	1	7
0	7	3	3	6
2	8	6	8	6

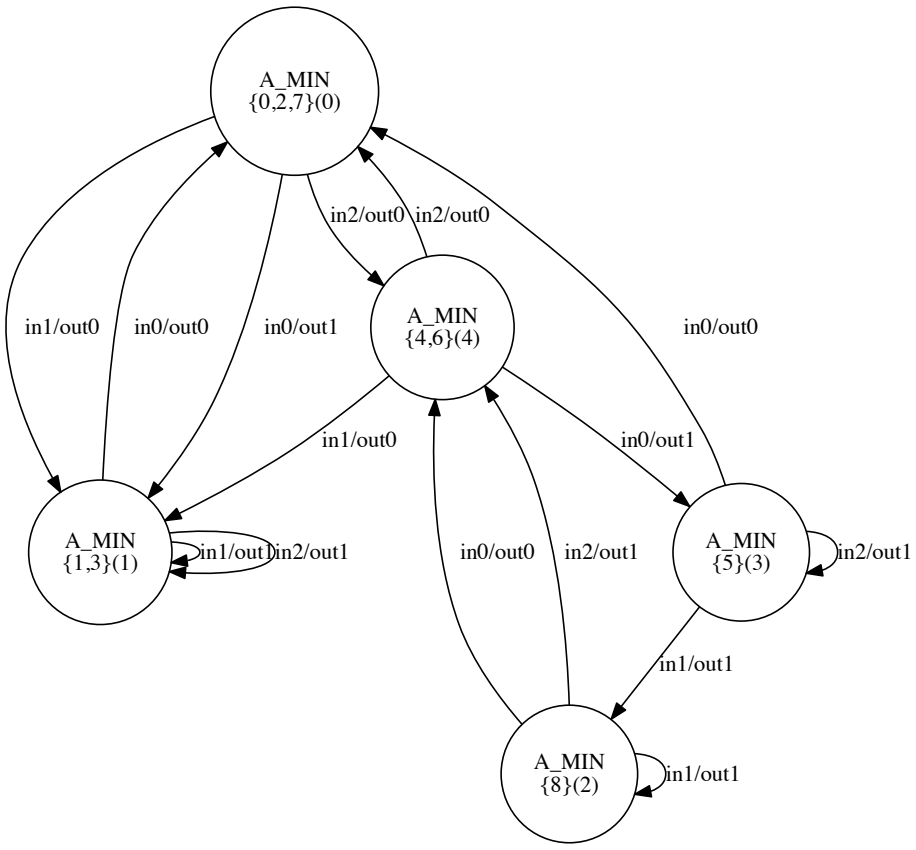


Figure 3.3: Minimised DFSM associated with DFSM in Fig. 3.2.

3.4 Characterisation Sets for DFSMs

The state covers and transition covers introduced above in Section 4.2 represent sets of input traces reaching every possible FSM state and transition, respectively. So with these two at hand, every FSM state can be checked for completeness, and every transition of an FSM can be checked with respect to output faults. What remains to be checked when aiming at proving I/O-equivalence, is that the target state q' reached under a certain SUT transition

is equivalent to the expected state q that is reached by the corresponding transition in the reference model.

Definition 3.2 (Characterisation Set) *Given an FSM* $M = (Q, \underline{q}, \Sigma_I, \Sigma_O, h)$*, a* characterisation set *for* M *is a set* $W \subseteq \Sigma_I^*$ *which is able to distinguish all non-equivalent states of* M*:*

$$\forall q_1, q_2 \in Q : q_1 \overset{W}{\sim} q_2 \Leftrightarrow q_1 \sim q_2$$

□

Definition 3.2 specifies a characterisation set W by the requirement that two states are I/O-equivalent if and only if every input sequence in W leads to the same outputs, when applied to both states. An alternative formulation of this property is

$$\begin{aligned}&\forall q_1, q_2 \in Q :\\ &\quad L(q_1) = L(q_2) \Leftrightarrow (\forall \overline{x} \in W, \overline{y} \in \Sigma_O^* : \overline{x}/\overline{y} \in L(q_1) \Leftrightarrow \overline{x}/\overline{y} \in L(q_2)),\end{aligned}$$

and another alternative is

$$\begin{aligned}&\forall q_1, q_2 \in Q :\\ &\quad L(q_1) \neq L(q_2) \Rightarrow (\exists \overline{x} \in W, \overline{y} \in \Sigma_O^* : (\overline{x}/\overline{y} \in L(q_1) \wedge \overline{x}/\overline{y} \notin L(q_2))\\ &\qquad\qquad \vee (\overline{x}/\overline{y} \notin L(q_1) \wedge \overline{x}/\overline{y} \in L(q_2))).\end{aligned}$$

The following lemma suggests a very simple method to construct a characterisation set, as soon as at least one distinguishing trace has been found for some pair of distinguishable states. Originally, this lemma and an associated proof have been stated in [9, Lemma 0]. Our version here has been slightly revised.

Lemma 3.2 *Let* $M = (Q, \underline{q}, \Sigma_I, \Sigma_O, h)$ *be a completely specified minimal DFSM and* $W \in \Sigma_I^*$ *a set of input traces partitioning* Q *into* k *state classes with* $0 < k < n$*. Then* $W \cup \Sigma_I.W$ *partitions* Q *into at least* $k + 1$ *classes.*

Proof. Two states are in the same class if and only if both produce the same outputs for all input traces from W. Now select two states q_1, q_2 from the same class and an input $x \in \Sigma_I$ such that $q_1' = q_1$-after-x and $q_2' = q_2$-after-x reside in *different* classes. Such an x always exists, since all states in Q

are distinguishable.[1] For these two classes, select an element $\overline{w} \in W$ such that different outputs occur if $\overline{w}$ is applied to q_1' or q_2', respectively. Then $x.\overline{w} \in \Sigma_I.W$ distinguishes at least q_1 and q_2, and these were indistinguishable before. Therefore, the state space has been partitioned into at least one additional class by adding the input traces from $\Sigma_I.W$ to W. This completes the proof. □

The following algorithm presented in [21, p. 92] can be used to determine a smaller characterisation set than that suggested by Lemma 3.2. The algorithm applies to an FSM which has been minimised before using P_k tables, as described in Section 3.3.

1. **Inputs.** Minimal DFSM $M = (Q, \underline{q}, \Sigma_I, \Sigma_O, h)$ and the P_k tables calculated for M as described in Section 3.3[2].
2. **Outputs.** Characterisation set $W \subseteq \bigcup_{i=1}^{|Q|-1} \Sigma_I^i$.
3. Initialise $W = \varnothing$.
4. For every pair (q_{i_0}, q_{j_0}) with $q_{i_0}, q_{j_0} \in Q, q_{i_0} \neq q_{j_0}$, proceed as follows.
   (a) If (q_{i_0}, q_{j_0}) are already distinguished by W, continue with the next pair of states in Step 4.
   (b) Otherwise find $\ell \geq 1$ such that (q_{i_0}, q_{j_0}) are equivalent in $P_{\ell-1}$, but reside in different equivalence classes in P_ℓ. (For the case where (q_{i_0}, q_{j_0}) are immediately distinguished by P_1, the undefined P_0 table is not required.)
   (c) Set $k = 1$.
   (d) If $\ell - k > 0$, proceed to Step 4f.
   (e) If $\ell - k = 0$, set x_k to any element of Σ_I, such that M produces different outputs for x_k, when applied in states $(q_{i_{k-1}}, q_{j_{k-1}})$, respectively. Add $x_1 \ldots x_k$ to W. Continue with Step 4.
   (f) Set x_k to any element of Σ_I, such that $(q_{i_{k-1}}, q_{j_{k-1}})$ are mapped to different state classes under x_k in table $P_{\ell-k}$. Let (q_{i_k}, q_{j_k}) be representatives of these different classes. Increment k by 1 and proceed with Step 4d.

[1] see also section on P_k-table construction.

[2] Note that these are the P_k-tables calculated for M, which is already minimal, and *not* the P_k tables constructed during the minimisation process resulting in M.

5. Reduce W to its maximal input traces.

6. Return W.

Observe that the "brute force method" based on product automata described in the previous section doesn't need any of the concepts of state cover, transition cover, or characterisation set, because it just depends on the hypothesis about the maximal number m of states contained in the minimal DFSM representing the SUT.

3.5 Transformation to Observable FSMs

Transformation algorithm Recall from Section 3.2 that an FSM is observable, if for any states q, q_1, q_2, input $x \in \Sigma_I$, and output $y \in \Sigma_O$, $(q, x, y, q_1), (q, x, y, q_2) \in h$ implies $q_1 = q_2$. Deterministic FSMs are automatically observable. Nondeterministic FSMs can always be transformed into equivalent observable ones.

The following theorem shows the basic principle, how an equivalent observable FSM M' can be constructed from the original FSM M by creating M'-states corresponding to sets of M-states that are reachable under the same input/output traces.

Theorem 3.1 *Let* $M = (Q, \underline{q}, \Sigma_I, \Sigma_O, h)$ *be a finite state machine. Let* $M' = (Q', \underline{q}', \Sigma_I, \Sigma_O, h')$ *be a finite state machine defined by*

1. $Q' = \{[\overline{x}/\overline{y}] \mid \overline{x}/\overline{y} \in L(M)\}$, *where* $[\overline{x}/\overline{y}] = \{q \in Q \mid (\underline{q}, \overline{x}, \overline{y}, q) \in h\}$ *for all* $\overline{x}/\overline{y} \in L(M)$.

2. $\underline{q}' = [\varepsilon]$

3. $h' = \{([\overline{x}/\overline{y}], x, y, [\overline{x}'/\overline{y}']) \mid \overline{x}/\overline{y}, \overline{x}'/\overline{y}' \in L(M) \wedge \overline{x}.x/\overline{y}.y = \overline{x}'/\overline{y}'\}$

Then M' *is initial connected, observable and* $L(M') = L(M)$. *Furthermore,* M' *is completely specified if* M *is completely specified.*

Proof. M' is observable; this follows directly from the definition of h', because $[\overline{x}.x/\overline{y}.y]$ is the only possible post-state of $[\overline{x}/\overline{y}]$ under input/output x/y. Extending h' in the natural way to a subset of $Q' \times \Sigma_I^* \times \Sigma_O^* \times Q'$, results in

$$h' = \{([\overline{x}_1/\overline{y}_1], \overline{x}, \overline{y}, [\overline{x}_2/\overline{y}_2]) \mid \overline{x}_i/\overline{y}_i \in L(M), i = 1, 2, \wedge \overline{x}_1.\overline{x}/\overline{y}_1.\overline{y} = \overline{x}_2/\overline{y}_2\}.$$

This implies $L(M') = L(M)$. Furthermore,

$$(\underline{q}', \overline{x}, \overline{y}, q') \in h' \Leftrightarrow \overline{x}/\overline{y} \in L(M) \wedge q' = [\overline{x}/\overline{y}].$$

As a consequence, according to the definition of Q', every state in Q' is reachable, so M' is initial connected.

Let $[\overline{x}/\overline{y}] \in Q'$ be any state and $x \in \Sigma_I$ any input. Then $\overline{x}/\overline{y} \in L(M)$. Suppose M is completely specified. Then there exists $y \in \Sigma_O$ such that $\overline{x}.x/\overline{y}.y \in L(M)$. Hence $([\overline{x}/\overline{y}], x, y, [\overline{x}.x/\overline{y}.y]) \in h'$ and M' is completely specified, too. □

The following algorithm originally presented in [45] can be used to implement the transformation method described in Theorem 3.1.

1. **Input.** An FSM $M = (Q, \underline{q}, \Sigma_I, \Sigma_O, h)$
2. **Output.** An observable FSM $M' = (Q', \underline{q}', \Sigma_I, \Sigma_O, h')$ satisfying $L(M') = L(M)$
3. Construct a new FSM $M' = (Q', \underline{q}', \Sigma_I, \Sigma_O, h')$ by setting
   - $\underline{q}'$:= FSM state labelled by label($\underline{q}'$) := $\{\underline{q}\}$ and set $\underline{q}'$'s colour to 'white'
   - Initialise $Q' := \{\underline{q}'\}$
   - Initialise $h' := \varnothing$
4. While there still exists an FSM state $q' \in Q'$ with colour 'white'
   (a) Select white node $q' \in Q'$
   (b) Mark q' as 'black'
   (c) For each $(x, y) \in \Sigma_I \times \Sigma_O$
   - Set label $\ell := \{q_2 \in Q \mid \exists q_1 \in \text{label}(q') : h(q_1, x, y, q_2)\}$
   - If $\ell = \varnothing$ continue with (4c)
   - If a node $q'' \in Q'$ with label(q'') = ℓ exists, then set $h' := h' \cup \{(q', x, y, q'')\}$
   - Otherwise create a new node q'' with label(q'') := ℓ, mark q'' with colour 'white', and set $Q' := Q' \cup \{q''\}$ and $h' := h' \cup \{(q', x, y, q'')\}$

5. Return the FSM M'; it is observable and fulfils $\mathsf{L}(\mathsf{M}') = \mathsf{L}(\mathsf{M})$

Lemma 3.3 *The algorithm above terminates, and the FSM* M' *created by the algorithm specified above is initial connected, observable, and satisfies* $\mathsf{L}(\mathsf{M}') = \mathsf{L}(\mathsf{M})$.

Proof. The termination of the algorithm follows from the fact that the while-loop (4) terminates as soon as no white Q'-states exist anymore. The algorithm, however, creates at most $2^{|\mathsf{Q}|}$ states for Q', because each state is uniquely labelled with a subset of nodes from Q. Moreover, every node in Q' is processed only once during the while-loop execution, because it is marked as 'black' when selected (Step (4b) of the algorithm).

It is straightforward to see that the algorithm generates Q', $\underline{\mathsf{q}}'$, and h' as specified by Theorem 3.1. □

Complexity considerations It is important to note, that in the worst case the observable FSM M' created by the algorithm above has $2^{|\mathsf{Q}|}$ states, because every subset of M-states can appear as a label of states in M'. In practise, however, the observable machine M' usually has far less states.

Example 2. In Fig. 3.4, a non-observable FSM is shown. Applying the algorithm above, its observable equivalent is presented in Fig. 3.5. □

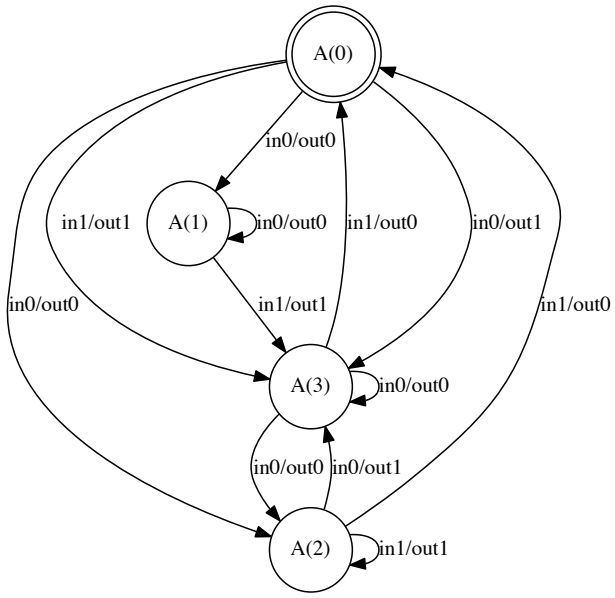


Figure 3.4: Nondeterministic and non-observable FSM for Example 2.

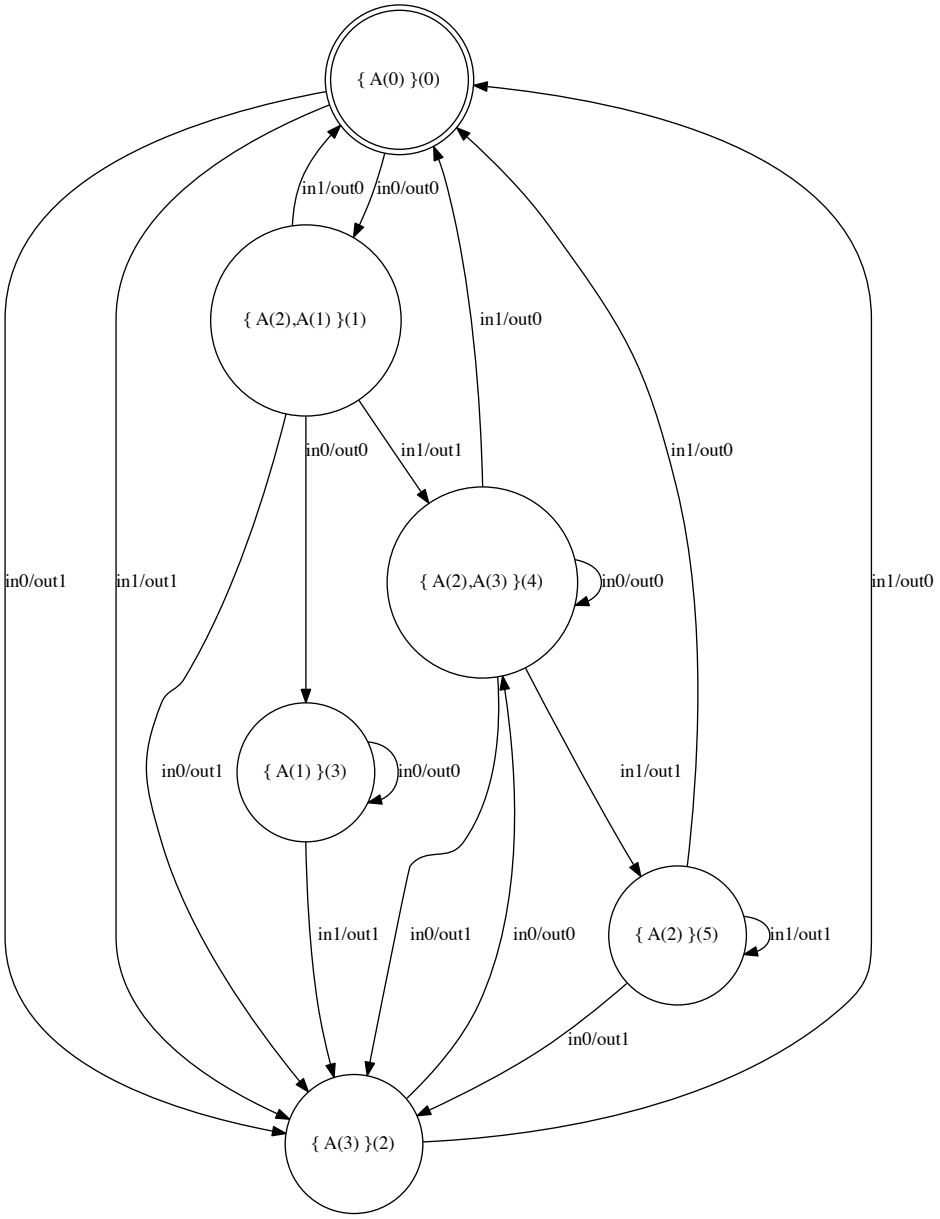


Figure 3.5: Nondeterministic observable FSM equivalent to the FSM in Fig. 3.4.

Abstraction to FSMs used in language theory It is interesting to note that observable nondeterministic FSMs as discussed here can be abstracted to *deterministic* FSMs typically used in language theory: there, FSM transitions are just labelled by a single element of the language alphabet, instead of distinguishing between input and output alphabets as we do it here for the purpose of testing. To construct such an abstraction, let $M = (Q, \underline{q}, \Sigma_I, \Sigma_O, h)$ be an observable NFSM. Now proceed as follows.

1. Let $\Sigma = \Sigma_I \times \Sigma_O$ be the the alphabet of the new "language" FSM M_L to be constructed.
2. Set $Q_L = Q \cup \{q^*\}$ such that $q^* \notin Q$, and define the set of *accepting states* of M_L to be $Q \subseteq Q_L$.
3. Let $\underline{q}$ be the initial state of M_L.
4. Define the transition relation of M_L by

$$\begin{aligned} h_L = \{&(q, (x, y), q') \in Q_L \times \Sigma \times Q_L) \mid \\ &(q, x, y, q') \in h \vee ((q, x, y, q') \notin h \wedge q' = q^*)\} \end{aligned}$$

Intuitively speaking, every transition $q \xrightarrow{x/y} q'$ in M gives rise to a transition $q \xrightarrow{(x,y)} q'$ in M_L. Moreover, every I/O x/y that is *not* associated with any transition emanating from q in M gives rise to a transition $q \xrightarrow{(x,y)} q^*$ in M_L leading to the only non-accepting state in Q_L.

Since M is supposed to be observable, there is at most one transition emanating from $q \in Q$ and labelled by x/y. In M_L, we enforce completeness by adding transitions to $q \xrightarrow{(x,y)} q^*$ for any other I/O-pair which does not occur in any M-transition from q. Trivially, M_L is completely defined and deterministic.

These considerations motivate the construction of the minimisation algorithm for observable FSMs which is presented in the next section.

3.6 Minimisation of Nondeterministic FSMs

For NFSMs, a minimisation algorithm can be designed that is quite similar to the P_k-table algorithm introduced for DFSM minimisation in Section 3.3, but

- the algorithm depends on the NFSM being observable (for this we have the algorithm defined in Section 3.5), and
- the table structure needs more storage, since the columns may no longer be indexed by inputs only (in the DFSM case, the table size is $O(|Q| \cdot |\Sigma_I|)$), but need to be indexed by all I/O-pairs in the cross product $|\Sigma_I \times \Sigma_O|$. As a consequence, the required storage is $O(|Q| \cdot |\Sigma_I| \cdot |\Sigma_O|)$.

OFSM-Tables We call the tables involved *OFSM-Tables*. The first one – OFSM-Table 0 – represents the OFSM as shown in the following table which represents the OFSM in Fig. 3.6: an OFSM-Table has one row per OFSM state. The second column lists the state numbers, and the first column the equivalence class the state is residing in. For OFSM-Table 0, all states reside in the same equivalence class 0. Columns 3, 4, 5, ... are indexed over all $x/y \in \Sigma_I/\Sigma_O$. The table entry for state q in column x/y contains

- the number of the post-state q', if and only if $(q, x, y, q') \in h$, and
- -1 if $(q, x, y, q') \notin h$.

OFSM-Table 0

[q]	q	0/0	0/1	1/0	1/1
0	0	1	3	6	0
0	1	2	2	3	-1
0	2	-1	0	-1	2
0	3	4	5	1	-1
0	4	-1	0	-1	4
0	5	-1	0	-1	2
0	6	7	7	3	-1
0	7	8	8	-1	7
0	8	7	4	6	8

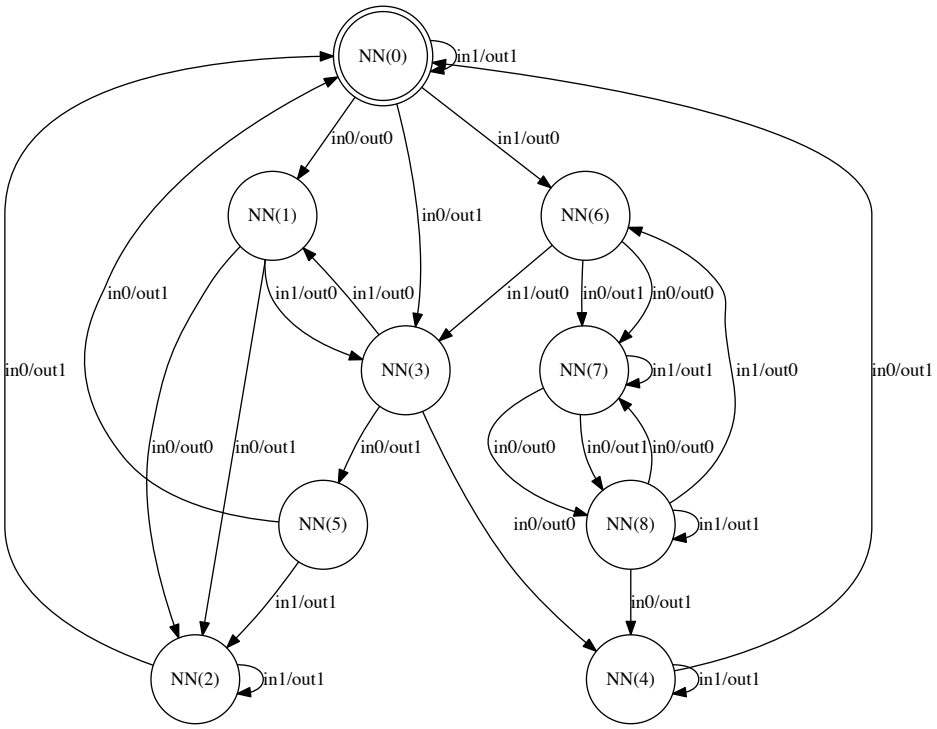


Figure 3.6: Nondeterministic, observable, unminimised FSM.

Construction of OFSM-Table 1 from OFSM-Table 0 Each new OFSM-Table is constructed from its direct predecessor by changing the $[q]$-column only, all other columns remain unchanged. OFSM-Table 1 is constructed from OFSM-Table 0 according to the following rule.

> Two states q_1, q_2 are associated with the same class $[q]$ in OFSM-Table 1, if and only if for every $x/y \in \Sigma_I/\Sigma_O$:
>
> - q_1 has an emanating transition labelled by x/y if and only if q_2 has an emanating transition labelled by x/y, that is,
>
> $$(\exists q_1' : h(q_1, x, y, q_1')) \Leftrightarrow (\exists q_2' : h(q_2, x, y, q_2'))$$

Intuitively speaking, two states are equivalent in OFSM-Table 1, if and only if their (-1)-entries in the x/y-columns are exactly in the same positions.

The following OFSM-Table is derived from OFSM-Table 0 for the OFSM in Fig. 3.6.

OFSM-Table 1

$[q]$	q	0/0	0/1	1/0	1/1
0	0	1	3	6	0
1	1	2	2	3	-1
2	2	-1	0	-1	2
1	3	4	5	1	-1
2	4	-1	0	-1	4
2	5	-1	0	-1	2
1	6	7	7	3	-1
3	7	8	8	-1	7
0	8	7	4	6	8

Construction of OFSM-Table $(j+1)$ from OFSM-Table j for $j \geq 1$
For $j \geq 1$, OFSM-Table $(j+1)$ is generated from OFSM-Table j according to the following rule.

> Two states q_1, q_2 are associated with the same class $[q]$ in OFSM-Table $(j+1)$, if and only if
>
> - q_1, q_2 are already associated with the same class in OFSM-Table j
>
> and for every $x/y \in \Sigma_I/\Sigma_O$:
>
> - if $q_1 \xrightarrow{x/y} q_1'$ and $q_2 \xrightarrow{x/y} q_2'$, then q_1' and q_2' are associated with the same class in OFSM-Table j.
>
> Otherwise class $[q]$ is split in OFSM-Table $(j+1)$, and q_1 and q_2 are associated with different classes in OFSM-Table $(j+1)$.

Below the OFSM-Table 2 generated from OFSM-Table 1 according to the rule above is shown. For example, states 0 and 8 reside in OFSM-Table 1 in the same class $[0]$, but $q_0 \xrightarrow{0/0} q_1$ and $q_8 \xrightarrow{0/0} q_7$ and q_1 resides in class $[1]$ in OFSM-Table 1, while q_7 resides in class $[3]$ in OFSM-Table 1. Therefore states 1 and 8 reside in *different* classes in OFSM-Table 2.

OFSM-Table 2

$[q]$	q	0/0	0/1	1/0	1/1
0	0	1	3	6	0
1	1	2	2	3	-1
2	2	-1	0	-1	2
1	3	4	5	1	-1
2	4	-1	0	-1	4
2	5	-1	0	-1	2
5	6	7	7	3	-1
3	7	8	8	-1	7
4	8	7	4	6	8

Termination condition for OFSM-Table construction If no new equivalence classes $[q]$ are created when constructing OFSM-Table $(j+1)$ from OFSM-Table j, then OFSM-Table j already contains the equivalence class $[q]$ of the minimised OFSM that is equivalent to the original one. The classes $[q]$ are the states of the minimised OFSM, and a transition $[q] \xrightarrow{x/y} [q']$ exists if and only if there are representatives $q_1 \in [q], q_1' \in [q']$, such that $q_1 \xrightarrow{x/y} q_1'$ in the original OFSM.

In our example, OFSM-Table 2 represents the minimised OFSM shown in Fog. 3.7.

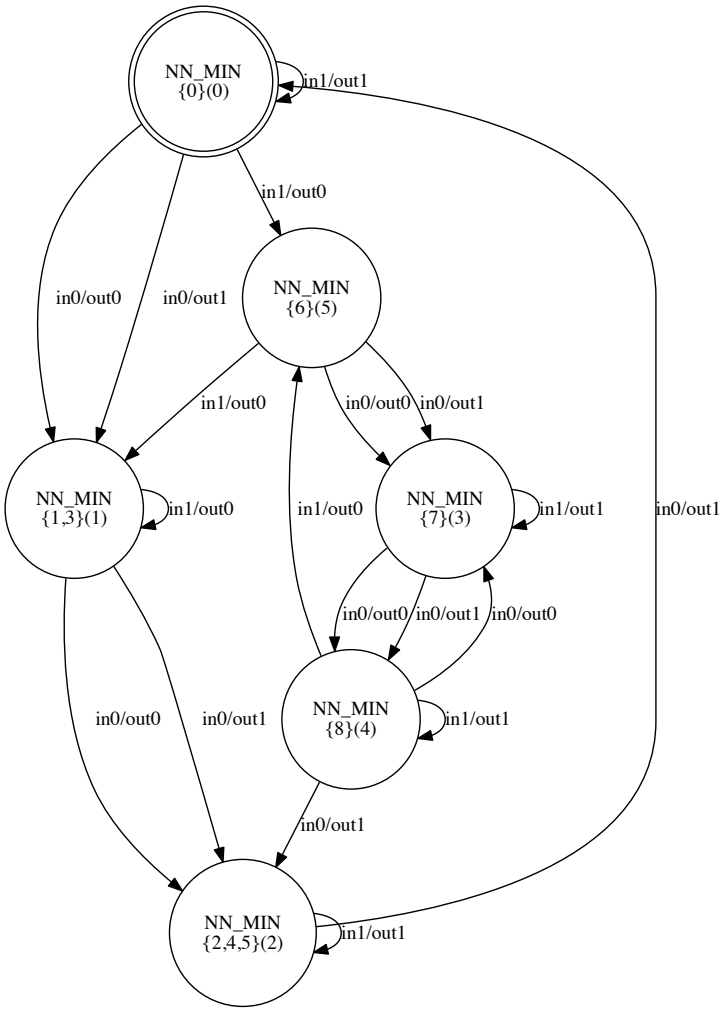


Figure 3.7: Nondeterministic, observable, minimised FSM equivalent to the one depicted in Fig. 3.6.

3.7 Characterisation Set and State Identification Sets of NFSMs

3.7.1 Characterisation set and State Identification Sets for NFSMs

The definition of a characterisation set W (Definition 3.2) applies to both deterministic and nondeterministic FSMs. Algorithms for calculating W, however, differ for the deterministic case (see the algorithm presented in Section 3.4) and the nondeterministic one: while, for example, P_k-tables could be used for DFSMs, these are not available for NFSMs. We therefore present a different algorithm with slightly higher complexity for calculating W in the case of NFSMs in this section. In analogy to the P_k-tables applied for DFSMs, this algorithm is based on a similar evaluation of the OFSM-tables introduced for NFSMs in Section 3.6. Moreover, we are interested in the – potentially smaller – sets W_i of input traces distinguishing just one FSM state q_i from other states.

Definition 3.3 (State Identification Sets) *Let W be a characterisation set for the observable, minimal FSM $M = (Q, q_0, \Sigma_I, \Sigma_O, h)$ with $Q = \{q_0, q_1, q_2, \ldots, q_{|Q|-1}\}$. A set W_i is a* state identification set *for $q_i \in Q$, if it fulfils the conditions*

1. *$W_i \subseteq \Sigma_I^*$, and every element of W_i is a prefix of an element of W.*
2. *W_i distinguishes q_i from all other states in Q, that is,*

$$\forall i \in 0, 1, \ldots (|Q| - 1), q \in Q : q_i \overset{W_i}{\sim} q \Leftrightarrow q_i = q$$

□

Lemma 3.4 *Let $M = (Q, \underline{q}, \Sigma_I, \Sigma_O, h)$ be an observable, minimal and completely specified FSM with n states. Let $W \subseteq \Sigma_I^*$ and $|Q/_{\underset{\sim}{W}}| = k < n$. Then there exists two states $q, q' \in Q$ satisfying $q \overset{W}{\sim} q', q \overset{\Sigma_I.W}{\not\sim} q'$. Consequently, there exists a subset of $\bigcup_{i=0}^{n-k} \Sigma_I^i.W$ which is a characterisation set for M.*

Proof. Since $|Q/_{\underset{\sim}{W}}| = k < n$, there are $q_1, q_2 \in Q$ two distinct but W-equivalent states, that is, $\forall \overline{x} \in W, \overline{y} \in \Sigma_O^* : \overline{x}/\overline{y} \in L(q_1) \Leftrightarrow \overline{x}/\overline{y} \in L(q_1)$.

Since M is minimal, $L(q_1) \neq L(q_2)$, there exists an input sequence $\tau \in \Sigma_I^*$ with the shortest length ≥ 1 satisfying

$$q_1 \overset{\tau.W}{\not\sim} q_2 \tag{3.1}$$

Let $\tau = \overline{x}.x$, $\overline{x} \in \Sigma_I^*, x \in \Sigma_I$. Then $q_1 \overset{\overline{x}}{\sim} q_2$, otherwise, $q_1 \overset{\overline{x}.W}{\not\sim} q_2$, a contradiction to the assumption that $\tau = \overline{x}.x$ is a shortest sequence fulfilling (3.1). In the case $\overline{x} = \varepsilon$, $\tau = x$, $q_1 \overset{\{x\}.W}{\not\sim} q_2$ and $q_1 \overset{W}{\sim} q_2$, there is nothing more to prove.

Suppose $\overline{x} \neq \varepsilon$. Define subset $A \subseteq \Sigma_O^*$ by $A = \{\overline{y} \in \Sigma_O^* \mid \overline{x}/\overline{y} \in L(q_1) \cap L(q_2)\}$. For any $\overline{y} \in A$, since M is observable, define $q_{\overline{y}} := q_1\text{-}\underline{\text{after}}\text{-}\overline{x}/\overline{y}$ and $q'_{\overline{y}} := q_2\text{-}\underline{\text{after}}\text{-}\overline{x}/\overline{y}$. Then $q_{\overline{y}} \overset{W}{\sim} q'_{\overline{y}}$, otherwise $q_1 \overset{\overline{x}.W}{\not\sim} q_2$, again a contradiction to the assumption that $\tau = \overline{x}.x$ is a shortest sequence fulfilling (3.1).

Furthermore, there exist some $\overline{y} \in A$ such that $q_{\overline{y}} \overset{\overline{x}.W}{\not\sim} q'_{\overline{y}}$. Suppose not, $q_{\overline{y}} \overset{\overline{x}.W}{\sim} q'_{\overline{y}}$, for any $\overline{y} \in A$. Then $q_1 \overset{\overline{x}.x.W}{\sim} q_2$, a contradiction to the assumption that $\tau = \overline{x}.x$ and $q_1 \overset{\tau.W}{\not\sim} q_2$. Hence there exist some $\overline{y} \in A$ with

$$q_{\overline{y}} \overset{\Sigma_I.W}{\not\sim} q'_{\overline{y}} \wedge q_{\overline{y}} \overset{W}{\sim} q'_{\overline{y}}$$

□

3.7.2 Algorithm 1. Calculation of W

We will first present an algorithm for calculating a characterisation set for a given observable, minimal FSM. The algorithm is inspired by the one presented for DFSMs in Section 3.4, but it operates on OFSM-tables. After that, a second algorithm is presented showing how to calculate minimal state identification sets using the result of the first algorithm.

1. **Inputs.** Observable, minimal FSM $M = (Q, \underline{q}, \Sigma_I, \Sigma_O, h)$ and the OFSM-tables calculated for M as described in Section 3.6[3].

[3]Note that these are the OFSM-tables calculated for M, which is already minimal, and *not* the OFSM-tables constructed during the minimisation process resulting in M.

We can assume that the states in Q are ordered (e.g. by numbering them from zero to $(|Q|-1)$), so that a total reflexive, anti-symmetric, and transitive ordering relation $q \leq q'$ is defined for all pairs of states.

2. **Outputs.** Characterisation set W

3. Initialise $W := \varnothing$.

4. For every $q \in Q$, proceed as follows.

   (a) For every $q' \in Q$ with $q' > q$, set $(q_0, q'_0) := (q, q')$ and calculate a distinguishing sequence as follows.

      i. If (q_0, q'_0) are already distinguished by W, continue with 4a.
      ii. Set $k := 1$.
      iii. Find $\ell \geq 1$ such that (q_0, q'_0) are equivalent in OFSM-Table-$(\ell-1)$, but reside in different equivalence classes in OFSM-Table-ℓ.[4]
      iv. If $\ell - k > 0$, proceed to Step 4(a)viii.
      v. If $\ell - k = 0$, set x_k to any element of Σ_I, such that OFSM-Table-0 has an I/O-column $x_k/y, y \in \Sigma_O$, such that q_{k-1} has entry (-1) in this column[5] and q'_{k-1} has an entry ≥ 0 in this column[6] or vice versa.
      vi. Set $W := W \cup \{x_1.x_2.x_3 \ldots x_k\}$.
      vii. Continue with Step 4a.
      viii. Set x_k to any element of Σ_I, such that there exists a $y \in \Sigma_O$, such that (q_{k-1}, q'_{k-1}) are mapped under x_k/y to states belonging to different state classes in OFSM-Table-$(\ell - k)$. Let (q_k, q'_k) be representatives of these different classes.
      ix. Set $k := k + 1$.
      x. Continue with Step 4(a)iv.

5. Remove all input traces from W that are prefixes of other input traces also contained in W.

[4]Recall that in OFSM-Table-0, all states reside in the same class 0. Moreover, M is assumed to be minimal; as a consequence, all states are distinguished in the last FSM table. Therefore this $\ell \geq 1$ always exists.

[5]This means that q_{k-1} does not have any emanting transition labelled by x_k/y.

[6]This means that q'_{k-1} has an outgoing transition labelled by x_k/y.

6. Minimise W by determining a smallest subset $W_0 \subseteq W$ that still distinguishes all states of W. This step is necessary, since the algorithm specified in the loop above does *not* always produce a minimal characterisation set. Set $W := W_0$.

7. Return W.

Observe that a naive minimisation of the initial solution for W has complexity $O(2^{|W|})$. It should further be noted, that even the minimisation of W in Step 6 does not guarantee that the resulting characterisation set will be as small as possible.

- In Step 4(a)i distinguishing traces for (q_0, q_0') are not constructed, if this pair of states is already distinguished by the current state of W. In this step one might skip the construction of an input trace that might distinguish even more states than the input traces already contained in W.

- In Steps 4(a)v and 4(a)viii an input x_k is identified which is not uniquely determined. Other suitable values may lead to other input traces which again might distinguish more additional states than the ones resulting from the choice of x_k that has been made.

The identification of a minimal characterisation set requires the identification of *all* distinguishing traces of minimal length. This is a problem of exponential complexity, involving not only the size of the state space, but also the size of the input and output alphabets. Therefore we are satisfied with the algorithm above that only tries to get "close" to the optimal solution.

3.7.3 Algorithm 2. Finding Minimal State Identification Sets

With a characterisation set at hand, the construction of minimal state identification sets is performed as follows.

1. **Inputs.** Observable, minimal FSM $M = (Q, \underline{q}, \Sigma_I, \Sigma_O, h)$ and a characterisation set W calculated by Algorithm 1 above.

2. **Outputs.** Collection $\underline{W} = \{W_q \mid q \in Q\}$ of state identification sets.

3. $\underline{W} := \varnothing$.

4. Create matrix Matrix $N = (z_{i,j}), i, j \in \{0, \ldots, |Q| - 1\}$ with $z_{i,j} \subseteq W$, such that for all $i \neq j$, *every* input sequence contained in $z_{i,j}$ distinguishes q_i from q_j.

5. For each $i \in \{0, \ldots, |Q| - 1\}$,

   (a) Set $S := \{z_{i,j} \mid j \in \{0, \ldots, |Q| - 1\} \wedge j \neq i\}$.

   (b) Set W_{q_i} := solution of the minimal hitting set problem *with minimal cardinality* for the set system (W, S), constructed according to the algorithm in Appendix A.[7]

   (c) Set $\underline{W} := \underline{W} \cup \{W_{q_i}\}$.

6. Return $\underline{W}$.

Example 3. Consider the nondeterministic, observable, minimal FSM shown in Fig 3.8. This FSM has the following OFSM-tables.

OFSM-Table-0

[q]	q	0/0	0/1	1/0	1/1
0	0	1	1	5	0
0	1	2	2	1	-1
0	2	-1	0	-1	2
0	3	4	4	-1	3
0	4	3	2	5	4
0	5	3	3	1	-1

OFSM-Table-1

[q]	q	0/0	0/1	1/0	1/1
0	0	1	1	5	0
1	1	2	2	1	-1
2	2	-1	0	-1	2
3	3	4	4	-1	3
0	4	3	2	5	4
1	5	3	3	1	-1

[7] As explained in Appendix A, the minimal hitting set problem may have several solutions with *different* cardinality. The algorithm presented there calculates all solutions, so that one with with minimal cardinality can be identified.

OFSM-Table-2

[q]	q	0/0	0/1	1/0	1/1
0	0	1	1	5	0
1	1	2	2	1	-1
2	2	-1	0	-1	2
3	3	4	4	-1	3
4	4	3	2	5	4
5	5	3	3	1	-1

Applying the algorithm above, results in the characterisation and state identification sets

$$\begin{aligned} W &= \{0.0, 1\} \\ W_0 &= \{0.0, 1\} \\ W_1 &= \{0.0\} \\ W_2 &= \{0.0\} \\ W_3 &= \{0.0, 1\} \\ W_4 &= \{0.0\} \\ W_5 &= \{0.0, 1\} \end{aligned}$$

The following table shows the set of output traces resulting from applying the input sequences of W to each FSM state.

	Input Trace from W_i	
q	**0.0**	**1**
0	{0.0, 0.1, 1.0, 1.1}	{0, 1}
1	{0.1, 1.1}	{0}
2	{1.0, 1.1}	{1}
3	{0.0, 0.1, 1.0, 1.1}	{1}
4	{0.0, 0.1, 1.1}	{0, 1}
5	{0.0, 0.1, 1.0, 1.1}	{0}

It is easy to see from the table that FSM state q_i is distinguished from all other nodes by the input traces in W_i. □

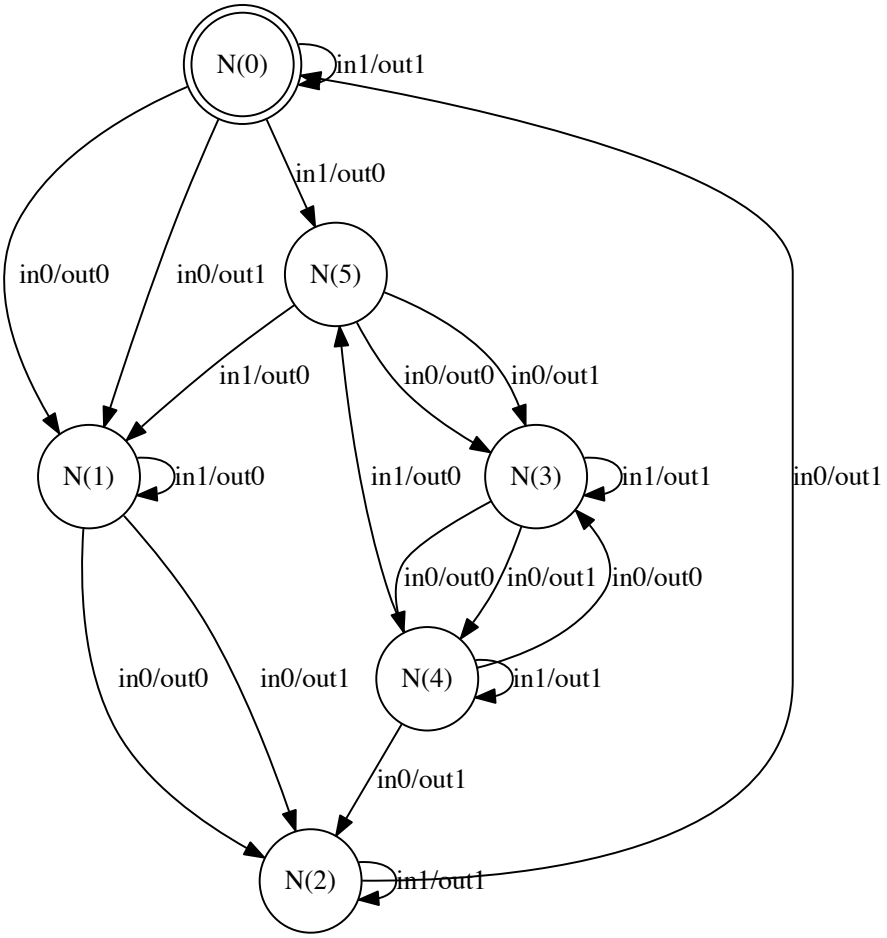


Figure 3.8: Nondeterministic, observable, minimal FSM used in Example 3.

3.8 Classification of FSM Fault Models

Fault models for FSMs from a given signature $\mathsf{Sig} = \mathrm{FSM}(\Sigma_\mathrm{I}, \Sigma_\mathrm{O})$ are denoted by $\mathcal{F} = (\mathsf{M}, \leq, \mathsf{Dom})$ with $\mathsf{M} \in \mathsf{Sig}$ and $\mathsf{Dom} \subseteq \mathsf{Sig}$, and in this lecture, we consider the conformance relations $\leq \in \{\sim, \preceq\}$ defined above. For

FSMs, the possible types of errors can be classified as follows [45].

1. Single output faults. An SUT behaving like an FSM $M' \in \mathsf{Dom}$ has a *single output fault*, if M' differs from the reference FSM M by one output in one transition. More precisely, fixing exactly one transition output, the resulting model M'' is I/O-equivalent to M.

2. Single transfer faults. An SUT behaving like an FSM $M' \in \mathsf{Dom}$ has a *single transfer fault*, if M' differs from the reference FSM M by the target state of a single transition. This means that by redirecting exactly one transition of M' to another target state, the resulting model M'' is I/O-equivalent to M.

Observe that a transfer fault will *not* invalidate the I/O-equivalence relationship, if the other target state of the transition under consideration is I/O-equivalent to the original state. To avoid this confusing distinction between "harmless" and "harmful" transfer faults, the fault domain and the reference model are often restricted to observable, minimal FSM: there any transfer fault will automatically violate I/O-equivalence.

3. Single extra transition faults. An SUT behaving like an FSM $M' \in \mathsf{Dom}$ has a *single extra transition fault*, if M' differs from the reference FSM M by a single additional transition. This means that by removing exactly one transition from M', the resulting model M'' is I/O-equivalent to M.

Note that adding an extra transition can also introduce a new state that has no corresponding state in M. If we consider single faults only, however, this new state can never be left, because this would require another extra transition.

4. Single missing transition faults. An SUT behaving like an FSM $M' \in \mathsf{Dom}$ has a *single missing transition fault*, if M' differs from the reference FSM M by a single missing transition. This means that by adding exactly one transition to M', the resulting model M'' is I/O-equivalent to M.

Exercise 3. Assuming that reference model M and SUT behaviour M' are both observable and minimal, identify all possible variants how an additional and a missing transition may look like. Distinguish between completely defined and incomplete FSMs, and between deterministic and nondeterministic

ones. Is there a variant that will *not* violate I/O-equivalence? Which of these variants will not only violate $\sim$, but also $\preceq$? $\square$

5. Multiple faults. An SUT behaving like an FSM $M' \in \mathbb{Dom}$ has *multiple faults*, if M' differs from the reference FSM M by multiple combinations of output, transfer, extra transition, and missing transition faults. By applying the fixing rules for fault types 1 – 4 multiple times, the resulting FSM M'' is again I/O-equivalent to M.

Fault injection functions In the testing theories to be presented in Chapter 4, it is useful to have a precise notion of the different faults defined above. Therefore we introduce *fault injection functions* describing how the faulty FSM differs from the reference FSM. Let $M = (Q, \underline{q}, \Sigma_I, \Sigma_O, h)$ be a reference model and $M' = (Q, \underline{q}, \Sigma_I, \Sigma_O, h')$ an FSM describing the true behaviour of the SUT.

An *output fault injection function* is defined by a function $\phi : h \to \Sigma_O$, such that

$$\forall (q, x, y, q') \in h : (q, x, \phi(q, x, y, q'), q') \notin h \vee \phi(q, x, y, q') = y$$

Function ϕ injects *multiple* output faults, if and only if $\{(q, x, y, q') \in h \mid \phi(q, x, y, q') \neq y\}$ has more than one element. FSM M' deviates from M by output faults, if an output fault injection function ϕ can be found such that

$$h' = \{(q, x, \phi(q, x, y, q'), q') \mid (q, x, y, q') \in h\}$$

A *transfer fault injection function* is defined by a function $\psi : h \to Q$, such that

$$\forall (q, x, y, q') \in h : (q, x, y, \psi(q, x, y, q')) \notin h \vee \phi(q, x, y, q') = q'$$

Function ψ injects *multiple* transfer faults, if and only if $\{(q, x, y, q') \in h \mid \psi(q, x, y, q') \neq q'\}$ has more than one element. FSM M' deviates from M by transfer faults, if an transfer fault injection function ψ can be found such that

$$h' = \{(q, x, y, \psi(q, x, y, q')) \mid (q, x, y, q') \in h\}$$

An *extra transition fault injection function* is defined by a function $\gamma : Q \to \mathbb{P}(\Sigma_I \times \Sigma_O \times Q)$, such that

$$\forall (q, x, y, q') \in h, (a, b, w) \in \gamma(q) : (q, a, b, w) \notin h \vee (a, b, w) = (x, y, q')$$

Function γ injects *multiple* extra transition faults, if and only if $\{(q, a, b, w) \mid q \in Q \wedge (a, b, w) \in \gamma(q)\} \setminus h$ has more than one element. FSM M' deviates from M by extra transition faults, if an extra transition fault injection function γ can be found such that

$$h' = h \cup \bigcup_{q \in Q} \{(q, x, y, q') \mid (x, y, q') \in \gamma(q)\}$$

A *missing transition fault injection function* is defined by a function $\mu : Q \to \mathbb{P}(\Sigma_I \times \Sigma_O \times Q)$, such that

$$\forall q \in Q, (a, b, w) \in \mu(q) : (q, a, b, w) \in h$$

Function μ injects *multiple* missing transition faults, if and only if $\{q \in Q \mid \mu(q) \neq \varnothing\}$ has more than one element. FSM M' deviates from M by missing transition faults, if a missing transition fault injection function μ can be found such that

$$h' = h \setminus \Big(\bigcup_{q \in Q} \{(q, x, y, q') \mid (x, y, q') \in \mu(q)\} \Big)$$

Chapter 4

Testing Theories for FSM

In this chapter, FSM test cases and a variety of complete testing theories for FSMs is presented; each theory is specialised for a certain class of fault models.

4.1 FSM Test Cases

For any model-based testing formalism, test cases can be introduced from two perspectives.

- The abstract perspective sees test cases as sets of observation traces (e.g. sequences of inputs and outputs) that are "forbidden" (so-called *fail traces*) and sets of observation traces the SUT needs to be able to perform (*pass traces*).
- The concrete perspective sees test cases as (executable) models with dynamic behaviour, running concurrently with the SUT, stimulating its input interfaces and checking its outputs.

Abstract FSM Test Cases

Definition 4.1 *Let* $\mathrm{FSM}(\Sigma_I, \Sigma_O)$ *be the set of FSMs over I/O-alphabet* (Σ_I, Σ_O)*. An (abstract)* FSM test case *is a tuple* $U = (U_{pass}, U_{fail})$ *with* $U_{pass}, U_{fail} \subseteq (\Sigma_I \times \Sigma_O)^*$*. The elements of* U_{pass} *are called* pass traces*, the elements of* U_{fail} *fail traces. by* $TC(\Sigma_I, \Sigma_O)$ *we denote the set of all test cases over I/O-alphabet* (Σ_I, Σ_O)*.* □

Following the general test case definition given in Chapter 2, we introduce two notions of FSMs passing or failing a test case. The first relation will be used for testing whether the SUT is I/O-equivalent to the reference model.

Definition 4.2 *Given* $M \in \mathrm{FSM}(\Sigma_I, \Sigma_O)$*, we say that* M passes test case $U = (U_{\mathrm{pass}}, U_{\mathrm{fail}})$ with respect to I/O-equivalence*, if and only if all pass traces of* U *and none of the fail traces are contained in the language of* M*. To this end, the notation*

$$M \underline{\mathrm{pass}}_{\sim} U \equiv \left(U_{\mathrm{pass}} \subseteq L(M) \wedge U_{\mathrm{fail}} \cap L(M) = \varnothing\right)$$

is used. □

The second relation will be used for testing for reduction.

Definition 4.3 *Given* $M \in \mathrm{FSM}(\Sigma_I, \Sigma_O)$*, we say that* M passes test case $U = (U_{\mathrm{pass}}, U_{\mathrm{fail}})$ with respect to reduction*, if and only if none of the fail traces are contained in the language of* M*. To this end, the notation*

$$M \underline{\mathrm{pass}}_{\preceq} U \equiv \left(U_{\mathrm{fail}} \cap L(M) = \varnothing\right)$$

is used. □

It is obvious that $M \underline{\mathrm{pass}}_{\sim} U$ implies $M \underline{\mathrm{pass}}_{\preceq} U$.

The following theorem justifies the definitions above. It states the well-known relationship between FSM test cases, I/O-equivalence, and reduction. Note that similar theorems also hold for other formalisms; in [25], for example, it is shown that certain types of test cases can characterise refinement relations in process algebras.

Theorem 4.1 (Abstract FSM tests characterise $\sim$ and $\preceq$)
Let $M_1, M_2 \in \mathrm{FSM}(\Sigma_I, \Sigma_O)$*. Then*

1. $M_2 \preceq M_1$ *if and only if*

$$\forall U \in TC(\Sigma_I, \Sigma_O) : M_1 \underline{\mathrm{pass}}_{\preceq} U \Rightarrow M_2 \underline{\mathrm{pass}}_{\preceq} U$$

2. $M_2 \sim M_1$ *if and only if*

$$\forall U \in TC(\Sigma_I, \Sigma_O) : M_1 \underline{\mathrm{pass}}_{\sim} U \Rightarrow M_2 \underline{\mathrm{pass}}_{\sim} U$$

Proof. 1. Suppose that $M_2 \preceq M_1$. By definition, this is equivalent to $L(M_2) \subseteq L(M_1)$. Not suppose that $M_1 \underline{\text{pass}}_{\preceq} U = (U_{\text{pass}}, U_{\text{fail}})$. By definition, this means that $U_{\text{fail}} \cap L(M_1) = \varnothing$. Since $L(M_2) \subseteq L(M_1)$, this implies $U_{\text{fail}} \cap L(M_2) = \varnothing$, and this is equivalent to $M_2 \underline{\text{pass}}_{\preceq} U$.

Conversely, suppose that $M_1 \underline{\text{pass}}_{\preceq} U \Rightarrow M_2 \underline{\text{pass}}_{\preceq} U$ holds for all tests in $\text{TC}(\Sigma_I, \Sigma_O)$. Now let $\iota = \overline{x}/\overline{y} \in (\Sigma_I \times \Sigma_O)^*$ be an arbitrary I/O-trace which is not contained in the language of M_1. Define test case $U = (U_{\text{pass}} = \varnothing, U_{\text{fail}} = \{\iota\})$. Then $M_1 \underline{\text{pass}}_{\preceq} U$, because $\iota \notin L(M_1)$. By assumption, M_2 passes this test as well. We conclude that ι is not in the language of M_2 either. This implies $L(M_2) \subseteq L(M_1)$, which means that $M_2 \preceq M_1$ holds. This concludes the proof of Statement 1.

2. Suppose that $M_2 \sim M_1$. By definition, this is equivalent to $L(M_2) = L(M_1)$. Not suppose that $M_1 \underline{\text{pass}}_{\sim} U = (U_{\text{pass}}, U_{\text{fail}})$. By definition, this means that $U_{\text{pass}} \subseteq L(M_1) \wedge U_{\text{fail}} \cap L(M_1) = \varnothing$. Since $L(M_2) = L(M_1)$, this implies $U_{\text{pass}} \subseteq L(M_2) \wedge U_{\text{fail}} \cap L(M_2) = \varnothing$, and this is equivalent to $M_2 \underline{\text{pass}}_{\sim} U$.

Conversely, suppose that $M_1 \underline{\text{pass}}_{\sim} U \Rightarrow M_2 \underline{\text{pass}}_{\sim} U$ holds for all tests in $\text{TC}(\Sigma_I, \Sigma_O)$. Now let $\iota = \overline{x}/\overline{y} \in (\Sigma_I \times \Sigma_O)^*$ be an arbitrary I/O-trace which is not contained in the language of M_1. Define test case $U = (U_{\text{pass}} = \varnothing, U_{\text{fail}} = \{\iota\})$. Then $M_1 \underline{\text{pass}}_{\preceq} U$, because $\iota \notin L(M_1)$. By assumption, M_2 passes this test as well. We conclude that ι is not in the language of M_2 either. This implies $L(M_2) \subseteq L(M_1)$. Now suppose that $\iota \in L(M_1)$. Then $M_1 \underline{\text{pass}}_{\sim} U'$ with $U' = (U'_{\text{pass}} = \{\iota\}, U'_{\text{fail}} = \varnothing)$. By our assumption, M_2 passes this test case as well, and this implies $\iota \in L(M_2)$. As a consequence, $L(M_1) \subseteq L(M_2)$, so we have shown that $L(M_2) = L(M_1)$. This concludes the proof of Statement 2. □

Concrete FSM Test Cases

Our concrete FSM test case definition follows closely the exposition given in [55]. FSM test cases are observable, acyclic, terminating, single-input, and output-complete NFSM $U = (Q_u, \underline{u}, \Sigma_I, \Sigma_O, h_u)$ with an additional deadlock state *fail* $\in Q_u$. Acyclic means that an execution never visits the same state twice. Terminating means that each execution is finite, because finally a deadlock state will be reached. Termination is a direct consequence of acyclicity: since an FSM has only finitely many states, and since a test case cannot visit a state more than once, it has to terminate before some state

would have to be visited again. Single-input denotes FSMs where in each state $u \in Q_u$ that does not deadlock exactly one input (denoted by $x(u)$) is defined. Output-complete means that from each non-deadlock state every output may occur. Single-input implies that U is *not* completely specified, and output-complete therefore implies that U must be nondeterministic (if $|\Sigma_O| > 1$). Since U is also observable, the occurrence of a specific output in a given state determines the post-state to be reached by U in a unique way. As a consequence, if $\overline{x}/\overline{y} \in L(U)$, the set $q\text{-}\underline{\text{after}}\text{-}\overline{x}/\overline{y}$ always contains a single state.

U operates in parallel with some FSM M representing the SUT, and parallel composition is the FSM intersection $M \cap U$ defined in Section 3.2. $M \cap U$ executes only those I/O-traces $\overline{x}/\overline{y}$ that can be executed by both U and M, since $L(M \cap U) = L(M) \cap L(U)$. A terminating execution of $M \cap U$ is called *test execution*.

Let (q, u) be a deadlock state of $M \cap U$ that forces the test execution to terminate. If M is completely specified, then the reason for $M \cap U$ to deadlock cannot be that the next input accepted by U is not accepted by M. Since U is output complete, the deadlock cannot have been caused because U does not accept the output produced by M. These considerations lead to the following lemma.

Lemma 4.1 *Let M be completely specified and U a test case over the same I/O-alphabet. Suppose that a test execution of $M \cap U$ terminates in state (q, u). Then at least one of the states q, u is a deadlock state of M and U, respectively.* □

The test execution *fails* if U terminates in *fail*, otherwise the test execution *passes*. For deterministic SUT M, $M \cap U$ can only perform a single test execution. If, however, M is nondeterministic, $M \cap U$ may perform more than one test execution, and some of these may fail, while others pass. Therefore we define

$$M \ \underline{\text{pass}} \ U \equiv \forall \overline{x}/\overline{y} \in L(M \cap U), u \in Q_u : u = \underline{u}\text{-}\underline{\text{after}}\text{-}\overline{x}/\overline{y} \Rightarrow u \neq \mathit{fail}$$

A test case U is *preset* if it engages in exactly one input sequence. This means that for all $\overline{x}/\overline{y} \in L(U)$, $\overline{x}$ is a prefix of a pre-defined maximal input sequence $\overline{x}_u$. If U is not preset, it is called *adaptive*. This means that the next input to be selected not only depends on the number of inputs performed so far, but on the current state of U.

When fixing an input alphabet Σ_I and an output alphabet Σ_O, the set of all FSM test cases operating on the same alphabets is denoted by $TC(\Sigma_I, \Sigma_O)$. Since FSM test cases are single-input, the set $DF(U)$ of non-blocking states induces a well-defined mapping $x : DF(U) \to \Sigma_I$ by setting $x(q) \in \Sigma_I$ to the uniquely defined input that can be processed in this state.

The following Theorem is the equivalent to Theorem 4.1, now formulated for concrete FSM test cases.

Theorem 4.2 (Concrete FSM tests characterise $\sim$ and $\preceq$)
Let $M_1, M_2 \in FSM(\Sigma_I, \Sigma_O)$. *Then*

1. $M_2 \preceq M_1$ *if and only if*

$$\forall U \in TC(\Sigma_I, \Sigma_O) : M_1 \underline{\text{pass}}\ U \Rightarrow M_2 \underline{\text{pass}}\ U$$

2. $M_2 \sim M_1$ *if and only if*

$$\forall U \in TC(\Sigma_I, \Sigma_O) : M_1 \underline{\text{pass}}\ U \Rightarrow M_2 \underline{\text{pass}}\ U$$

Proof. We prove Statement 1; then Statement 2 follows from the fact that $M_2 \sim M_1$ holds if and only if $M_2 \preceq M_1$ and $M_1 \preceq M_2$.

Suppose that $M_2 \preceq M_1$. Then, by definition of $\preceq$, $L(M_2) \subseteq L(M_1)$ follows. For any test case $U = (Q_u, \underline{u}, \Sigma_I, \Sigma_O, h)$ we have $L(M_2 \cap U) \subseteq L(M_2)$, so $L(M_2 \cap U) \subseteq L(M_1 \cap U)$ follows. Now suppose that M_1 $\underline{\text{pass}}$ U and suppose that $\overline{x}/\overline{y} \in L(M_2 \cap U)$ with $\overline{x} = x_0 \dots x_k$, $\overline{y} = y_0 \dots y_k$ and $k \geq 0$. Since $L(M_2 \cap U) \subseteq L(M_1 \cap U)$, $\overline{x}/\overline{y} \in L(M_1 \cap U)$ follows, and M_1 $\underline{\text{pass}}$ U implies that $\overline{x}/\overline{y}$ is a $\underline{\text{pass}}$-I/O-trace of $M_1 \cap U$.

Since U is observable, there exists a unique sequence of states $u_0 \dots u_{k+1}$ with $u_0 = \underline{u}$, such that $x(u_i) = x_i$ and $u_{i+1} = u_i\text{-}\underline{\text{after}}\text{-}x_i/y_i$ for $i = 0 \dots, k$. Then $u_i \neq \mathit{fail}$ for $i = 0, \dots, k+1$, because M_1 passes the test. Since $u_0 \dots u_{k+1}$ is unique, the test U, when running in parallel composition with M_2, runs through the same state sequence $u_0 \dots u_{k+1}$. Since none of the u_i is a *fail*-state, M_2 $\underline{\text{pass}}$ U follows.

Conversely, suppose that $M_2 \not\preceq M_1$. Then there exists $\overline{x}/\overline{y} \in L(M_2)$ with $\overline{x}/\overline{y} \notin L(M_1)$. Now construct a test $U \in TC(\Sigma_I, \Sigma_O)$ in such a way that it only produces input trace $\overline{x}$ or prefixes thereof, passes for all I/O-traces $\overline{x}'/\overline{y}' \in L(M_1)$ where $\overline{x}'$ is a prefix of (and including) $\overline{x}$, and fails for all I/O-traces $\overline{x}''/\overline{y}'' \notin L(M_1)$, where $\overline{x}''$ is a prefix of $\overline{x}$ and $\overline{y}''$ is a prefix of $\overline{y}$ of the same length as $\overline{x}''$. Then, M_1 $\underline{\text{pass}}$ U, but M_2 $\underline{\text{fail}}$ U. This completes the proof. $\square$

4.2 State Cover and Transition Cover

In most testing theories it is essential to cover every state or transition occurring in the reference model. This leads to the following definitions.

Definition 4.4 *Given an FSM* $M = (Q, \underline{q}, \Sigma_I, \Sigma_O, h)$*, a* state cover *of* M *is a set* $SCOV(M) \subseteq \Sigma_I^*$ *of input sequences, such that for all reachable states* $q \in Q$*, there exists an input sequence* $\underline{x} \in SCOV(M)$ *and an output sequence* $\underline{y} \in \Sigma_O^*$*, such that* $(\underline{q}, \underline{x}, \underline{y}, q) \in h$*.*

Observe that the empty sequence ε is contained in $SCOV(M)$, because it is needed to "reach" the initial state.

Definition 4.5 *Given a completely specified FSM* $M = (Q, \underline{q}, \Sigma_I, \Sigma_O, h)$*, a* transition cover *of* M *is the set* $TCOV(M) = SCOV(M).\Sigma_I$ *consisting of all sequences from* M*'s state cover; each sequence extended by every possible input.*

Note that for FSMs that are not completely specified, the transition cover only appends those inputs to a state cover input sequence $\underline{x}$ that are defined in the target states reached by $\underline{x}$.

4.3 The T-Method

The following testing theory has been introduced in [47]; it is called the *T-Method*, because each test suite is derived from the reference model by performing a *transition tour*, i.e. every transition of the reference model is performed at least once by the test suite.

Fault models In its most restricted version, the T-Method takes deterministic, completely defined FSMs as reference models; for given alphabets Σ_I, Σ_O we denote this set as $DFSM_C(\Sigma_I, \Sigma_O)$. The fault model $Dom(M) \subseteq DFSM_C(\Sigma_I, \Sigma_O)$ depends on the reference machine $M = (Q, \underline{q}, \Sigma_I, \Sigma_O, h) \in DFSM_C(\Sigma_I, \Sigma_O)$ and consists of all deterministic completely defined FSMs M' that deviate from M by output faults, specified by some output fault injection function $\phi : h \to \Sigma_O$. As conformance relation, I/O-equivalence $\sim$ is choosen.

Summarising, we consider the set F of all fault models $\mathcal{F}(M, \sim, \mathrm{Dom}(M))$ with

$$\begin{aligned} M &= (Q, \underline{q}, \Sigma_I, \Sigma_O, h) \in \mathrm{DFSM}_C(\Sigma_I, \Sigma_O) \\ \mathrm{Dom}(M) &= \{\ (Q, \underline{q}, \Sigma_I, \Sigma_O, h') \mid \exists \phi : h \to \Sigma_O : \\ &\qquad h' = \{(q, x, \phi(q, x, y, q'), q') \mid (q, x, y, q') \in h\}\ \} \end{aligned}$$

Abstract test cases The abstract test cases for applying the T-Method are constructed as follows: for $\overline{x} \in \Sigma_I^*$, define

$$U(\overline{x}) = (U_{\mathrm{pass}}, U_{\mathrm{fail}}) = (\{\overline{x}/\overline{y} \mid \overline{x}/\overline{y} \in L(M_1)\}, \varnothing)$$

$U(\overline{x})$ is the test case that can only be passed if the SUT produces the input/output trace $\overline{x}/\overline{y}$ which is in the language of the reference model.

Concrete test cases As concrete test cases we take terminating, acyclic, single-input DFSMs $U(\overline{x})$ with $\overline{x} = x_1 \dots x_k \in \Sigma_I^*$, such that M' $\underline{\mathrm{pass}}$ $U(\overline{x})$, if and only if the I/O-trace $\overline{x}/\overline{y}$ performed by $M' \cap U$ is an element of $L(M)$. More formally, $U(\overline{x})$ is defined by

$$\begin{aligned} U(\overline{x}) &= (Q_u, \underline{u}, \Sigma_I, \Sigma_O, h_u) \\ Q_u &= \{\underline{u}, u_1, \dots, u_{\#\overline{x}}, \mathit{fail}\} \\ h_u &= \{(\underline{u}, \varepsilon, \varepsilon, \underline{u})\} \cup \\ &\quad \{(\underline{u}, x_1, y, u_1) \mid x_1/y \in L(M)\} \cup \\ &\quad \{(\underline{u}, x_1, y, \mathit{fail}) \mid x_1/y \notin L(M)\} \cup \\ &\quad \bigcup_{i=1}^{\#\overline{x}-1} \{(u_i, x_{i+1}, y, u_{i+1}) \mid \exists y_1, \dots, y_i \in \Sigma_O : \\ &\qquad\qquad x_1 \dots x_i.x_{i+1}/y_1 \dots y_i.y \in L(M)\} \cup \\ &\quad \bigcup_{i=1}^{\#\overline{x}-1} \{(u_i, x_{i+1}, y, \mathit{fail}) \mid \forall y_1, \dots, y_i \in \Sigma_O : \\ &\qquad\qquad x_1 \dots x_i.x_{i+1}/y_1 \dots y_i.y \notin L(M)\} \end{aligned}$$

Completeness Theorem for the T-Method

Theorem 4.3 *For each of the fault models* $\mathcal{F}(M, \sim, \mathrm{Dom}(M))$ *introduced above, the T-Method test suite* $\mathbf{TS}(M) = \{U(\overline{x}) \mid \overline{x} \in \mathrm{TCOV}(M)\}$ *is complete.*

Proof. Let $U(\overline{x}) = (\{\overline{x}/\overline{y}\}, \varnothing) \in \mathbf{TS}(M)$. Then, by definition of $U(\overline{x})$, $\overline{x}/\overline{y} \in L(M)$. Therefore $M' \underset{\sim}{\text{pass}} U(\overline{x})$ holds for all DFSMs with $L(M') = L(M)$. As a consequence, the test suite $\mathbf{TS}(M)$ is sound.

Now let $M' = (Q', \underline{q}', \Sigma_I, \Sigma_O, h') \in \mathrm{Dom}(M)$, such that $L(M') \neq L(M)$. Then there exists an input trace $\overline{x}.x_1$ and associated output traces $\overline{y}, \overline{y}.y_1, \overline{y}.y_2$, such that $\overline{x}/\overline{y} \in L(M') \cap L(M)$, but $\overline{x}.x_1/\overline{y}.y_2 \in L(M')$ and $\overline{x}.x_1/\overline{y}.y_1 \in L(M)$ and $y_2 \neq y_1$.

Let $\overline{x}'$ be an element of the state cover of M, such that $\underline{q}\text{-}\underline{\text{after}}\text{-}\overline{x} = \underline{q}\text{-}\underline{\text{after}}\text{-}\overline{x}'$. Since M' differs from M only in certain outputs, but has the same number of states and the same transition topology, $\underline{q}'\text{-}\underline{\text{after}}\text{-}\overline{x} = \underline{q}'\text{-}\underline{\text{after}}\text{-}\overline{x}'$ follows. By construction, $\overline{x}'.x_1 \in \mathrm{TCOV}(M)$, and there exist output traces $\overline{y}'$ and $\overline{y}''$ such that

$$h(\underline{q}, \overline{x}'.x_1, \overline{y}'.y_1, \underline{q}\text{-}\underline{\text{after}}\text{-}\overline{x}.x_1) \quad \text{and} \quad h'(\underline{q}', \overline{x}'.x_1, \overline{y}''.y_2, \underline{q}'\text{-}\underline{\text{after}}\text{-}\overline{x}.x_1),$$

such that the output traces $\overline{y}'.y_1$ and $\overline{y}''.y_2$ differ at least in the last outputs y_1 and y_2. As a consequence, M' fails the test $U(\overline{x}'.x_1) \in \mathbf{TS}(M)$ which shows that $\mathbf{TS}(M)$ is exhaustive. □

Exercise 4. The objective of this exercise is to apply the T-Method in practise, using the FSM C++ library `fsmlib-cpp` which is available as open source on github – please download from

`origin`, `https://github.com/agbs-uni-bremen/fsmlib-cpp.git`

Build the library using Cmake; this will result in 4 libraries located in your build directory which have to be linked when building an executable using the fsmlib-cpp:

```
./fsm/libfsm-fsm.a
./interface/libfsm-interface.a
./sets/libfsm-sets.a
./trees/libfsm-trees.a
```

Use the test harness for the Ceiling Speed Monitor from Exercise 2 which is now modified and extended by a class TestCase as shown below.

1. Encode the DFSM you created in Exercise 2 in the *.fsm-Format. This format consists of lines structured as

   ```
   <source-state> <input-number> <output-number> <target-state>
   ```

   This FSM definition file is read by the Dfsm-Constructor used in the function `createTestSuite()` called from the main program (see below).

2. Fill in the input conversion table for your FSM in `TestCase.hpp`, as indicated by the example made available there. (The output conversion table is already defined.)

3. Run the test suite which uses test cases created according to the T-Method, using the method invocation `dfsm.tMethod()` in `createTestSuite()`.

4. Inject faults into your CSM implementation which is tested by this configuration and identify typical faults that are uncovered by the test suite and others that are not. Give explanations why the test suite based on the T-Method could not find these faults.

5. Inject faults into your CSM implementation that cannot be found by any DFSM testing method, because your input discretisation is not suitable for this fault.

□

The test harness `main.cpp` is written for this exercise as follows.

```
1  #include <string>
2  #include <vector>
3  #include <iostream>
4  #include <fstream>
5
6  #include "CSM.hpp"
7  #include "TestCase.hpp"
8  #include "fsm/Dfsm.h"
9  #include "fsm/InputTrace.h"
10 #include "fsm/IOTrace.h"
11 #include "interface/FsmPresentationLayer.h"
12 #include "trees/IOListContainer.h"
13
14 std::vector<TestCase*> testCases;
15
```

```
16 void assertAndLog(bool cnd, std::string testCase, float v, float vMax, bool c,
17                   int svcBrake, int emerBrake, CSM::display_t d) {
18
19     std::string verdict;
20
21     verdict = (cnd) ? "PASS" : "FAIL";
22
23     std::cout << testCase << " : " << verdict
24     << " v = " << v << " vMax = " << vMax << " c = " << c
25     << " svcBrake = " << svcBrake << " emerBrake = " << emerBrake
26     << " d = " << d << std::endl;
27 }
28
29
30 int main(int argc, const char * argv[]) {
31
32     float v;
33     float vMax;
34     bool c;
35     int svcBrake;
36     int svcBrakeEpd;
37     int emerBrake;
38     int emerBrakeEpd;
39     CSM::display_t d;
40     CSM::display_t dEpd;
41
42
43     createTestSuite();
44
45     CSM csm;
46
47     // Run test cases in the TEST HARENESS
48     for ( size_t tc = 0; tc < testCases.size(); tc++ ) {
49
50
51         std::cout << std::endl << "========== Test Case " << tc
52         << " ==========" << std::endl;
53         while ( testCases[tc]->getNext(v,vMax,c,
54                                        svcBrakeEpd,emerBrakeEpd,dEpd) ) {
55             csm.doCSM(v, vMax, c, svcBrake, emerBrake, d);
56             assertAndLog(svcBrake == svcBrakeEpd and
57                                  emerBrake == emerBrakeEpd and d == dEpd,
58                          testCases[tc]->toString(),
59                          v,vMax,c,svcBrake,emerBrake,d);
60         }
```

```
61
62              csm.reset();
63          }
64
65          return 0;
66
67      }
```

The `createTestSuite()` function is specified by

```
1  void createTestSuite() {
2
3      std::shared_ptr<FsmPresentationLayer> pl =
4              std::make_shared<FsmPresentationLayer>();
5
6
7      Dfsm dfsm("csm.fsm",pl,"CSM");
8
9      dfsm.toDot("csm");
10
11     IOListContainer c = dfsm.tMethod();
12
13     for ( std::vector<int> v : *c.getIOLists() ) {
14         InputTrace i(v,pl);
15         IOTrace io = dfsm.applyDet(i);
16         TestCase* t = new TestCase(io);
17         testCases.push_back(t);
18     }
19
20 }
```

The TestCase class is specified as follows.

```
1  //
2  //  TestCase.hpp
3  //  CSM
4  //
5  //  Created by Jan Peleska on 2016-10-25.
6  //  Copyright (c) 2016 Jan Peleska. All rights reserved.
7  //
8  #ifndef TestCase_hpp
9  #define TestCase_hpp
10
11 #include <stdio.h>
12 #include <string>
13 #include <vector>
14
15 #include "CSM.hpp"
16 #include "fsm/IOTrace.h"
17 #include "fsm/InputTrace.h"
18 #include "fsm/OutputTrace.h"
19
20 typedef struct inTable {
21     float v;
22     float vMax;
23     bool c;
24 } inTable_t;
25
26 typedef struct outTable {
27     int svcBrake;
28     int emerBrake;
29     CSM::display_t d;
30 } outTable_t;
31
32 class TestCase {
33
34 private:
35
36     std::vector<int> inputs;
37     std::vector<int> outputs;
38     unsigned int step;
39
40 public:
41
42     // @todo Here is an example for an FSM with input
43     // alphabet 0..6. Exchanges this by your complete
44     // input conversion table.
```

```
45      constexpr static const inTable_t inConv[]  = {
46          { 30.0f, 50.0f, true },  // 0
47          { 52.0f, 50.0f, true },  // 1
48          { 55.0f, 50.0f, true },  // 2
49          { 56.0f, 50.0f, true },  // 3
50          { 60.0f, 50.0f, true },  // 4
51          { 0.0f,  50.0f, false },   // 5
52          { 30.0f, 50.0f, false },   // 6
53
54      };
55
56      constexpr static const outTable_t outConv[] = {
57          { 0, 0, CSM::OK },    // 0
58          { 0, 0, CSM::OVR },   // 1
59          { 0, 0, CSM::WRN },   // 2
60          { 1, 0, CSM::IV },    // 3
61          { 1, 1, CSM::IV },    // 4
62      };
63
64
65      TestCase(std::string tc);
66
67      TestCase(IOTrace io);
68
69      bool getNext(float& v, float& vMax, bool& c,
70                   int &svcBrake, int &emerBrake, CSM::display_t &d);
71
72      std::string toString();
73  };
74  #endif /* TestCase_hpp */
```

```
1  //
2  //  TestCase.cpp
3  //  CSM
4  //
5  //  Created by Jan Peleska on 2016-10-25.
6  //  Copyright (c) 2016 Jan Peleska. All rights reserved.
7  //
8  #include <iostream>
9  #include <sstream>
10 #include "TestCase.hpp"
11
12
13 const inTable_t TestCase::inConv[];
14 const outTable_t TestCase::outConv[];
15
16
17 TestCase::TestCase(IOTrace io) {
18
19     step = 0;
20
21     InputTrace i = io.getInputTrace();
22     OutputTrace o = io.getOutputTrace();
23
24     for ( size_t n = 0; n < i.get().size(); n++ ) {
25         inputs.push_back(i.get()[n]);
26         outputs.push_back(o.get()[n]);
27     }
28
29
30 }
31
32 TestCase::TestCase(std::string tc) {
33
34     step = 0;
35
36     size_t n = tc.find("(");
37
38     while ( n != std::string::npos ) {
39
40         size_t m = tc.find(",",n);
41
42         if ( m == std::string::npos ) {
43             std::cerr << "Cannot parse tc string " << tc << std::endl;
44             return;
45         }
```

```
46
47          int inp = atoi(tc.substr(n+1,m-n-1).c_str());
48
49          inputs.push_back(inp);
50
51          size_t p = tc.find(")",m);
52
53          if ( p == std::string::npos ) {
54              std::cerr << "Cannot parse tc string " << tc << std::endl;
55              return;
56          }
57
58          int outp = atoi(tc.substr(m+1,p-m-1).c_str());
59
60          outputs.push_back(outp);
61
62          n = tc.find("(",p);
63
64      }
65
66  }
67
68  bool TestCase::getNext(float &v, float &vMax, bool& c,
69                         int &svcBrake, int &emerBrake, CSM::display_t &d) {
70
71      bool ret = true;
72
73      if ( step < inputs.size() ) {
74          int n = inputs[step];
75          v = inConv[n].v;
76          vMax = inConv[n].vMax;
77          c = inConv[n].c;
78      }
79      else {
80          ret = false;
81      }
82
83      if ( ret and step < outputs.size() ) {
84          int n = outputs[step];
85          svcBrake = outConv[n].svcBrake;
86          emerBrake = outConv[n].emerBrake;
87          d = outConv[n].d;
88          step++;
89      }
90      else {
```

```
91          ret = false;
92      }
93
94      return ret;
95  }
96
97  std::string TestCase::toString() {
98      std::ostringstream s;
99      bool first = true;
100     for ( size_t n = 0; n < inputs.size(); n++ ) {
101         if ( not first ) {
102             s << ".";
103         }
104         s << "(" << inputs[n] << "," << outputs[n] << ")";
105         first = false;
106     }
107     return s.str();
108 }
```

4.4 Test Oracles for Checking I/O-Equivalence and Reduction

Given a nondeterministic FSM reference model M_1, let FSM M_2 describe the true behaviour of a given SUT. Consider any test case $\overline{x}$ given by an input sequence $\overline{x} = x_1 \dots x_k \in \Sigma_I^*$. When checking whether M_2 is a reduction of M_1, it has to be shown that any I/O-sequence $\overline{x}/\overline{y}$ produced by M_2 when reacting to test case $\overline{x}$ is also an element of M_1's language, that is, whether $\overline{x}/\overline{y} \in L(M_1)$. Since M_2 may be nondeterministic, it is not guaranteed that erroneous behaviour will be uncovered in a single execution of the test case against the SUT: it may be produce an erroneous I/O-trace $\overline{x}/\overline{y}' \notin L(M_1)$ only after a number of executions of $\overline{x}$ or not at all, since nondeterminism does not guarantee that every behaviour of the system will be revealed within a finite number of executions.

For complete test suites checking nondeterministic systems, it is therefore necessary to state the *complete testing assumption* [26]. This hypothesis assumes the existence of a constant $c \geq 1$, such that executing a test case c times against the SUT will reveal *every* possible behaviour of the SUT. Under this assumption, it suffices to execute every test case c times in order to ensure that every possible reaction of the SUT to the given test case,

including the erroneous ones, will come up at least once.

When checking for reduction, each of the c executions of $\overline{x}$ resulting in I/O-traces $\overline{x}/\overline{y}_1, \ldots, \overline{x}/\overline{y}_c$ just need to be checked with respect to containment in $L(M_1)$. When checking for I/O-equivalence, however, it has to be checked that *every* reaction to $\overline{x}$ contained in $L(M_1)$ has been observed. This means that it has to be checked whether

$$\{\overline{x}/\overline{y}_1, \ldots, \overline{x}/\overline{y}_c\} = \left\{\overline{x}/\overline{y} \in \{\overline{x}\} \times \Sigma_O^* \mid \overline{x}/\overline{y} \in L(M_1)\right\}$$

holds.

In practice, such a constant c is determined by some form of grey-box testing, where, for example, the code and the states covered during the tests are monitored, so that it can be determined whether other behaviours of the SUT that have not been observed yet, might still come up.

4.5 A Complete Testing Theory Derived From Product Automata

In this section we prove a theorem which is quite well known in both the fields of testing and model checking of finite state systems: if two completely specified, possibly nondeterministic, FSMs M_1, M_2 have n and m states, respectively, it suffices to test all input traces in Σ_I^* up to a length of mn, in order to check whether M_2 and M_1 are I/O-equivalent, or whether M_2 is a reduction of M_1. This is the first and most fundamental theorem showing that non-terminating systems can be checked for I/O-equivalence or reduction with only finitely many (though really *very* many!) tests.

For an FSM $M_1 = (Q, \underline{q}, \Sigma_I, \Sigma_O, h)$ and $q_1 \in Q$, recall from Section 3.2 that

$$q_1\text{-}\underline{\text{after}}\text{-}\overline{x} = \{q' \mid \exists \overline{y} \in \Sigma_O^* : h(q_1, \overline{x}, \overline{y}, q')\}.$$

We extend this definition to sets $V \subseteq \Sigma_I^*$ of input traces by setting

$$q_1\text{-}\underline{\text{after}}\text{-}V = \{q' \mid \exists \overline{x} \in V : q' \in q_1\text{-}\underline{\text{after}}\text{-}\overline{x}\}.$$

Using this notation, the following lemma describes a simple fact about sets V of input traces and the associated sets of states that are reachable under these traces: if the input traces in V do *not* reach all reachable states, then the extended traces set $V \cup V.\Sigma_I$ reaches at least one additional state of the FSM.

Lemma 4.2 *Let* $M = (Q, \underline{q}, \Sigma_I, \Sigma_O, h)$ *be a (possibly nondeterministic) FSM over input alphabet* Σ_I *and output alphabet* Σ_O*. Let* $V \subseteq \Sigma_I^*$ *be a finite set of input traces containing the empty trace* ε*. Then either*

1. $\underline{q}$-after-V *contains all reachable states, i.e.,* $\underline{q}$-after-$V = \underline{q}$-after-Σ_I^**, or*
2. $\underline{q}$-after-$(V \cup V.\Sigma_I)$ *contains at least one additional state which is not contained in* $\underline{q}$-after-V*, that is,* $\underline{q}$-after-$V \subsetneq \underline{q}$-after-$(V \cup V.\Sigma_I)$*.*

Proof. Suppose that $q \in Q - (\underline{q}\text{-after-}V)$ is a reachable state of M_1. Then q may be reached via some input trace $\bar{x} \in \Sigma_I^*$, that is, $q \in \underline{q}$-after-$\bar{x}$. By assumption, $\bar{x} \notin V$. Then it is possible to decompose $\bar{x}$ into $\bar{x} = \bar{x}_1.x.\bar{x}_2$ with $x \in \Sigma_I$ and $\bar{x}_1, \bar{x}_2 \in \Sigma_I^*$, such that

- $\underline{q}$-after-$\bar{z} \subseteq \underline{q}$-after-$V$ holds for all prefixes $\bar{z}$ of $\bar{x}_1$ including $\bar{x}_1$, and
- there exists a state q' such that $q' \in (\underline{q}\text{-after-}\bar{x}_1.x) \setminus (\underline{q}\text{-after-}V)$.

This means that $\bar{x}_1$ is the longest prefix of $\bar{x}$ such that all states reachable by $\bar{x}_1$ and its prefixes can also be reached by some input traces from V. This $\bar{x}_1$ always exists, because it may also be the empty trace which is contained in V. Observe that $\bar{x}_1$ is not necessarily an element of V.

Now let $q'' \in \underline{q}$-after-$\bar{x}_1$, such that $h(q'', x, y, q')$ for some $y \in \Sigma_O$. Then q'' can be reached by an input trace $\bar{v} \in V$ according to our assumption about $\bar{x}_1$. As a consequence, $q' \in \underline{q}$-after-$\bar{v}.x$, and $\bar{v}.x \in V.\Sigma_I$. Now we have identified a state q' which is not contained in $\underline{q}$-after-V, but in $\underline{q}$-after-$(V \cup V.\Sigma_I)$. This completes the proof. □

The lemma above has an important corollary.

Corollary 4.1 *Let* $M_1 = (Q_1, \underline{q}_1, \Sigma_I, \Sigma_O, h_1)$ *and* $M_2 = (Q_2, \underline{q}_2, \Sigma_I, \Sigma_O, h_2)$ *be FSMs over the same alphabets with* n *and* m *states, respectively. Then every reachable state of the product FSM* $M_1 \cap M_2$ *can be reached by an input trace from* $\bigcup_{i=0}^{nm-1} \Sigma_I^i$*. Therefore,* $\bigcup_{i=0}^{nm-1} \Sigma_I^i$ *is a state cover of* $M_1 \cap M_2$*.*

Proof. By the definition of the FSM product, $M_1 \cap M_2$ has at most nm states. Define $V = \{\varepsilon\}$. Now apply Lemma 4.2 $nm - 1$ times to conclude that $V \cup V.\Sigma_I \cup \cdots \cup V.\Sigma_I^{nm-1}$ reaches all states (in the worst case, all nm states) of $M_1 \cap M_2$. Observing that $\Sigma_I^0 = \{\varepsilon\}$ by definition, we conclude that

$$V \cup V.\Sigma_I \cup \cdots \cup V.\Sigma_I^{nm-1} = \bigcup_{i=0}^{nm-1} \Sigma_I^i,$$

and this completes the proof. □

Theorem 4.4 *Let* $M_1 = (Q_1, \underline{q}_1, \Sigma_I, \Sigma_O, h_1)$, $M_2 = (Q_2, \underline{q}_2, \Sigma_I, \Sigma_O, h_2)$ *be two observable FSMs with* n *and* m *states, respectively. Then*

$$L(M_2) \subseteq L(M_1) \quad \textit{if and only if} \quad L(M_2) \cap \bigcup_{i=0}^{mn} (\Sigma_I \times \Sigma_O)^i \subseteq L(M_1) \cap \bigcup_{i=0}^{mn} (\Sigma_I \times \Sigma_O)^i.$$

Proof. We prove by contradiction that the assumption $L(M_2) \cap \bigcup_{i=0}^{mn} (\Sigma_I \times \Sigma_O)^i \subseteq L(M_1) \cap \bigcup_{i=0}^{mn} (\Sigma_I \times \Sigma_O)^i$ implies $L(M_2) \subseteq L(M_1)$. Suppose therefore that the assumption holds, but $L(M_2) \not\subseteq L(M_1)$.

Then there exists a shortest $\pi \in (\Sigma_I \times \Sigma_O)^*$ such that $\pi \in L(M_2) \setminus L(M_1)$. Since $L(M_2) \cap \bigcup_{i=0}^{mn} (\Sigma_I \times \Sigma_O)^i \subseteq L(M_1) \cap \bigcup_{i=0}^{mn} (\Sigma_I \times \Sigma_O)^i$, the I/O-trace π satisfies $|\pi| \geq mn + 1$. Let $\pi = \pi'.x/y$ for some $\pi' \in (\Sigma_I \times \Sigma_O)^{|\pi|-1}$, $x/y \in \Sigma_I \times \Sigma_O$. Since $\pi \in L(M_2)$, we have $\pi' \in \text{pref}(\pi) \subseteq L(M_2)$. Since π is the shortest trace satisfying $\pi \in L(M_2) \setminus L(M_1)$, we have $\pi' \in L(M_2) \cap L(M_1)$.

Since M_1 and M_2 are observable, there is a unique pair of states (q_1, q_2) satisfying $(q_1, q_2) = (\underline{q}_1\text{-}\underline{\text{after}}\text{-}\pi', \underline{q}_2\text{-}\underline{\text{after}}\text{-}\pi')$.

Then $x/y \in L_{M_2}(q_2) \setminus L_{M_1}(q_1)$. Since $\bigcup_{i=0}^{mn-1} \Sigma_I^i$ is a state cover of $M_1 \cap M_2$ by Corollary 4.1, there exists a trace $\tau \in (\Sigma_I \times \Sigma_O)^*$, $|\tau| \leq mn - 1$ such that $(\underline{q}_1\text{-}\underline{\text{after}}\text{-}\tau, \underline{q}_2\text{-}\underline{\text{after}}\text{-}\tau) = (\underline{q}_1\text{-}\underline{\text{after}}\text{-}\pi', \underline{q}_2\text{-}\underline{\text{after}}\text{-}\pi') = (q_1, q_2)$. Since $x/y \in L_{M_2}(q_2) \setminus L_{M_1}(q_1)$, we have $\tau.x/y \in L(M_2) \setminus L(M_1)$ and

$$|\tau.x/y| \leq mn - 1 + 1 = mn < mn + 1 \leq |\pi|.$$

This is a contradiction to the assumption that π is the shortest trace contained in $L(M_2) \setminus L(M_1)$ and completes the proof. □

As a direct consequence of Theorem 4.4, we get the following theorem on I/O-equivalence,

Theorem 4.5 *Let* M_1, M_2 *be two observable FSMs with* n, m *states, respectively. Then*

$$L(M_2) = L(M_1) \quad \textit{if and only if} \quad L(M_2) \cap \bigcup_{i=0}^{mn} (\Sigma_I \times \Sigma_O)^i = L(M_1) \cap \bigcup_{i=0}^{mn} (\Sigma_I \times \Sigma_O)^i.$$

Proof. By applying Theorem 4.4 two times, once for $L(M_2) \cap \bigcup_{i=0}^{mn} (\Sigma_I \times \Sigma_O)^i \subseteq L(M_1) \cap \bigcup_{i=0}^{mn} (\Sigma_I \times \Sigma_O)^i$, and once for $L(M_1) \cap \bigcup_{i=0}^{mn} (\Sigma_I \times \Sigma_O)^i \subseteq L(M_2) \cap \bigcup_{i=0}^{mn} (\Sigma_I \times \Sigma_O)^i$ the theorem is proven. □

Complexity of Reduction Testing. According to Theorem 4.4, it has to be checked that all I/O-traces up to length mn that can be produced by the SUT M_2 are also contained in the language of the reference model M_1. The test cases – that is, the input traces associated with these I/O-traces come from the set

$$\bigcup_{i=0}^{mn} \Sigma_I^i.$$

Applying the formula for the *sum of a geometric progression* [42, p. 31] which is calculated as

$$\sum_{i=0}^{k} x^i = \frac{1 - x^{k+1}}{1 - x},$$

we get an upper bound for the maximal number of test cases required for reduction testing specified by

$$\sum_{i=0}^{mn} |\Sigma_I|^i = \frac{1 - |\Sigma_I|^{mn+1}}{1 - |\Sigma_I|}. \tag{4.1}$$

Asymptotically, this results in

$$O(|\Sigma_I|^{mn}). \tag{4.2}$$

According to the complete testing assumption explained above, there exists a $c \geq 1$ such that every possible output is produced by the SUT when exercising an input trace against the SUT c times. Therefore, the number of test executions is bounded by

$$c \cdot \frac{1 - |\Sigma_I|^{mn+1}}{1 - |\Sigma_I|}. \tag{4.3}$$

If the FSMs are completely specified, the asymptotic estimate given in formula (4.2) becomes a non-asymptotic upper bound: suppose that the SUT produces an erroneous I/O-trace $\overline{x}/\overline{y}'$ for some test case $\overline{x}$ with $|\overline{x}| < mn$. Then we can execute a longer input trace $\overline{x}.\overline{u}$ against the SUT which has exactly length mn. Since the SUT accepts this trace and languages are prefix-closed, it will produce I/O-trace $\overline{x}.\overline{u}/\overline{y}'.\overline{z}$ at least once when executing test case $\overline{x}.\overline{u}$ at least c times. Therefore, the erroneous output $\overline{y}'$ will also be detected when exercising only $\overline{x}.\overline{u}$, but not $\overline{x}$ against the SUT. As a consequence, the number of test executions to be performed is

$$|\Sigma_I|^{mn} \quad \text{for completely specified FSMs,} \tag{4.4}$$

and only the test cases from $\Sigma_I|^{mn}|$ need to be performed. Observe that this does *not* hold for incomplete FSMs, because then it may be the case that non of the suffixes $\overline{u}$ extending $\overline{x}$ to a trace $\overline{x}.\overline{u}$ of length mn are accepted by reference model and/or SUT.

Minimal values for n and m. Theorem 4.4 and Theorem 4.5 do not require minimality of the FSMs M_1 (reference model) and M_2 (system under test).

It is advisable, however, to minimise the reference model and make an estimate for the number of state m in the minimised version of the FSM M_2 reflecting the SUT behaviour. This is because language-equivalent observable FSMs have the smallest possible number of states when being minimised.

Test cases $\overline{x}$ with $|\overline{x}| < mn$ are insufficient for reduction testing. The following example shows that the maximal length mn of test cases to be executed for reduction testing cannot be reduced. Consider the reference model M_1 depicted in Fig. 4.1 and the SUT model M_2 shown in Fig. 4.2. Both FSMs are completely specified, nondeterministic, observable, and minimal with $n = 3n$ $m = 4$, input alphabet $\Sigma_I = \{a\}$ and output alphabet $\Sigma_O = \{0, 1\}$. It is easy to see that all test cases consisting of at most $mn-1 = 11$ a-inputs lead to SUT-outputs that are also possible according to the reference model. Only the test case

$$\underbrace{a \ldots a}_{nm = 12 \text{ times}} \ ,$$

when applied c times to the SUT, will reveal the erroneous output b which is not allowed after 12 a-inputs according to the reference model.

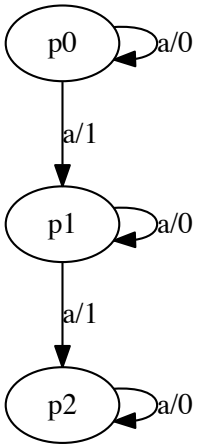


Figure 4.1: Reference model M_1

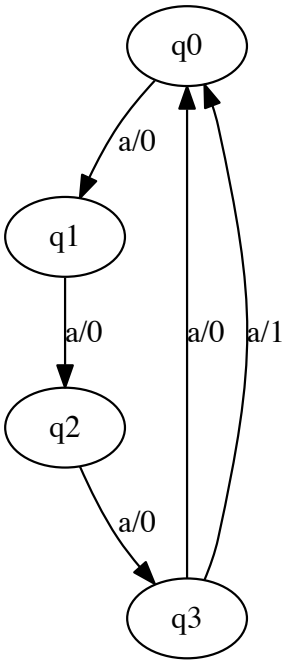


Figure 4.2: SUT model M_2

4.6 The W-Method

The W-Method was one of the first complete strategies for proving I/O-equivalence between deterministic, completely specified FSM by means of testing [69, 9].

In Section 4.3, the notation $\mathrm{DFSM}_C(\Sigma_I, \Sigma_O)$ has been introduced for the set of completely specified deterministic state machines over input alphabet Σ_I and output alphabet Σ_O. This notation is now extended to

$$\mathrm{DFSM}_C(\Sigma_I, \Sigma_O, m) \subseteq \mathrm{DFSM}_C(\Sigma_I, \Sigma_O), \quad m \in \mathbb{N}$$

which denotes the completely specified deterministic FSMs with at most m states.

Throughout this section, it is assumed that the reference model $M_1 \in \mathrm{DFSM_C}(\Sigma_I, \Sigma_O)$ is represented by its minimal equivalent. Recall that an algorithm for minimising DFSMs has been presented in Chapter 3. The true behaviour of the SUT is represented by some $M_2 \in \mathrm{DFSM_C}(\Sigma_I, \Sigma_O)$, and – since we cannot observe M_2 – we might as well assume that it is also minimised.

The following lemma shows that the existence of a homomorphism between completely specified deterministic FSM, which uses the identity functions on input and output alphabets, already implies I/O-equivalence.

Lemma 4.3 *Let* $M_1 = (Q_1, \underline{q}_1, \Sigma_I, \Sigma_O, h_1), M_2 = (Q_2, \underline{q}_2, \Sigma_I, \Sigma_O, h_2)$ *be two complete and deterministic FSMs over the same input alphabet* Σ_I*. Suppose there is a mapping* $f : Q_1 \to Q_2$ *inducing a homomorphism between* M_1 *and* M_2 *by satisfying* $f(\underline{q}_1) = \underline{q}_2$ *and* $(q, x, y, q') \in h_1 \Rightarrow (f(q), x, y, f(q')) \in h_2$ *for all* $q, q' \in Q_1, x \in \Sigma_I, y \in \Sigma_O$*. Then* $L(M_1) = L(M_2)$*, so* $M_1 \sim M_2$ *holds.*

Proof. Suppose that $f : Q_1 \to Q_2$ induces a homomorphism. Let $\overline{x}/\overline{y} = x_1 \dots x_k/y_1 \dots y_k \in L(M_1)$, $k \geq 1$ be a non-empty I/O-sequence. Then, because M_1 is completely specified, there are $q_1, \dots, q_k \in Q_1$ such that $(q_{i-1}, x_i, y_i, q_i) \in h_1$, for all $i = 1 \dots, k$, where $q_0 = \underline{q}_1$. Then $(f(q_{i-1}), x_i, y_i, f(q_i)) \in h_2$, for all $i = 1 \dots, k$, and $f(q_0) = f(\underline{q}_1) = \underline{q}_2$. Hence $\overline{x}/\overline{y} \in L(M_2)$ and $L(M_1) \subseteq L(M_2)$. Since M_1 is complete, for any $\overline{x}/\overline{y} \in L(M_2)$, there exists $\overline{x}/\overline{y}' \in L(M_1) \subseteq L(M_2)$. Since M_2 is deterministic, $\overline{y} = \overline{y}'$. Hence $\overline{x}/\overline{y} \in L(M_1)$ and $L(M_1) = L(M_2)$. □

In the following we use the notation q-<u>after</u>-$\overline{x}$ instead of $\delta(q, \overline{x})$. Since the FSMs under consideration are completely specified and deterministic, q-<u>after</u>-$\overline{x}$ always denotes a single well-defined state.

The test cases $U(\overline{x})$ referenced in the theorem are again the ones already introduced in Section 4.3.

Theorem 4.6 (W-method) *Let* $M = (Q, \underline{q}, \Sigma_I, \Sigma_O, h)$ *be a complete, minimal and deterministic FSM over input alphabet* Σ_I *and output alphabet* Σ_O*. Let* $|Q| = n$ *and* m *an integer with* $m \geq n$*. Let* V *be a state cover of* M *and* W *a characterisation set of* M*. Then, defining*

$$\overline{W} = V.\bigcup_{i=0}^{m-n+1} \Sigma_I^i.W \quad \textit{and} \quad \mathrm{Dom} = \mathrm{DFSM_C}(\Sigma_I, \Sigma_O, m)$$

the test suite

$$\mathbf{TS} = \{\ U(\overline{x}) \mid \overline{x} \in \overline{W}\ \}$$

is complete for fault model $\mathcal{F}(M, \sim, \mathbf{Dom})$.

Proof. Let $M' = (Q', \underline{q}', \Sigma_I, \Sigma_O, h') \in \mathbf{Dom}$.

Suppose $M' \sim M$. Then $M' \overset{\overline{W}}{\sim} M$, since I/O-equivalence is equivalent to $M' \overset{\Sigma_I^*}{\sim} M$ and $\overline{W} \subseteq \Sigma_I^*$.

Now suppose $M' \overset{\overline{W}}{\sim} M$. We are going to show that $M' \sim M$. Let $Q = \{q_1, \ldots q_n\}$ with $\underline{q} = q_1$. Let $V = \{\overline{x}_1, \ldots, \overline{x}_n\}$ with $\underline{q}$-after-$\overline{x}_i = q_i, i = 1, \ldots, n$. Recall that $\varepsilon \in V$, because V is a state cover: $\underline{q}_1 = \underline{q}$-after-$\varepsilon$. Note further, that $W \subseteq \overline{W}$, because $\varepsilon \in V.\Sigma_I^0$.

Let $q'_i = \underline{q}'$-after-$\overline{x}_i$ for $i = 1, \ldots, n$. Then $\underline{q}' \overset{W}{\sim} \underline{q}$, because $W \subseteq \overline{W}$. Furthermore, $\underline{q}' \overset{V.W}{\sim} \underline{q}$, because $V.W \subseteq \overline{W}$. Applying this fact to every $\overline{x}_i \in V$, this yields $q_i \overset{W}{\sim} q'_i$ for each $i = 1, \ldots, n$.

Since W is a characterisation set, $q_i \overset{W}{\not\sim} q_j$, for all $i \neq j$ in range $1, \ldots, n$. Hence also $q'_i \overset{W}{\not\sim} q'_j$, because otherwise $q'_i \overset{W}{\sim} q'_j$, $q'_i \overset{W}{\sim} q_i$, and $q'_j \overset{W}{\sim} q_j$ would imply $q_i \overset{W}{\sim} q_j$, due to transitivity of $\overset{W}{\sim}$, and this contradicts the fact that $q_i \overset{W}{\not\sim} q_j$.

Now $q'_i \overset{W}{\not\sim} q'_j$ for all $i \neq j$ in range $1, \ldots, n$ implies that also the q'_i are pairwise distinct for $i = 1, \ldots, n$, because $\overset{W}{\sim}$ is reflexive. As a consequence, $|\{q'_1, \ldots, q'_n\}| = |\{q_1, \ldots, q_n\}| = n$. Now we have shown that V reaches n distinct states of M', but it could still be possible that M' has *more* than n states, so V is not yet established as a state cover of M'.

By Lemma 4.2,

$$V.\bigcup_{i=0}^{m-n} \Sigma_I^i$$

is a state cover of M', because M' has at most $m - n$ additional states, since it is an element of the fault domain $\mathbf{Dom}$. Hence for any $q' \in Q'$, there exists $\overline{x} \in V.\Sigma_I^j$, for some $j \leq m - n$, such that $q' = \underline{q}'$-after-$\overline{x}$. Let

$q = \underline{q}\text{-}\underline{\text{after}}\text{-}\overline{x} \in Q$ be the corresponding M-state. Since $\underline{q}' \overset{\overline{W}}{\sim} \underline{q}$, this implies

$$\begin{aligned} q' &\overset{W}{\sim} q \quad \text{because} \quad \{\overline{x}\}.W \subseteq \overline{W} \\ q' &\overset{\Sigma_I}{\sim} q \quad \text{because} \quad \{\overline{x}\}.\Sigma_I.W \subseteq \overline{W} \\ q' &\overset{\Sigma_I.W}{\sim} q \quad \text{because} \quad \{\overline{x}\}.\Sigma_I.W \subseteq \overline{W} \end{aligned}$$

As a consequence

$$\forall x \in \Sigma_I : \omega(q', x) = \omega(q, x)$$

and this implies $(q', x, y, q'\text{-}\underline{\text{after}}\text{-}x) \in h' \Rightarrow (q, x, y, q\text{-}\underline{\text{after}}\text{-}x) \in h$, and $(q'\text{-}\underline{\text{after}}\text{-}x) \overset{W}{\sim} (q\text{-}\underline{\text{after}}\text{-}x)$ for all $x \in \Sigma_I$. Summarising, we have shown that for any reachable state q' of M', there exists a reachable, W-equivalent state q in M. Since W is a characterisation set of M, this state q is uniquely determined.

Define $f : Q' \to Q$ by $f(q') = q \Leftrightarrow q' \overset{W}{\sim} q$. Then $f(\underline{q}') = \underline{q}$. The proof steps above imply that for all $q' \in Q'$, $x \in \Sigma_I$, and $y \in \Sigma_O$

$$(q', x, y, q'\text{-}\underline{\text{after}}\text{-}x) \in h' \Rightarrow (f(q'), x, y, f(q')\text{-}\underline{\text{after}}\text{-}x) \in h$$

and

$$(q'\text{-}\underline{\text{after}}\text{-}x) \overset{W}{\sim} (f(q')\text{-}\underline{\text{after}}\text{-}x)$$

which implies $f(q'\text{-}\underline{\text{after}}\text{-}x) = f(q')\text{-}\underline{\text{after}}\text{-}x$ and

$$(q', x, y, q'\text{-}\underline{\text{after}}\text{-}x) \in h' \Rightarrow (f(q'), x, y, f(q'\text{-}\underline{\text{after}}\text{-}x)) \in h$$

Therefore f is a homomorphism from M' to M. By Lemma 4.3 we have $L(M') = L(M)$ which completes the proof. □

Corollary 4.2 *Let $M = (Q, \underline{q}, \Sigma_I, \Sigma_O, h)$ be a complete, minimal and deterministic FSM over input alphabet Σ_I and output alphabet Σ_O. Let $|Q| = n$ and m an integer with $m \geq n$. Define*

$$Dom = DFSM_C(\Sigma_I, \Sigma_O, m)$$

Then the test suite

$$\mathbf{TS} = \{\ U(\overline{x}) \mid \overline{x} \in \bigcup_{i=0}^{m+n-1} \Sigma_I^i\ \}$$

is complete for fault model $\mathcal{F}(M, \sim, Dom)$.

Proof.

By Lemma 4.2 and Lemma 3.4 there exist a state cover $\varepsilon \in V$ of M and a characterisation set W of M such that

$$V, W \subseteq \bigcup_{i=0}^{n-1} \Sigma_I^i$$

because M has n states.

Define $\overline{W} := V.\bigcup_{i=0}^{m-n+1} \Sigma_I^i.W$. Then $\overline{W} \subseteq \bigcup_{i=0}^{m+n-1} \Sigma_I^i \subseteq \Sigma_I^*$. From the W-method we have $\underline{q}' \sim \underline{q} \Rightarrow \underline{q}' \overset{\bigcup_{i=0}^{m+n-1} \Sigma_I^i}{\sim} \underline{q} \Rightarrow \underline{q}' \overset{\overline{W}}{\sim} \underline{q} \Rightarrow \underline{q}' \sim \underline{q}$ for any $M' = (Q', \underline{q}', \Sigma_I, \Sigma_O, h') \in \mathbf{Dom}$. □

Complexity considerations Let's analyse the test complexity advantages gained when using the W-Method instead of the product-automata technique presented in Section 4.5 (remember that the latter's test complexity was $|\Sigma_I|^{mn}$).

To this end, we need an estimate for the cardinality of $\overline{W}$, which is the product of

- the number of elements in the state cover V,
- the number of elements in $\bigcup_{i=0}^{m-n+1} \Sigma_I^i$, and
- the number of elements in the characterisation set W.

The state cover contains n input traces to reach each of the n non-equivalent states of M.

Using the formula for the *sum of a geometric progression* [42, p. 31] which is calculated as

$$\sum_{i=0}^{k} x^i = \frac{1 - x^{k+1}}{1 - x},$$

the number of elements in $\bigcup_{i=0}^{m-n+1} \Sigma_I^i$ is determined by

$$\left| \bigcup_{i=0}^{m-n+1} \Sigma_I^i \right| = \sum_{i=0}^{m-n+1} |\Sigma_I|^i = \frac{1 - |\Sigma_I|^{m-n+2}}{1 - |\Sigma_I|}$$

For the characterisation set W, suppose $W = \{\overline{x}_1, \ldots, \overline{x}_k\}$ is a minimal characterisation set and define $W_i = \{\overline{x}_1, \ldots, \overline{x}_i\}$, $\forall i = 1, \ldots, k$. Then $W_1 \subset$

$W_2 \subset \cdots \subset W_k = W$. W_1 separates the state space Q into n_1 equivalence classes with equivalence relation $\overset{W_1}{\sim}$. Then $n_1 \geq 2$, since W_1 must distinguish at least two states – otherwise it would not be contained in W. Since W is minimal, W_i distinguishes the state space Q into n_i classes and $n_i \geq n_{i-1}+1$, for all $i = 2, \ldots k$, and $n_k = n$. Hence

$$n = n_k \geq n_{k-1} + 1 \geq n_{k-2} + 2 \geq \cdots \geq n_1 + (k-1) \geq k+1$$

so

$$|W| = k \leq n - 1$$

Summarising, this results in the approximation

$$|\overline{W}| \leq n \cdot \frac{1 - |\Sigma_I|^{m-n+2}}{1 - |\Sigma_I|} \cdot (n-1) = (n^2 - n) \cdot \frac{1 - |\Sigma_I|^{m-n+2}}{1 - |\Sigma_I|}$$

Asymptotically, we get the test complexity

$$O(n^2 \cdot |\Sigma_I|^{m-n+1}),$$

so the good new is, that the W-Method reduces the exponential test complexity of the product-automaton algorithm to polynomial complexity in the case where the SUT is known to have just as many states as the reference model, i.e. $m = n$. Otherwise it is exponential in the *difference* $m - n$ of states in M_2 and M_1.

Let us now consider the worst-case number of test steps that may occur in a test case. The longest possible input sequence in a transition cover V has $(n-1)$ elements; the longest sequences in $\bigcup_{i=0}^{m-n+1} \Sigma_I^i$ have length $m-n+1$, and from [21, Section 4.5] it is known that in the worst case, the elements of W have length $n-1$. This results in a maximal length of

$$(n + m - 1) \qquad \text{test steps in a test case}$$

For the case $m = n$, this results asymptotically in

$$O(n^3 \cdot |\Sigma_I|)$$

test steps to be performed over all test cases.

Example 4. The following example has been adopted from [9, Fig. 6]. Let $Q = \{q_1, q_2\}$, $\Sigma_I = \{a, b\}$, $\Sigma_O = \{0, 1\}$. Let M, M' be two minimal and

completely specified DFSMs over (Σ_I, Σ_O) with state space Q and initial state q_1. The transition relations are given by

$$h_M = \{(q_1, a, 0, q_2), (q_1, b, 1, q_2), (q_2, a, 1, q_1)(q_2, b, 0, q_2)\}$$

and

$$h_{M'} = \{(q_1, a, 0, q_2), (q_1, b, 1, q_2), (q_2, a, 1, q_2)(q_2, b, 0, q_2)\}.$$

The machines are shown in Fig. 4.3.

M and M' are not I/O equivalent since $a.a.a/0.1.0 \in L(M) \setminus L(M')$: there is a transition fault in M' with the transition $(q_2, a, 1, q_2) \in h_{M'}$ but $(q_2, a, 1, q_1) \in h_M$. For applying the W-Method, we observe that

1. $V = \{\varepsilon, a\}$ a state cover of M and
2. $W = \{a\}$ a characterisation set of M. Then
3. $V \cup V.\Sigma_I = \{\varepsilon, a, b, a.a, a.b\}$ is a transition cover of M.
4. $\overline{W} = (V \cup V.\Sigma_I).W = \{a, a.a, b.a, a.a.a, a.b.a\}$ is the complete test suite generated by the W-Method under the assumption that $m = n$.

Since $M \overset{V \cup V.\Sigma_I}{\sim} M'$, the test suite $\{U(\overline{x}) \mid \overline{x} \in V \cup V.\Sigma_I\}$ is not able to detect the transition fault in M'. But test suite $\{U(\overline{x}) \mid \overline{x} \in \overline{W}\}$ can detect the transition fault by applying the test case $U(a.a.a)$.

□

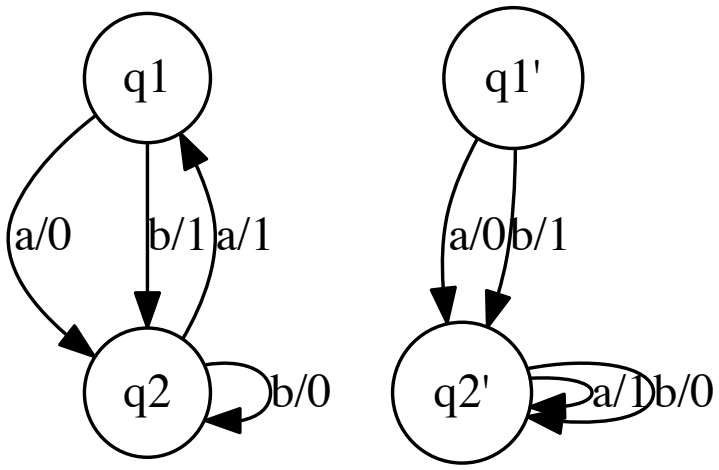


Figure 4.3: Single transition fault

4.7 The H-Method

4.7.1 Motivation

As discussed in [13], complete test methods for FSMs depend on the technique how to distinguish different states in a crucial way. The choice of these distinction methods is also the main "tool" for reducing the overall number of test cases without losing completeness: in any complete test strategy, we need to visit every state of the reference model, check correctness of input/output behaviour from there and explore possibly uncovered states of the SUT with all traces up to length. Therefore, all input traces of the state cover V, followed by traces from the set $\sum_{i=0}^{m-n+1} \Sigma_I^i$ of arbitrary input traces up to length $(m - n + 1)$ need to be explored in any method.

These considerations have led to the development of the *H-Method*[1], originally published in [13]. In Fig. 4.4, it can be seen that the H-Method significantly reduces the test suite size, when compared to the W-Method discussed in Section 4.6. We will not discuss the HSI-Method in these lecture notes, since the H-Method is a refinement thereof. The authors of the SPY method [63] state that SPY outperforms all other known FSM testing methods for equivalence. The results from [15] shown in Fig. 4.4, however, do not confirm this statement. The *P-Method* shown in Fig. 4.4 is only applicable for the case $\mathfrak{m} = \mathfrak{n}$, i.e. the assumption that the SUT does not have more distinguishable states than the reference model must be valid.

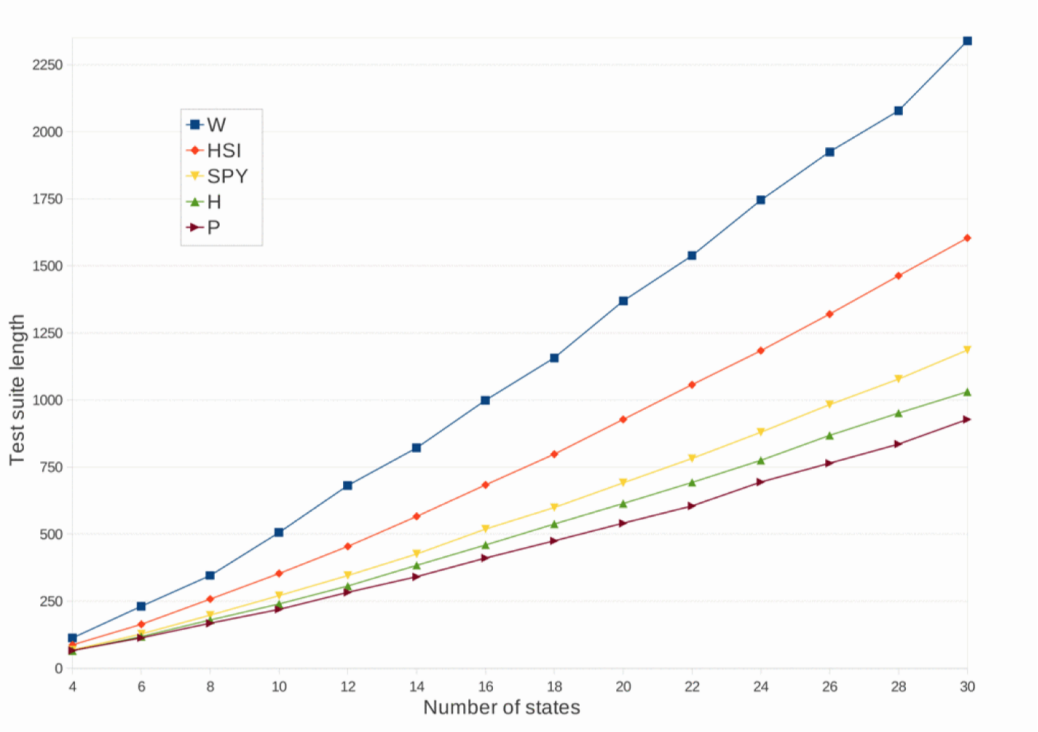


Figure 4.4: Comparison of various complete FSM-based test methods, from [15].

The H-Method has the following attractive features.

[1]The "H" stands for *harmonised* state identifiers (i.e., distinguishing traces); this term was introduced in [60].

- It guarantees completeness for arbitrary $m \geq n$ with significantly smaller test suites, when compared to other complete methods.
- It is applicable to deterministic or nondeterministic, complete or incomplete observable models.
- The W-Method presented in Section 4.6 and the Wp-Method from Section 4.8.1 ar direct corollaries of the H-Method.

4.7.2 Definitions related to the H-Method

For the remainder of this section, let $M = (Q, \underline{q}, \Sigma_I, \Sigma_O, h)$ be a minimal and observable FSM over $\Sigma = \Sigma_I \times \Sigma_O$ with finite state space Q, $|Q| = n$. Let

$$\omega(q, x) = \{y \in \Sigma_O \mid \exists q' \in Q, (q, x, y, q') \in h\}$$

be the set-valued function returning all possible FSM outputs to a given state q and input x. Note that $\omega(q, x) = \varnothing$ if M is not completely specified and input x is undefined in state q. This set-valued output function can be extended to input traces by setting

$$\begin{aligned} \omega &: Q \times \Sigma_I^* \to \mathbb{P}(\Sigma_O^*) \\ \omega(q, \varepsilon) &= \{\varepsilon\} \\ \omega(q, \overline{x}) &= \{\overline{y} \in \Sigma_O^* \mid \exists q' : (q, \overline{x}, \overline{y}, q') \in h\} \quad \text{for} \quad |\overline{x}| > 0 \end{aligned}$$

Recall that $q\text{-}\underline{\text{after}}\text{-}x = \{q' \in Q \mid \exists y \in \Sigma_O, (q, x, y, q') \in h\}$ and $q\text{-}\underline{\text{after}}\text{-}x/y = q'$ if and only if $(q, x, y, q') \in h$.

Definition 4.6 *Let $\overline{x} \in \Sigma_I^*$ be any input sequence. We call $\overline{x}$ is a* distinguishing trace *of states $q_1, q_2 \in Q$, if $\omega(q_1, \overline{x}) \neq \omega(q_2, \overline{x})$. By $\Delta(q_1, q_2) \subseteq \Sigma_I^*$ we denote the set of all distinguishing traces of q_1, q_2. The* distinguishing I/O-traces *of states $q_1, q_2 \in Q$ are specified by*

$$\Delta_{IO}(q_1, q_2) = \{\overline{x}/\overline{y} \mid \overline{x}/\overline{y} \in (L(q_1) \setminus L(q_2)) \cup (L(q_2) \setminus L(q_1))\}.$$

Observe that $\overline{x}/\overline{y} \in \Delta_{IO}(q_1, q_2)$ always implies that $\overline{x} \in \Delta(q_1, q_2)$.

Definition 4.7 *Let $W \subseteq \Sigma_I^*$. W is called a* characterisation set *of M, if $W \cap \Delta(q_1, q_2) \neq \varnothing$ holds for any $q_1 \neq q_2 \in Q$.*

4.7.3 H-Method Theorems

The first theorem states the H-Method as a language-theoretic insight.

Theorem 4.7 (H-Method) *Let* M *be defined as given above. Let* $\varepsilon \in V = \{v_1, \ldots, v_n\} \subseteq \Sigma_I^*$ *and* $\varepsilon \in \Pi = \{v_1/u_1, \ldots, v_n/u_n\} \subseteq L(M)$ *such that* $\{q\text{-}\underline{\text{after}}\text{-}v_i/u_i \mid i = 1, \ldots, n\} = Q$. *For* $m \geq n$, *define fault domain* $\mathcal{D}$ *as the set of all minimal and observable FSM over* Σ *with at most* m *states. Let*

$$A = \Pi \times \Pi,$$

$$B = \Pi \times \Pi.\bigcup_{k=1}^{m-n+1} \Sigma^k$$

$$C = \bigcup_{\pi \in \Pi, \tau \in \bigcup_{k=1}^{m-n+1} \Sigma^k} \{\pi\}.\mathrm{pref}(\tau) \times \{\pi.\tau\}$$

Let $\mathbf{TS}_H \subseteq \Sigma^*$ *be a set containing*

- $\Pi.\bigcup_{k=0}^{m-n+1} \Sigma^k$ *and*
- $\{\alpha.w, \beta.w\}$, *for any* $(\alpha, \beta) \in A \cup B \cup C$, $q\text{-}\underline{\text{after}}\text{-}\alpha \neq q\text{-}\underline{\text{after}}\text{-}\beta$ *and some* $w \in \Sigma^*$ *with* $L(q\text{-}\underline{\text{after}}\text{-}\alpha) \cap \{w\} \neq L(q\text{-}\underline{\text{after}}\text{-}\beta) \cap \{w\}$.

Then

$$L(M') = L(M) \Leftrightarrow L(M') \cap \mathbf{TS}_H = L(M) \cap \mathbf{TS}_H$$

Proof. It is trivial that $L(M') = L(M) \Rightarrow L(M') \cap \mathbf{TS}_H = L(M) \cap \mathbf{TS}_H$. Therefore, the following proof only shows $L(M') \cap \mathbf{TS}_H = L(M) \cap \mathbf{TS}_H \Rightarrow L(M') = L(M)$.

Suppose that $L(M') \cap \mathbf{TS}_H = L(M) \cap \mathbf{TS}_H$ holds, but $L(M') \neq L(M)$. Then there exist I/O sequences which are contained in $L(M) \setminus L(M')$ or $L(M') \setminus L(M)$. Define $\Delta_{IO}(M, M') = \{\iota \in \Sigma^* \mid L(M') \cap \{\iota\} \neq L(M) \cap \{\iota\}\}$. Then $\Delta_{IO}(M, M') \neq \varnothing$.

Define $\Omega = \{\tau \in \Sigma^* \mid \exists \pi \in \Pi, L(M') \cap \{\pi.\tau\} \neq L(M) \cap \{\pi.\tau\}\}$. Since $\varepsilon \in \Pi$, $\Delta_{IO}(M, M') \subseteq \Omega \neq \varnothing$. Let $\tau \in \Omega$ be a shortest trace and $\pi \in \Pi$ satisfying $L(M') \cap \{\pi.\tau\} \neq L(M) \cap \{\pi.\tau\}$.

Since $L(M') \cap \mathbf{TS}_H = L(M) \cap \mathbf{TS}_H$, we conclude that $\pi.\tau \notin \mathbf{TS}_H$, and, therefore, $|\tau| > m - n + 1$. Since $\Pi.\bigcup_{i=0}^{m-n+1} \Sigma^i \subseteq \mathbf{TS}_H$, we have

$L(M') \cap \Pi.\bigcup_{i=0}^{m-n+1} \Sigma^i = L(M) \cap \Pi.\bigcup_{i=0}^{m-n+1} \Sigma^i$ and, consequently, $\Pi \subseteq L(M')$. Furthermore, any proper prefix τ' of τ must satisfy $\pi.\tau' \in L(M) \cap L(M')$, because otherwise τ is not the shortest element of Ω.

Let $\tau = \sigma_1 \ldots \sigma_k, k > m - n + 1$. and $\tau_i := \sigma_1 \ldots \sigma_i, i = 1, \ldots, m - n + 1$. Then $\pi.\sigma_1 \ldots \sigma_i \in L(M) \cap L(M'), i = 1, \ldots, m - n + 1$. Consider $\underline{q}'\text{-}\underline{\text{after}}\text{-}\pi_1, \ldots, \underline{q}'\text{-}\underline{\text{after}}\text{-}\pi_n, \underline{q}'\text{-}\underline{\text{after}}\text{-}\pi.\tau_1, \ldots, \underline{q}'\text{-}\underline{\text{after}}\text{-}\pi.\tau_{m-n+1} \in Q'$. These -$\underline{\text{after}}$-expressions specify $m + 1$ states. Therefore, since $|Q'| = m$, at least two of these states must be the same, and we can distinguish the following three cases.

$$\exists i < j \in \{1, \ldots, n\} : \underline{q}'\text{-}\underline{\text{after}}\text{-}\pi_i = \underline{q}'\text{-}\underline{\text{after}}\text{-}\pi_j \tag{4.5}$$

$$\exists i \in \{1, \ldots, n\}, j \in \{1, \ldots, m - n + 1\} : \underline{q}'\text{-}\underline{\text{after}}\text{-}\pi_i = \underline{q}'\text{-}\underline{\text{after}}\text{-}\pi.\tau_j \tag{4.6}$$

$$\exists 1 \leq i < j \leq m - n + 1 : \underline{q}'\text{-}\underline{\text{after}}\text{-}\pi.\tau_i = \underline{q}'\text{-}\underline{\text{after}}\text{-}\pi.\tau_j \tag{4.7}$$

Now we have shown the existence of $\alpha, \beta \in L(M')$ satisfying $(\alpha, \beta) \in A \cup B \cup C$ and $\underline{q}'\text{-}\underline{\text{after}}\text{-}\alpha = \underline{q}'\text{-}\underline{\text{after}}\text{-}\beta$ in M'.

Suppose that $\underline{q}\text{-}\underline{\text{after}}\text{-}\alpha \neq \underline{q}\text{-}\underline{\text{after}}\text{-}\beta$ in M. Then there is some $w \in \Delta_{IO}(\underline{q}\text{-}\underline{\text{after}}\text{-}\alpha, \underline{q}\text{-}\underline{\text{after}}\text{-}\beta)$ such that

$$\{\alpha.w, \beta.w\} \subseteq \mathbf{TS}_H \text{ and } |\{\alpha.w, \beta.w\} \cap L(M)| = 1.$$

Since $\underline{q}'\text{-}\underline{\text{after}}\text{-}\alpha = \underline{q}'\text{-}\underline{\text{after}}\text{-}\beta$,

$$|\{\alpha.w, \beta.w\} \cap L(M')| \in \{0, 2\}$$

follows. This shows that

$$L(M') \cap \{\alpha.w, \beta.w\} \neq L(M) \cap \{\alpha.w, \beta.w\}$$

and implies $L(M') \cap \mathbf{TS}_H \neq L(M) \cap \mathbf{TS}_H$, a contradiction to the assumption $L(M') \cap \mathbf{TS}_H = L(M) \cap \mathbf{TS}_H$. Hence $\underline{q}\text{-}\underline{\text{after}}\text{-}\alpha = \underline{q}\text{-}\underline{\text{after}}\text{-}\beta$ in M.

Now we apply the case analysis from formulas (4.5) to (4.7) to this pair (α, β).

1. If Formula (4.5) applies, this means that $\alpha = \pi_i$ and $\beta = \pi_j$ with traces $\pi_i, \pi_j \in \Pi$. By definition of Π, however, $\underline{q}\text{-}\underline{\text{after}}\text{-}\pi_i \neq \underline{q}\text{-}\underline{\text{after}}\text{-}\pi_j$ holds. This is a contradiction to the fact that $\underline{q}\text{-}\underline{\text{after}}\text{-}\alpha = \underline{q}\text{-}\underline{\text{after}}\text{-}\beta$ as shown above.

2. If Formula (4.6) applies, we have $\alpha = \pi_i$ and $\beta = \pi.\tau_j$. Let $\iota = \sigma_{j+1} \dots \sigma_k$, so that $\pi.\tau_j.\iota = \pi.\tau$. Then $|\iota| < |\tau|$ and $\pi.\tau = \beta.\iota$. Since $L(\underline{q}\text{-}\underline{after}\text{-}\alpha) = L(\underline{q}\text{-}\underline{after}\text{-}\beta)$ and $L'(\underline{q}'\text{-}\underline{after}\text{-}\alpha) = L'(\underline{q}'\text{-}\underline{after}\text{-}\beta)$,

$$\pi.\tau \in L(M) \Leftrightarrow \alpha.\iota \in L(M) \text{ and } \pi.\tau \in L(M') \Leftrightarrow \alpha.\iota \in L(M')$$

follows. Since $L(M') \cap \{\pi.\tau\} \neq L(M) \cap \{\pi.\tau\}$, we have

$$L(M') \cap \{\alpha.\iota\} \neq L(M) \cap \{\alpha.\iota\},$$

which is equivalent to $L(M') \cap \{\pi_i.\iota\} \neq L(M) \cap \{\pi_i.\iota\}$, a contradiction to the fact that τ is a shortest element of Ω.

3. If Formula (4.7) applies, we have the case $\alpha = \pi.\tau_i$ and $\beta = \pi.\tau_j$ with $1 \leq i < j \leq m - n + 1$. As in the previous case, let $\iota = \sigma_{j+1} \dots \sigma_k$, so that $\pi.\tau_j.\iota = \pi.\tau$. Let $\iota' = \tau_i.\iota$. Then $|\iota'| < |\tau|$, $\pi.\tau = \beta.\iota$, and $\alpha.\iota = \pi.\tau_i.\iota = \pi.\iota'$. Since $L(M') \cap \{\pi.\tau\} \neq L(M) \cap \{\pi.\tau\}$, we have, just as in the previous case,

$$L(M') \cap \{\alpha.\iota\} \neq L(M) \cap \{\alpha.\iota\},$$

which is equivalent to $L(M') \cap \{\pi.\iota'\} \neq L(M) \cap \{\pi.\iota'\}$. Again, this is a contradiction to the fact that τ is a shortest element of Ω.

Now we have derived a contradiction for all possible cases of α and β. As a consequence, the assumption $L(M') \neq L(M)$ must be wrong, and this completes the proof. □

The following corollary re-formulates Theorem 4.7 as a test strategy.

Corollary 4.3 *Let M be an minimal and observable FSM over $\Sigma = \Sigma_I \times \Sigma_O$ with finite state space Q, $|Q| = n$. (M is not necessary deterministic or completely specified.) Let $\varepsilon \in V = \{v_1, \dots, v_p\} \subseteq \Sigma_I^*$ be a state cover of M. Let $m \geq n$, $\mathcal{F} = (\mathcal{M}, \sim, \mathcal{D})$, where $\mathcal{D}$ is the set of all minimal and observable FSM over Σ with at most m states.*

Let

$$A = V \times V,$$

$$B = V \times V.\bigcup_{k=1}^{m-n+1} \Sigma_I^k$$

$$C = \{(\alpha, \beta) \mid \alpha, \beta \in V.\bigcup_{k=1}^{m-n+1} \Sigma_I^k, \alpha \in \mathit{pref}(\beta)\}$$

Then the test suite $\mathbf{TS}_H \subseteq \Sigma_I^*$ *containing*

- $V.\bigcup_{k=0}^{m-n+1} \Sigma_I^k$ *and*
- $\{\alpha.w, \beta.w\}$*, for any* $(\alpha, \beta) \in A \cup B \cup C$, q-after-$\alpha \neq q$-after-β *and some* $w \in \Sigma_I^*$ *with* $w \in \Delta(q, q')$, $q \in q$-after-$\alpha \neq q' \in q$-after-β,

is complete for $\mathcal{F} = (\mathcal{M}, \sim, \mathcal{D})$.

The following corollary shows that the W-Method is a simple consequence of Theorem 4.7, and that it holds also for nondeterministic incomplete FSMs.

Corollary 4.4 *Let* M *be an minimal and observable FSM over* $\Sigma = \Sigma_I \times \Sigma_O$ *with finite state space* Q, $|Q| = n$. *(*M *is not necessary deterministic or completely specified.) Let* $\varepsilon \in V = \{v_1, \ldots, v_p\} \subseteq \Sigma_I^*$ *be a state cover of* M. *Let* $W \subseteq \Sigma_I^*$ *be a characterisation set of* M. *Let* $m \geq n$, $\mathcal{F} = (\mathcal{M}, \sim, \mathcal{D})$, *where* $\mathcal{D}$ *is the set of all minimal and observable FSM over* Σ *with at most* m *states.*

Then the test suite $\mathbf{TS}_W = V.\bigcup_{k=0}^{m-n+1} \Sigma_I^k.W$ *is complete for* $\mathcal{F} = (\mathcal{M}, \sim, \mathcal{D})$.

4.8 FSM Testing Theories for Nondeterministic Systems

4.8.1 A Nondeterministic Variant of the Wp-Method

The Wp-Method originally presented in [19] was an improvement of the W-Method (Section 4.6) with lower test complexity. We present here a further extension of the Wp-Method elaborated in [45], which is applicable to nondeterministic FSMs. The conformance relation under consideration is I/O-equivalence $\sim$, just as for the original W-Method. We will discuss later on that for nondeterministic systems, complete test suites for reduction $\preceq$ are more suitable in most application cases.

In analogy to the W-Method, test cases of the Wp-Method are again represented by a set $\overline{Wp} \subseteq \Sigma_I^*$, and this set is constructed using the following algorithm.

1. **Input** to the algorithm: an observable, minimal FSM $M = (Q, \underline{q}, \Sigma_I, \Sigma_O, h)$.

2. **Output** of the algorithm: a test suite $\overline{W\!p}$.

3. Calculate a state cover SCOV of M.

4. Calculate a characterisation set W of M.

5. Calculate *state identification sets* $\{W_0, \ldots, W_{|Q|-1}\}$ of M, such that[2]

   - $W_i \subseteq \mathrm{pref}(W)$ for $i = 0, \ldots, |Q| - 1$.
   - W_i distinguishes q_i from all other states in Q.

6. Set
$$W\!p_1 = \mathrm{SCOV}.\Big(\bigcup_{i=0}^{m-n} \Sigma_I^i\Big).W$$

7. Set
$$W\!p_2 = \mathrm{SCOV}.\Sigma_I^{m-n+1} \oplus \{W_0, \ldots, W_{|Q|-1}\},$$
   where for any $V \subseteq \Sigma_I^*$, the $\oplus$-operator is defined by
$$V \oplus \{W_0, \ldots, W_{|Q|-1}\} = \bigcup \big\{\, \{\overline{x}\}.W_i \mid i \in \{0, \ldots, |Q| - 1\} \wedge \overline{x} \in V \wedge q_i \in \underline{q}\text{-}\underline{\mathrm{after}}\text{-}\overline{x} \,\big\}$$

8. Set $\overline{W\!p} = W\!p_1 \cup W\!p_2$.

9. Return $\overline{W\!p}$.

In the algorithm above, the specification of $W\!p_2$ deserves some explanation. Intuitively speaking, $W\!p_2$ contains input sequences from $\mathrm{SCOV}.\Sigma_I^{m-n+1}$ that are extended by sequences from one or more W_i according to the following recipe: Given an input sequence $\overline{x}$ in $\mathrm{SCOV}.\Sigma_I^{m-n+1}$, consider all target states that are reachable from the initial state by applying $\overline{x}$. These target states are specified by the set $\underline{q}$-$\underline{\mathrm{after}}$-$\overline{x}$. Now $\overline{x}$ is extended by every trace from W_i, if and only if q_i is among these target states reachable under $\overline{x}$.

Remark 4.1 *The set $W\!p_2$ of Wp-Method was originally presented by*
$$W\!p_2' = R.\Sigma_I^{m-n} \oplus \{W_0, \ldots, W_{|Q|-1}\}$$
where $R = \mathrm{SCOV}.\Sigma_I \setminus \mathrm{SCOV}$. *Obviously,* $\overline{W\!p} = W\!p_1 \cup W\!p_2 = W\!p_1 \cup W\!p_2'$.

[2] $\mathrm{pref}(W)$ is the set of all prefixes of all input traces in W.

Theorem 4.8 *The Wp-Method described by the algorithm above is complete for all fault models* $\mathcal{F} = (M, \sim, \mathrm{Dom})$*, where* M *is an observable minimal FSM with* n *states, and fault domain* Dom *contains all FSMs over the same input and output alphabets as* M*, whose prime machines have at most* $m \geq n$ *states.*

Corollary 4.5 *The W-Method is complete for all fault models* $\mathcal{F} = (M, \sim, \mathrm{Dom})$*, where* M *is an observable minimal FSM with* n *states, and fault domain* Dom *contains all FSMs over the same input and output alphabets as* M*, whose prime machines have at most* $m \geq n$ *states.*

When testing nondeterministic systems, completeness only holds under the *complete testing assumption* [26]: this is a fairness hypothesis stating the existence of some $k > 0$, so that, when applying test case U to an implementation M' k times, every input/output sequence $\overline{x}/\overline{y}$ that *can* be observed with $M \cap U$ *will* be observed.

The Wp-Method cannot prove reduction The following example shows that the Wp-Method is *not* exhaustive (and therefore not complete) when testing whether the implementation is a reduction of the reference model.

Example 5. Let $M_i = (Q_i, \underline{q}_i, \Sigma_I, \Sigma_O, h_i)$, $i = 1, 2, 3$ be three FSMs, with $Q_i = \{s_0, s_1, s_2, s_3\}$, $\underline{q}_i = s_0$, $\Sigma_I = \{a, b\}$, $\Sigma_O = \{d, e, f\}$, with transition graphs as shown in Fig. 4.5, 4.6, and 4.7. DFSM M_1 is the reference model. Implementation M_2 has a transition fault in the outgoing transition from s_3 labelled by a/f. Implementation M_3 has the same error, but additionally the transition $s_0 \xrightarrow{a/e} s_3$ from M_1 is missing. Both implementations have the same number of states as the reference model.

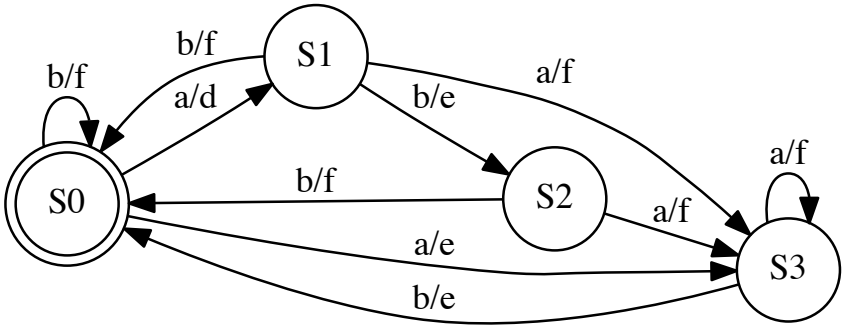


Figure 4.5: M_1 – reference model.

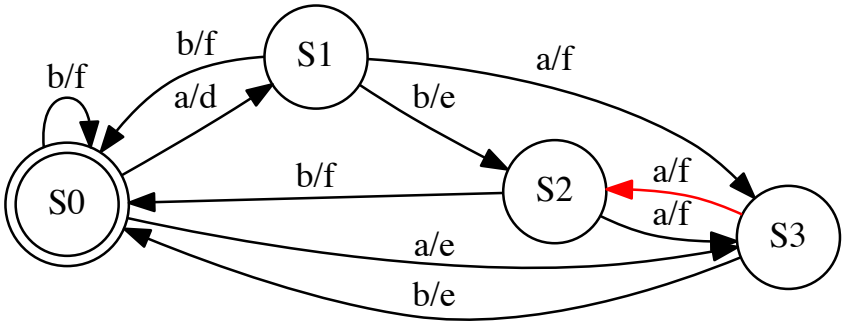


Figure 4.6: M_2

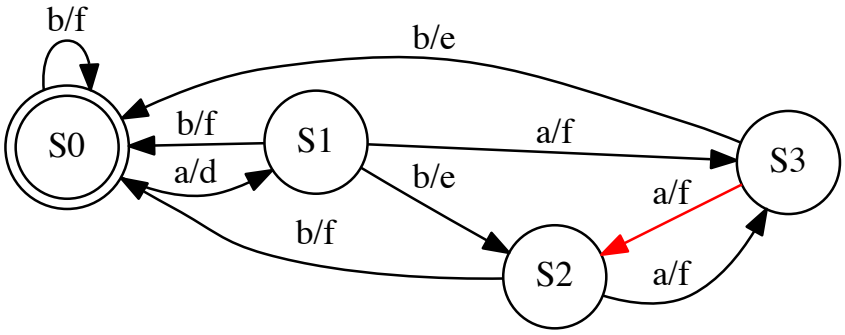


Figure 4.7: M_3

□

Constructing a Wp-Test Suite from reference model M_1 leads to the following results. $W = \{a, b\}$ is a characterisation set and $W_0 = \{a\}$, $W_1 = W_3 = \{b\}$, $W_2 = \{a, b\}$ are state identification sets. The set $V = \{\varepsilon, a, a.b\}$ is a state cover, the Wp-Test Suite looks as follows.

$$\begin{aligned} \mathbf{TS} &= (V.W) \cup (V.\Sigma_I \oplus \{W_0, \ldots, W_3\}) \\ &= \{a, b, a.a, a.b, a.b.a, a.b.b, b.a, a.a.b, a.b.a.b, a.b.b.a\} \end{aligned}$$

The following table shows the outputs produced by each M_i when applying this test suite.

Input	Output(M_1)	Output(M_2)	Output(M_3)
a	d, e	d, e	d
b	f	f	f
a.a	d.f, e.f	d.f, e.f	d.f
a.b	d.e, d.f, e.e	d.e, d.f, e.e	d.e, d.f
a.b.a	d.e.f, d.f.d, d.f.e e.e.d, e.e.e	d.e.f, d.f.d, d.f.e, e.e.d, e.e.e	d.e.f, d.f.d
a.b.b	d.e.f, d.f.f, e.e.f	d.e.f, d.f.f, e.e.f	d.e.f, d.f.f
b.a	f.d, f.e	f.d, f.e	f.d
a.a.b	d.f.e, e.f.e	d.f.e, e.f.f	d.f.e
a.b.a.b	d.e.f.e, d.f.d.f, d.f.d.e, d.f.e.e, e.e.d.e, e.e.d.f, e.e.e.e	d.e.f.e, d.f.d.f, d.f.d.e, d.f.e.e, e.e.d.e, e.e.d.f, e.e.e.e	d.e.f.e, d.f.d.f, d.f.d.e
a.b.b.a	d.e.f.d, d.e.f.e, d.f.f.d, d.f.f.e, e.e.f.d, e.e.f.e	d.e.f.d, d.e.f.e, d.f.f.d, d.f.f.e, e.e.f.d, e.e.f.e	d.e.f.d, d.f.f.d

In M_1, the outputs produced by applying input sequence a.a.b are $\{d.f.e, e.f.e\}$ but in M_2, $\{d.f.e, e.f.f\}$. Hence $M_1 \not\simeq M_2$, and the transition fault $(s_3, a, f, s_2) \in h_2 \setminus h_1$ can be detected by using the test suite TS.

M_3 is not a reduction of M_1, since the transition fault $(s_3, a, f, s_2) \in h_3 \setminus h_1$ exists. But in M_3, the transition $(s_0, a, e, s_3) \in h_1 \setminus h_3$ doesn't exist. If we are only testing for reduction, this is not an error. Now the outputs produced in M_3 by TS can all be accepted by M_1, because the transition (s_0, a, e, s_3) would have been a "short cut" to the state s_3 from where the transition fault could be uncovered by means of input sequence a.a.b. Intuitively speaking, the test cases generated by the Wp-Method are too short to uncover the fault, and the missing "short cut" transition does not lead to an error since we are testing for reduction. As a consequence, the test suite TS is not suitable to detect the fault for reduction.

Theorem 4.9 *The Wp-Method never generates more test cases than the W-Method.*

Proof. Suppose each state identification set W_i, $i = 0, \ldots, n-1$, is a subset of W (this means that we do not care about the possibility that a member of a state identification set could be a *prefix* of an input trace of the characterisation set: we always take a full-length member of W to insert into the respective $W = i$). Then the test suite $\overline{Wp}$ is a subset of the test suite $\overline{W}$. Therefore $\overline{Wp}$ contains at most many test cases as in $\overline{W}$.
Let SCOV be a state cover, W a characterisation set. Let identification sets $W_i \subseteq W$, $\forall i = 0, \ldots, n-1$. Let $\overline{W} = \text{SCOV}.\big(\bigcup_{i=0}^{m-n+1} \Sigma_I^i\big).W$, and $\overline{Wp} = Wp_1 \cup Wp_2$, where

$$Wp_1 = \text{SCOV}.\big(\bigcup_{i=0}^{m-n} \Sigma_I^i\big).W$$

and

$$Wp_2 = \text{SCOV}.\Sigma_I^{m-n+1} \oplus \{W_0, \ldots, W_{n-1}\}.$$

Obviously, $Wp_1 \subseteq \overline{W}$, and

$$\begin{array}{rcll} Wp_2 & = & \text{SCOV}.\Sigma_I^{m-n+1} \oplus \{W_0, \ldots, W_{n-1}\} & \\ & \subseteq & \text{SCOV}.\Sigma_I^{m-n+1} \oplus \{W\} & [W_i \subseteq W] \\ & = & \text{SCOV}.\Sigma_I^{m-n+1}.W & \\ & \subseteq & \overline{W} & \end{array}$$

Hence $\overline{Wp} = (Wp_1 \cup Wp_2) \subseteq \overline{W}$. □

4.8.2 Testing Nondeterministic FSMs for Reduction Using the State Counting Method

When testing nondeterministic FSMs, the conformance relation to be applied in most cases is reduction $\preceq$, because in most of these application scenarios the reference FSM is an *over approximation* of the intended behaviour. The implementation will then be deterministic or nondeterministic, but in any case it will realise less behaviours than "allowed" according to the reference model.

In the case of nondeterministic implementations, one input sequence may stimulate several legal output sequences. This suggests to use adaptive test cases as defined in Section 4.1, because it is often useful to adapt the test

behaviour to the actual reactions of the SUT observed during the test execution.

The complete testing assumption introduced above must hold just as for the case when testing nondeterministic FSMs for equivalence.

The most effective testing algorithms for nondeterministic FSMs and reduction as conformance relation have been suggested by [26, 55]. In this section, we present the simpler version of reduction testing with preset test cases which uses the *state counting* method.

Recall that we already have established a complete test suite for reduction testing in Theorem 4.4. There, the resulting number of test cases to be executed was $O(|\Sigma|^{mn})$ (see formula (4.2)). As a consequence, any new complete reduction testing method should require less than that amount of test cases in order to be of interest for us.

The state counting method exploits facts about *some* states being reachable in a deterministic way, though the FSM reference model is nondeterministic. This motivates the following definitions.

Definition 4.8 *Let* $M = (Q, \underline{q}, \Sigma_I, \Sigma_O, h)$ *be a completely specified, minimal and observable FSM. A state* q *is* d-reachable *("deterministically reachable") if and only if there is some input sequence* $\overline{x}$ *such that*

$$q = \underline{q}\text{-}\underline{\text{after}}\text{-}\overline{x}/\overline{y} \text{ for all } \overline{y} \text{ satisfying } \overline{x}/\overline{y} \in L(M).$$

The subset of d-reachable states of M *is denoted by* $Q^d \subseteq Q$.

Note that the initial state of M is always d-reachble, so Q^d is never empty.

Definition 4.9 *A set* V *of input sequences is called a* d-state cover *if and only if for any d-reachable state* q*, there exists an input trace* $\overline{x} \in V$*, such that* $\underline{q}\text{-}\underline{\text{after}}\text{-}\overline{x} = q$.

The following definition describes when two states q, q' of a nondeterministic FSM are "reliably" distinguishable by a single input x. This is the case when *all* possible outputs y to be observed when inputting x in state q differ from *all* possible outputs to be obtained when applying input x in state q'.

Definition 4.10 *Two states* q, q' *are* r(1)-distinguishable *if and only if there is some defined input* x *satisfying*

$$\forall y \in \Sigma_O : \big(x/y \in L(q) \Rightarrow x/y \notin L(q')\big) \wedge \big(x/y \in L(q') \Rightarrow x/y \notin L(q)\big).$$

The singleton set $\{x\}$ *is called an* r-distinguishing *set of* q, q'.

In some cases, we will use the equivalent defining formula

$$L(q) \cap L(q') \cap \{x/y \mid y \in \Sigma_O\} = \varnothing \tag{4.8}$$

for r(1)-distinguishable states.

Definition 4.11 *Two states* q, q' *are* $r(k)$-distinguishable *for* $k > 1$ *if and only if either* q, q' *are* $r(j)$*-distinguishable for some* $1 \leq j < k$ *or there is some defined input* x *satisfying*

$$q\text{-}\underline{\text{after}}\text{-}x/y \text{ and } q'\text{-}\underline{\text{after}}\text{-}x/y \text{ are } r(j)\text{-distinguishable for some } j < k$$

for all $y \in \Sigma_O$ *with* $x/y \in L(q) \cap L(q')$.

Definition 4.12 *Let* q, q' *be* $r(k)$*-distinguishable states. Let* $x \in \Sigma_I$*, such that for any* $x/y \in L(q) \cap L(q')$*,* q*-after-*x/y *and* q'*-after-*x/y *are* $r(j)$*-distinguishable, for some* $j < k$*. Then*

$$W(q, q') = \{x\}. \bigcup_{x/y \in L(q) \cap L(q')} W(q\text{-after-}x/y, q'\text{-after-}x/y)$$

is called an r-distinguishing set *of* q, q'.

Definition 4.13 *A set* W *of input sequences is called an* r-characterisation set *if* W *r-distinguishes any r-distinguishing states.*

Definition 4.14 *@todo Let* $R_1, \ldots, R_t$ *be maximal sets of* r*-distinguishable states.* $\bigcup_{i=1}^{t} R_i = Q$*. Denote* $R_i^d = R_i \cap Q^d$*, where* $Q^d = \{q \in Q \mid q$ *is d-reachable*$\}$*. Let* $q \in Q$ *be d-reachable. Let* $\alpha = \overline{x}/\overline{y} \in L(q)$ *of length* k*. Set* $q_i = q$*-after-*α_i*,* $\forall 1 \leq i \leq k$*, where* $\alpha_i = \alpha^{[1,\ldots,i]}$*. Define* $n_j(q, \alpha) = \sum_{i=1}^{k} |\{q_i\} \cap R_j|$*. The number of states of* R_j *that are traversed by* α *from state* q *is denoted by* $n_j(q, \alpha)$.

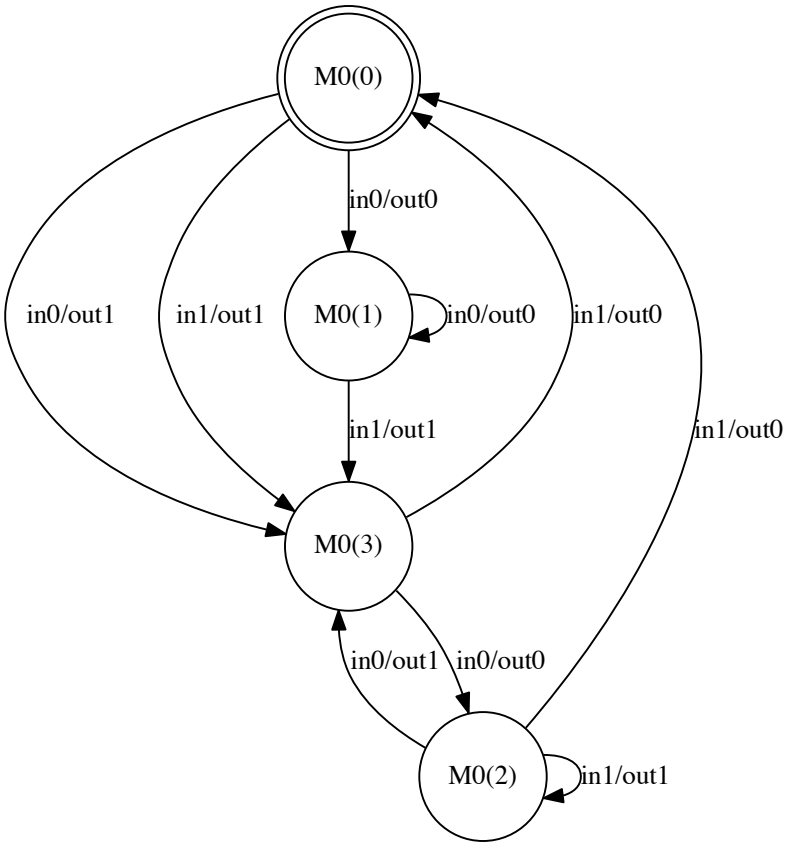


Figure 4.8: Nondeterministic FSM M_0 (example taken from [26]).

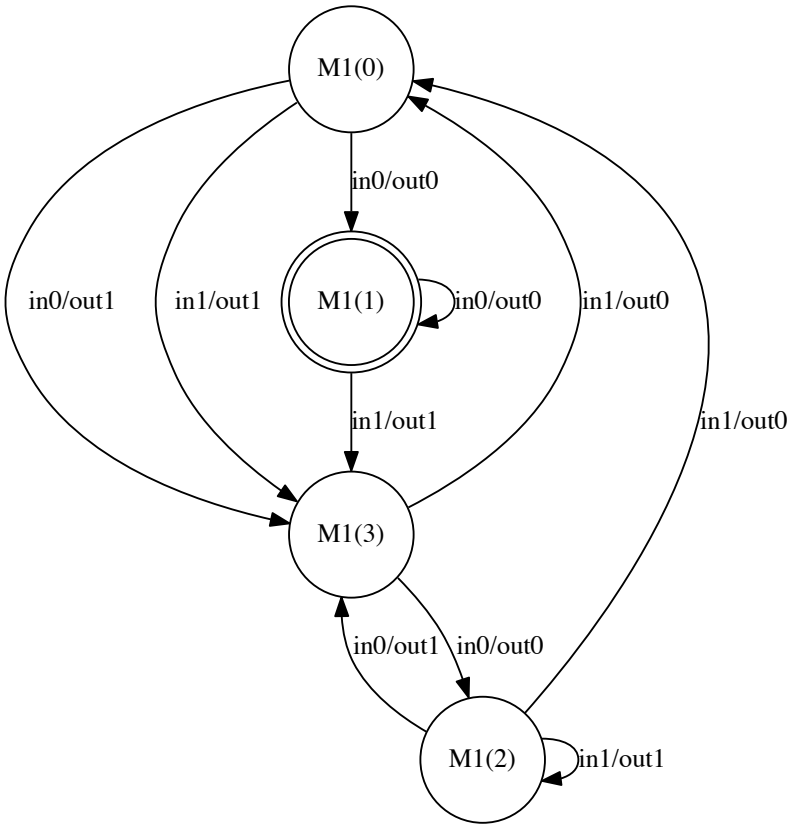


Figure 4.9: Nondeterministic FSM M_1 – identical to M_0, but with start state 1 instead of 0.

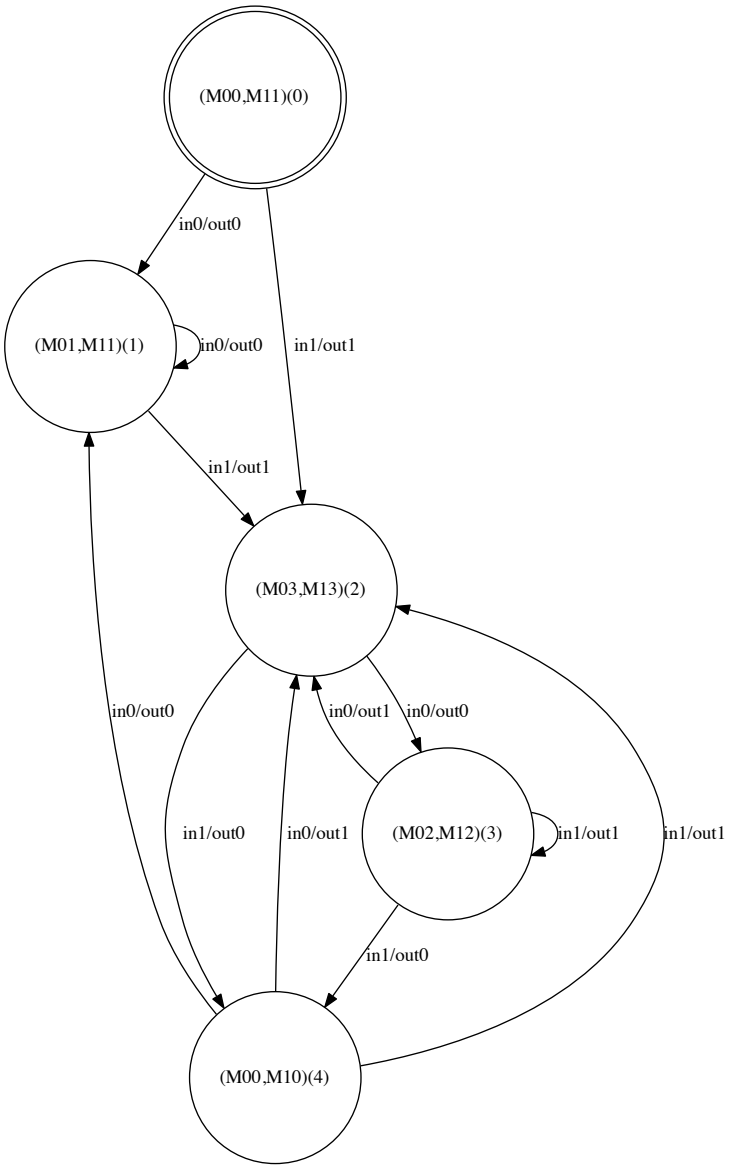


Figure 4.10: Intersection $M_0 \cap M_1$ contains a (in fact, is a) completely defined sub-machine. Therefore state 0 and 1 of M_0 are *not* r-distinguishable.

Traversal sets [58]

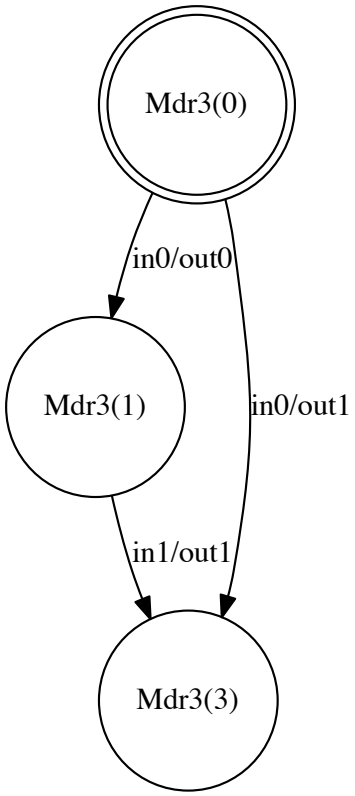


Figure 4.11: State 3 of M_0 is definitely reachable.

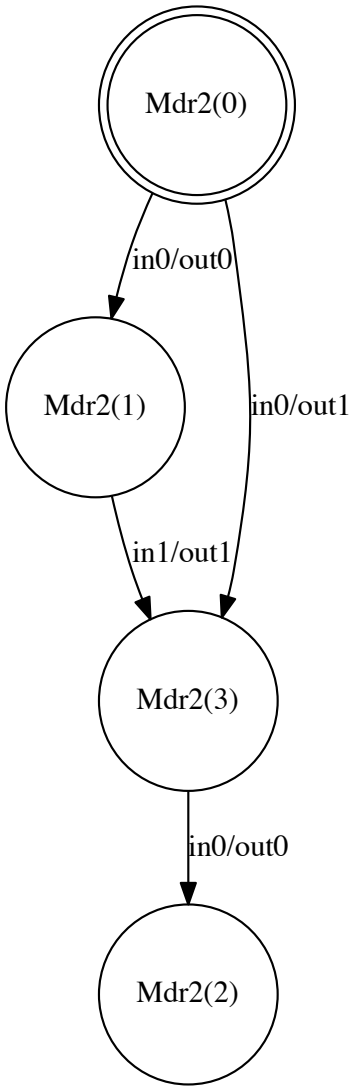


Figure 4.12: State 2 of M_0 is definitely reachable.

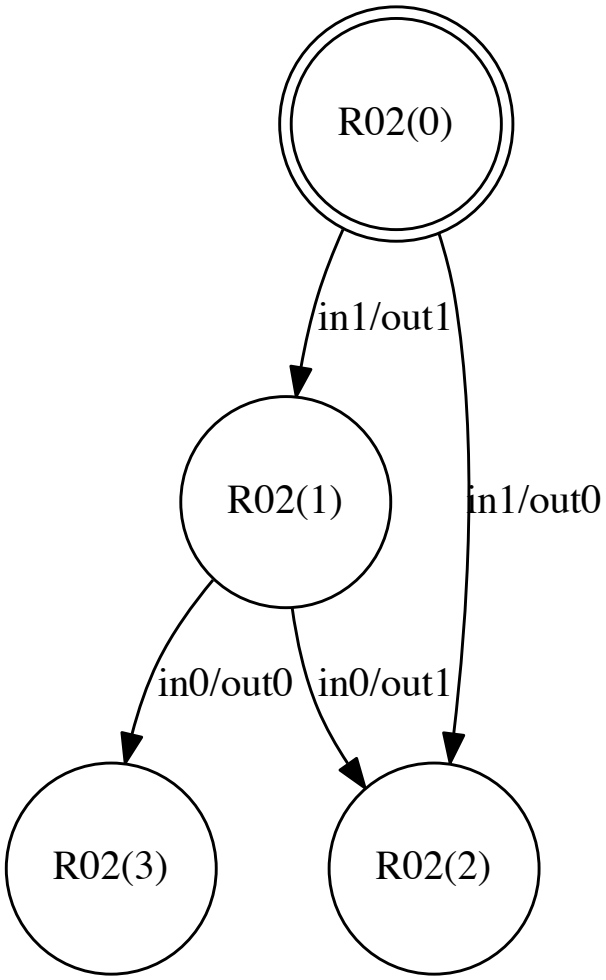


Figure 4.13: State separator for states 0 and 2 in M_0.

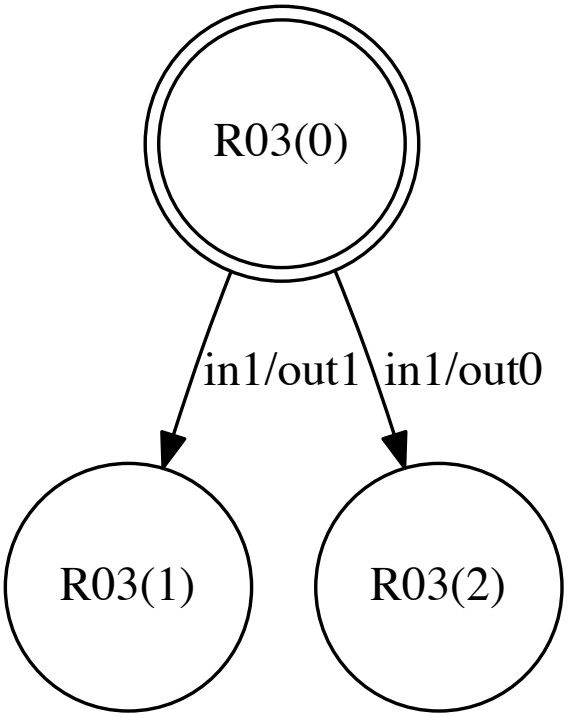


Figure 4.14: State separator for states 0 and 3 in M_0.

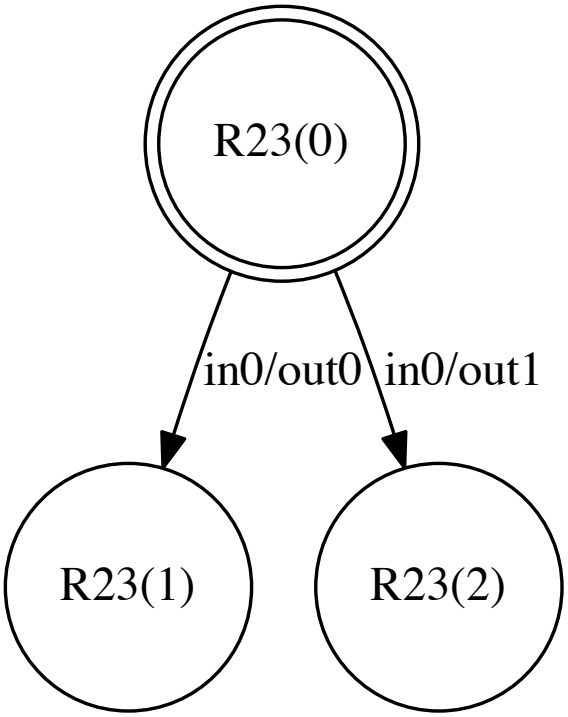


Figure 4.15: State separator for states 2 and 3 in M_0.

Part III

Equivalence Class Partition Testing

Chapter 5

Introduction to Equivalence Class Partition Testing

5.1 Objectives

Model-based testing against finite state machine models is limited by the fact that FSMs only admit finite input data domains, finite output data domains, and finitely many internal states. When dealing with test models possessing infinite data domains – for example, real-valued time-continuous observables in physical models – or very large data domains that cannot be enumerated with acceptable effort, new methods are required to reduce the potentially infinite number of test cases needed to achieve completeness or at least acceptable test strength for the test strategies to be generated.

A well-known heuristic for this objective is *equivalence class partition testing (ECPT)*. Infinite (input, output or internal state) data domains are partitioned into finitely many domains, and it is argued that the implementation is likely to "behave equivalently" within each partition. This means, that all data elements from the same partition will be processed by the same algorithm. Therefore it is likely that an error in this algorithm will be detected when choosing one or just a few representatives from this class. The research question investigated in the subsequent chapters is whether this heuristic can be formalised to yield complete testing theories for models and SUTs with infinite (or very large) data domains.

Example 6. Suppose

```
float f(float x)
```

is a stateless operation supposed to implement the transformation of input x according to the linear equation $f(x) = a \cdot x + b$ with $a \neq 0$. Consider the fault domain of all function implementations f' that are linear, that is, all members of this domain implement some linear equation $f'(x) = a' \cdot x + b'$. Then the whole float-data domain represents a single equivalence class, and two distinct representatives are sufficient to uncover every error in an implementation from this fault domain. For example, the representatives can be $x_0 = 0$ and $x_1 = 1$. Applying x_0 determines the parameter b' used by implementation f', and consecutive application of x_1 determines the factor a', since

$$a' = \frac{f'(x_1) - b'}{x_1}$$

for all $x_1 \neq 0$. This concept can be generalised to more complex transformations than just linear functions of one argument.

1. A polynomial transformation

   $$f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

   is fully determined by $(n + 1)$ test cases.

2. This can be generalised to functions

   $$f = \mathbb{R}^n \to \mathbb{R}^m, \quad f(x_1, \ldots, x_n) = (f_1(x_1, \ldots, x_n), \ldots, f_m(x_1, \ldots, x_n)),$$

   where each $f_i(x_1, \ldots, x_n)$ is a polynomial of degree k with n arguments $(x_1, \ldots, x_n)$.

□

5.2 Three Types of Equivalence Classes

Structural equivalence classes. When classes are derived from the control structures of a specification model, or – in the case of structural software tests – from the control structures of the source code, they are called *structural equivalence classes* [16, B.19]. Model or SUT execution traces covering the same sequence of classes are considered as equivalent, because they are subject to the same sequences of control decisions. It is argued that only a few of these equivalent traces are needed to uncover errors in data transformations or control decisions along the same sequence of classes.

Input equivalence classes. When input data domains are partitioned into subdomains, where equivalent SUT behaviour is expected, the resulting classes are denoted as *input equivalence classes* [64, p. 101]. Whereas structural equivalence classes require a model (or code) revealing control decisions to be performed by the SUT, input classes can be derived from implicit specifications of the required behaviour, such as pre/post-conditions in the case of terminating SUT or temporal logic or trace logic specifications in the case of – possibly non-terminating – reactive systems.

Output equivalence classes. If classes are derived from output domains whose values are assumed to be processed in an equivalent way, they are called *output equivalence classes* [64, p. 103]. The input data is now constructed by determining sufficiently many input data, so that all output domains are covered by the resulting tests.

5.3 Formal Background

The theoretical foundations for dealing with the reduction of test suites of unmanageable size have been laid out in [20]: the justification of equivalence classes is just a special case of proving the validity of a *uniformity hypothesis*. Given an SUT $\mathcal{S}'$, a subdomain D of its semantic domain, and a specification Φ with free variable in D, this hypothesis states

$$\forall \iota \in \mathsf{D} : \big(\mathcal{S}' \models \Phi(\iota) \Rightarrow (\forall \tau \in \mathsf{D} : \mathcal{S}' \models \Phi(\tau))\big) \tag{5.1}$$

This means that it can be assumed that the SUT will satisfy specification Φ for *every* member of D if $\mathcal{S}'$ fulfils Φ for *one* $\iota \in \mathsf{D}$. This formalises the informal notion of "similar treatment" or "equivalent behaviour" for the elements of an equivalence class partition.

In the case of non-terminating systems (these are in the main focus of this article), the reduction of infinite data domains to finitely many partitions does not immediately induce finite test cases, since the SUT will generally be able to produce an infinite number of observation traces of unbounded length. As outlined in [20], a reduction to a complete finite test suite consisting of *finite* test cases depends on the validity of a *regularity hypothesis*. This is of the form

$$\big(\forall \mathsf{t} \in \mathsf{T} : |\mathsf{t}| \leq \mathsf{k} \Rightarrow \mathcal{S}' \models \Phi(\mathsf{t})\big) \Rightarrow \big(\forall \mathsf{t} \in \mathsf{T} : \mathcal{S}' \models \Phi(\mathsf{t})\big) \tag{5.2}$$

where T is the (potentially infinite) set of tests t and may include tests of infinite length. Then $|\mathsf{t}|$ is some $\mathbb{N}$-valued function assessing the "size" of a test, such as, for example, the length of the trace of inputs to be passed successively to the SUT. The regularity hypothesis states that if all tests of some finite maximal size k have been passed by the SUT P, then P will in fact pass *all* possible tests from T.

5.4 Main Results

In the subsequent chapters, a subdomain of Kripke structures called *reactive I/O-state-transition systems (RIOSTS)* is investigated. Its members have large (possibly infinite) input variable domains and finite domains for internal state variables and outputs. Typical examples for these systems are train speed supervision systems (reference speed and actual speed have large domains, while outputs are ON/OFF triggers for the emergency brakes), airbag controllers (crash sensors have large analogue input ranges, but outputs are Boolean activation triggers for the inflation of the airbags), or route controllers for train interlocking systems (possible train positions in the railway network represent large input domains, while outputs are discrete commands for a limited number of points and signals). These examples show that the domain of RIOSTS covers a large variety of embedded control systems: it is one of the important paradigms of embedded systems design to "condense" a conceptually infinite amount of input information to a small set of control decisions.

In this context, the following main results about *input equivalence class partition testing (IECPT)* are elaborated.

- It is shown that complete black-box testing theories for (deterministic or nondeterministic) FSMs can be translated to likewise complete black-box input equivalence class partition testing theories for RIOSTS. To this end, input and state equivalence relations are defined, allowing to transform RIOSTSs to FSMs without any loss or addition of information about the observable behaviour.

- The complexity of RIOSTS test suites translated from complete FSM suites is the same as that of the FSM suites. In particular, *finite* FSM suites result, when translated, in finite RIOSTS suites.

- The application of this translation method results in several IECPT theories for nondeterministic RIOSTS that were, to our best knowledge, not known before. As conformance relations, I/O-equivalence (reference model and SUT can perform exactly the same I/O-traces) and reduction (the SUT performs a subset of the I/O-traces of the reference model) can be used. Previous results in this field obtained by the authors were only applicable to deterministic systems [31], where I/O-equivalence is the only useful conformance relation.

- It is proven that the completeness results are preserved when performing random selections from input equivalence class X whenever a representative of X is needed. This extension of the strategies (normally, fixed representatives would be chosen from each class) has been shown to yield considerable test strength improvements for SUTs *outside* the fault domain. The experimental evaluation supporting this claim has been performed for airbag systems, train speed supervision monitors, and interlocking systems and has been described in [34, 52].

- While known approaches to equivalence class testing usually focused on specific modelling formalisms (this is described in more detail in Section 11 on related work), the results presented here are applicable to *any* concrete formalism whose behavioural semantics can be expressed by means of RIOSTS. This covers, for example, SysML (for details, see [31]), and extended state machines (see Section 11).

5.5 Proof Strategy and Overview

The material on equivalence class partition testing presented here follows [30, 31, 4, 34, 32]. We focus on the investigation of input equivalence class partitions inducing complete testing theories.

Complete black-box testing theories accept correct implementations and reject faulty ones. They depend on fault models specifying the reference model and the conformance relation to be tested against. Moreover, fault models comprise fault domains capturing the types of (faulty or correct) behaviours to be expected in an implementation (Section 2.3).

The proof strategy for obtaining our main results is as follows. Testing theories are typically elaborated in a specific semantic domain or a subset thereof, such as the domain of finite state machines, labelled transition

systems, or Kripke structures. To translate a theory from one domain to another, we create two maps: the model map transfers concrete models of one domain into the other, and the test case map translates test cases in the opposite direction. If this pair of maps fulfils a so-called satisfaction condition, one can conclude that every complete testing theory elaborated in the co-domain of the model map gives rise to an equally complete theory in the source domain of this map. This general approach is described in Section 2.6 and expressed by Theorem 2.1. Chapter 11 presents references to the literature where the associated model-theoretic underpinning is elaborated in more detail.

In Chapter 6, the concrete semantic domain to be investigated for the purpose of equivalence class testing is introduced: reactive I/O-transition systems (RIOSTSs) are defined as a subdomain of Kripke structures. They operate on input, output, and internal state variables and distinguish quiescent (i.e. stable) states from transient ones. For comparing RIOSTSs, I/O-equivalence (both systems perform the same input/output traces) and reduction (the system representing the implementation performs a subset of the I/O-traces of the reference system) are considered as conformance relations.

In analogy to the test cases introduced for FSMs in Section 4.1, we define abstract test cases for RIOSTSs as sets of pass traces and fail traces: an RIOSTS passes such a test, if and only if its language contains all pass traces and none of the fail traces specified for the test.

RIOSTSs allow for variables with arbitrary types; we focus here on the RIOSTS subdomain with input variable types of arbitrary (possibly infinite) size, while internal state variables and outputs are restricted to finite types that can be enumerated for the purpose of test data computation. It is the objective of this article to elaborate various input equivalence class partition test strategies for this semantic subdomain.

In Chapter 3 the semantic domain from where existing complete testing theories will be transferred to the domain of RIOSTSs has already been introduced: this domain consists of finite state machines which may be deterministic or nondeterministic. As conformance relations language equality (also called I/O-equivalence) or language inclusion (also called reduction) were considered. We described the existing definitions of FSM parallel composition (which equals language intersection), as well as the notions of minimality and observability.

The model map from the RIOSTS subdomain under consideration into the domain of FSMs is constructed in Chapter 7. We show that these

RIOSTSs can be abstracted by creating pairs of state and input equivalence class partitionings, so that the future behaviour of an RIOSTS residing in a state class is independent on the class representative and only depends on the sequence of input classes, but not on the input class representatives that are present in an input trace. This allows us to abstract each member of the RIOSTS subdomain to a uniquely determined minimal and observable FSM. This FSM operates on input equivalence class identifiers as input alphabet, and its output alphabet is in on-to-one correspondence to the different output vector valuations that can be be produces by the associated RIOSTS. It is shown that the model map induced by this abstraction respects the conformance relations, which is the first requirement of the satisfaction condition (Theorem 7.3).

In Chapter 8 the test case map is constructed. An abstract FSM test case is mapped to an RIOSTS test case by mapping each pass trace and fail trace of the former to a pass or fail trace of the latter. When mapping such an FSM I/O-trace, each input referring to an input equivalence class partition is replaced to a randomly selected input value residing in this partition. Each FSM output is mapped to the corresponding RIOSTS output vector valuation.

With model map T from RIOSTSs to FSMs and test case map T^* from FSM test cases to RIOSTS test cases at hand, the second requirement of the associated satisfaction condition is proven in Chapter 9: for each RIOSTS model $\mathcal{S}$ in the domain of the model map it is shown that the associated FSM $\mathsf{T}(\mathcal{S})$ passes a test case U if and only if $\mathcal{S}$ passes the translated test case $\mathsf{T}^*(\mathsf{U})$.

In Chapter 10 the general theory translation Theorem 2.1 is applied to the concrete model map T from the RIOSTS subdomain to FSMs and its counter part, the test case map T^*. This allows us to translate arbitrary complete FSM testing theories applicable to FSMs in the co-domain of T to complete IECPT theories of the RIOSTS subdomain. As a result, complete IECPT theories for RIOSTSs are obtained for (1) deterministic reference models and deterministic implementations (this is a known result, but proven here again in the more general theory translation framework), (2) nondeterministic models (different theories are available for I/O-equivalence and reduction), and (3) nondeterministic reference models and deterministic implementations (reduction only). As an example of a weaker testing strategy it is shown that (4) the T-Method which is known to uncover FSM output faults gives rise to an IECPT theory uncovering RIOSTS output faults. The IECPT strategies

(2), (3), and (4) are new for the RIOSTS subdomain under consideration. Since the test case map leaves a degree of freedom regarding the selection of representatives from each input equivalence class, all of these strategies can be randomised with respect to this selection.

Chapter 11 presents detailed references to related work.

Chapter 6

State Transition Systems and Kripke Structures

State Transition Systems The testing theory to be elaborated in the subsequent chapters applies to all modelling formalisms whose behavioural semantics can be expressed as a variant of *state transition systems (STS)*; these are triples $\mathcal{S} = (S, \underline{s}, R)$ with state space S, initial state $\underline{s} \in S$ and transition relation $R \subseteq S \times S$. A finite sequence of states $s_1 \dots s_n$ is called a *trace* of $\mathcal{S}$, if it starts in the initial state (i.e. $s_1 = \underline{s}$) and if each pair of consecutive states is related by the transition relation, that is, $\forall i \in \{1, \dots, n-1\} : R(s_i, s_{i+1})$. The set of all traces of $\mathcal{S}$ is denoted by traces($\mathcal{S}$). A *deadlock state* of $\mathcal{S}$ is a state $s \in S$ from where no emanating transitions exist, that is, $(s, s') \notin R$ for all $s' \in S$. A *termination trace* of $\mathcal{S}$ is a traces that ends in a deadlock state. We denote the subset of states that do not deadlock by $\mathrm{DF}(S)$.

Input/Output State Transition Systems We restrict the STSs under consideration to those possessing a notion of variable valuations, input, and output. An *input/output state transition system (IOSTS)* is an STS $\mathcal{S} = (S, \underline{s}, R)$ where states $s \in S$ are valuation functions $s : V \to D$, such that V is a set of variable symbols and $D = \bigcup_{v \in V} D_v$, where D_v is the domain of variable v, and $s(v) \in D_v$ holds for every $v \in V$ and $s \in S$. The variable symbol set V is finite and can be partitioned into disjoint sets $V = I \cup M \cup O$ called input variables, (internal) model variables, and output variables, respectively. The state space S of an IOSTS can be partitioned into *quiescent* and *transient* states, $S = S_Q \cup S_T$. All transitions from quiescent states change inputs only, while outputs and internal state variables remain un-

changed. Conversely, all transitions from transient states change internal states and outputs only. Transitions from quiescent to transient states can only be performed by *changing* the input valuation. This is expressed by the condition $\forall s \in S_Q, s' \in S_T : R(s, s') \Rightarrow (\exists x \in I : s(x) \neq s'(x))$ which restricts the admissible transition relations of IOSTS. This reflects the fact that writing the current valuation again to a shared variable interface is not observable. We require, however, that *stuttering* is admissible which means that for quiescent states s that do not deadlock, (s, s) is a member of the transition relation: $\forall s \in S_Q \cap DF(S) : R(s, s)$.

Kripke Structures By associating a set AP of atomic propositions with free variables in V, any IOSTS can be extended to a Kripke Structure $K(\mathcal{S}) = (S, \underline{s}, R, V, D, L, AP)$. The labelling function $L : S \to 2^{AP}$ maps $s \in S$ to the set of all atomic propositions $p \in AP$ that evaluate to `true`, when replacing every free variable v of p by its valuation $s(v)$ in state s. Observe, however, that the standard definition (see, e.g. [10]) of Kripke Structures requires the transition relation to be total, so that deadlock states are not admissible. In the context of testing, the notion of deadlock is required in order to express, for example, termination properties of test cases.

Notation For IOSTS $\mathcal{S} = (S, \underline{s}, R)$, the following notational conventions are used. The set of input variables is denoted by $I = \{x_1, \ldots, x_k\}$, and the tuple of all inputs (usually called the *input vector*) is denoted by $\boldsymbol{x} = (x_1, \ldots, x_k)$. The valuation of the input vector in a given state $s \in S$ is written as $s(\boldsymbol{x}) = (s(x_1), \ldots, s(x_k))$. $D_I = D_{x_1} \times \cdots \times D_{x_k}$ denotes the set of all input vector valuations. Finite sequences of such valuations are written as $\overline{\boldsymbol{x}} \in D_I^*$. In analogy, output variables and (sequences of) output vectors are defined: $O = \{y_1, \ldots, y_p\}$, $\boldsymbol{y} = (y_1, \ldots, y_p)$, $s(\boldsymbol{y}) = (s(y_1), \ldots, s(y_p))$, $\overline{\boldsymbol{y}} \in D_O^*$. Changing a state $s_1 : V \to D$ with respect to its input vector valuation is written as $s_2 = s_1 \oplus \{\boldsymbol{x} \mapsto \boldsymbol{c}\}, \boldsymbol{c} \in D_I$. State s_2 coincides with s_1 for all variables $v \in V - I$, but $s_2(\boldsymbol{x}) = \boldsymbol{c}$.

The tuple $\mathtt{Sig} = (I, O, D_I, D_O)$ represents an interface type specification; it is called the *signature* of $\mathcal{S}$.

For any quiescent state $s \in S_Q$, input vector $\boldsymbol{c} \in D_I$ *is defined in* s if and only if $(s, s \oplus \{\boldsymbol{x} \mapsto \boldsymbol{c}\}) \in R$. The set of these admissible input changes is denoted by

$$\mathbf{C}(s) = \{\boldsymbol{c} \in D_I \mid R(s, s \oplus \{\boldsymbol{x} \mapsto \boldsymbol{c}\})\}$$

An IOSTS $\mathcal{S}$ over variables from $V = I \cup M \cup O$ is *completely specified* if and only if from every quiescent state, every input change is possible, that is, $(\forall s \in S_Q : \mathbf{C}(s) = D_I)$.

Nondeterminism An IOSTS $\mathcal{S}$ is called *deterministic*, if every transient state has exactly one post-state; otherwise it is called *nondeterministic*. Following the terminology in [62], $\mathcal{S}$ exhibits *bounded nondeterminism*, if transient states can branch into finitely many post-states only; otherwise $\mathcal{S}$ has *unbounded nondeterminism*.

Livelock Freedom and the /-Operator An IOSTS $\mathcal{S}$ is called *livelock-free* if and only if there exists no infinite sequence of consecutive transient states linked by the transition relation. Livelock-free IOSTSs allow for an abstraction of state traces by removing all transient states from this trace. The resulting sequences of quiescent states are called *q-traces*. We will now introduce the /-operator for constructing all q-traces emanating from a quiescent state $s \in S_Q$. The construction is performed by "applying" nonempty input traces $\overline{x}$ to s. Since the IOSTS may be nondeterministic, there may be more than one q-trace associated with the same input trace. Therefore the operator application $s/\overline{x}$ is typed as a set of q-traces. More formally, it is defined as follows.

$$s_0/\mathbf{c} = \begin{cases} \varnothing & \text{if } \mathbf{c} \notin \mathbf{C}(s_0) \\ \{s_0 \oplus \{x \mapsto \mathbf{c}\}\} & \text{if } \mathbf{c} \in \mathbf{C}(s_0) \\ & \text{and } s_0 \oplus \{x \mapsto \mathbf{c}\} \in S_Q \\ \{s' \in S_Q \mid \exists k \geq 1, s_1, \ldots, s_k \in S_T : & \\ \qquad s_1 = s_0 \oplus \{x \mapsto \mathbf{c}\} \wedge & \text{if } \mathbf{c} \in \mathbf{C}(s_0) \\ \qquad \forall i \in \{0, \ldots, k-1\} : & \text{and } s_0 \oplus \{x \mapsto \mathbf{c}\} \in S_T \\ \qquad\qquad R(s_i, s_{i+1}) \wedge R(s_k, s')\} & \end{cases}$$

An input trace $\overline{x} = \mathbf{c}_1 \ldots \mathbf{c}_k$ *is defined in state* $s_0 \in S_Q$, if and only if there are $s_1, \ldots, s_k \in S_Q$ such that $\mathbf{c}_{i+1} \in \mathbf{C}(s_i)$ and $s_{i+1} \in s_i/\mathbf{c}_{i+1}$ for $i = 0, \ldots, k-1$. We extend the /-operator "$s_0/\_$" recursively to input traces that are defined in s_0.

$$s_0/\varepsilon = \{\varepsilon\}$$

$$s_0/\mathbf{c}_1 \ldots \mathbf{c}_k = \{s_1 \ldots s_k \mid s_{i+1} \in s_i/\mathbf{c}_{i+1}, i = 0, \ldots, k-1\}$$

Obviously, $s_0/\overline{x} = \varnothing$ if input trace $\overline{x}$ is not defined in s_0.

If $\overline{s} = s_1 \dots s_n \in s/\overline{x}$, then $\overline{y} = \overline{s}(y) = s_1(y) \dots s_n(y)$ denotes the *output trace* created during application of input trace $\overline{x}$ to s. Define $(s/\overline{x})(y) = \{\overline{s}(y) \mid \overline{s} \in s/\overline{x}\}$. Abstracting $\overline{s}$ to inputs and outputs only, results in the *I/O-trace* $\overline{x}/\overline{y} = \overline{s}(x, y) = s_1(x, y) \dots s_n(x, y)$. The set of all I/O-traces starting in some quiescent state s is specified by $L(s) = \{\varepsilon\} \cup \{\overline{x}/\overline{y} \mid \overline{x} \in D_I^* \wedge \exists \overline{s} \in (s/\overline{x}) : \overline{s}(y) = \overline{y}\}$. The set $L(\mathcal{S})$ of I/O-traces associated with $\mathcal{S}$ is identical to $L(\underline{s})$, the set of I/O-traces associated with the initial state. For any $s \in S_Q, \overline{x}/\overline{y} \in L(s)$, define $s\text{-}\underline{\text{after}}\text{-}\overline{x}/\overline{y} = \{\text{last}(\overline{s}) \mid \overline{s} \in s/\overline{x} \wedge \overline{s}(y) = \overline{y}\}$. When considering input traces only, the definition $s\text{-}\underline{\text{after}}\text{-}\overline{x} = \{\text{last}(\overline{s}) \mid \overline{s} \in s/\overline{x}\}$ applies.

Reactive I/O-Transition Systems A *reactive I/O transition system (RIOSTS)* $\mathcal{S} = (S, \underline{s}, R)$ is an IOSTS, such that every transient state has quiescent post-states only. The intuition behind this definition is that sequences of transitions between transient states are typically not observable in test executions, at least when performing HW/SW integration tests. Therefore these sequences can be aggregated to a single transition from a transient state to a quiescent post-state. This aggregation can always be performed if the RIOSTS is deterministic and free of livelocks; an algorithm to perform this is given in [31, Section 3]. In the nondeterministic case it has to be additionally required that nondeterminism is bounded. This condition is trivially fulfilled in the context of this paper, because we will specialise on RIOSTS with possibly infinite input domains, but with finite domains for internal states and outputs.

Example 7. We specify an alarm indication system as an RIOSTS $\mathcal{S}$ which inputs non-negative real values on variable x and raises an alarm on output y if the values of x violate some threshold $max > 0$. For $x = max$ a nondeterministic decision whether to raise the alarm is admissible. When the value is back in normal range, the alarm shall be switched off. To avoid unstable `ALARM, OK`-outputs in situations where the values of x are close to max and rapidly changing, an active alarm shall only be de-activated after x has dropped below $max - \delta$ for some $0 < \delta < max$.

$\mathcal{S}$ has variables $V = I \cup M \cup O = \{x\} \cup \{m\} \cup \{y\}$ with domains $D_x = [0, \infty)$, $D_m = \{0, 1, 2\}$, and $D_y = \{\texttt{OK}, \texttt{ALARM}\}$. The initial state $\underline{s}$ is specified by $\underline{s}(x, m, y) = (0, 0, \texttt{OK})$ (recall that we use vector notation for function arguments: $\underline{s}(x, m, y) = (0, 0, \texttt{OK})$ is short for $(\underline{s}(x), \underline{s}(m), \underline{s}(y)) = (0, 0, \texttt{OK})$).

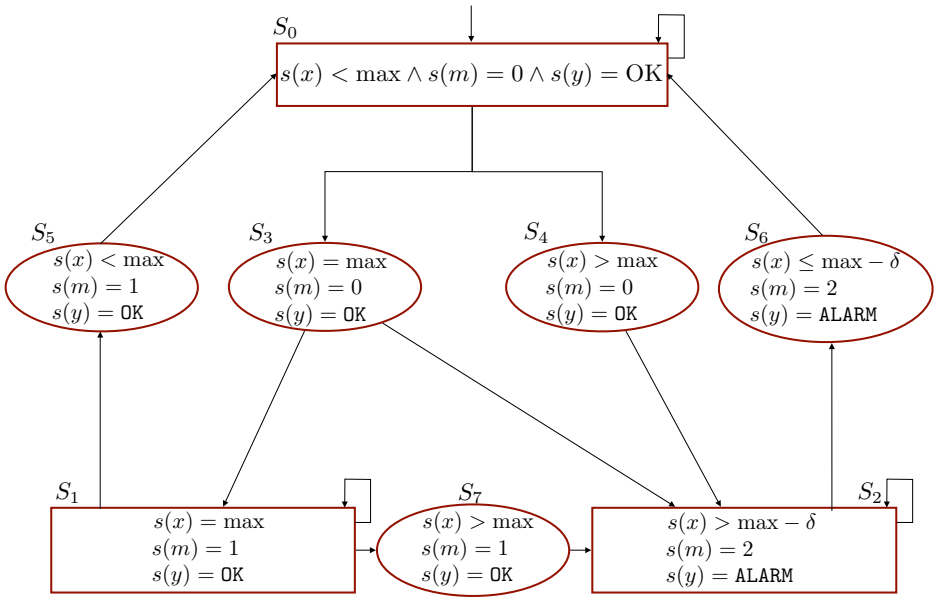


Figure 6.1: Transition diagram of RIOSTS $\mathcal{S}$ from Example 1. Boxes specify sets of quiescent states, ovals transient states.

The sets of quiescent and transient states, as well as the transition relation are visualised in Fig. 6.1. More formally, the quiescent states of $\mathcal{S}$ are specified by

$$
\begin{aligned}
\mathsf{S_Q} &= \mathsf{S_0} \cup \mathsf{S_1} \cup \mathsf{S_2} \\
\mathsf{S_0} &= \{\mathsf{s} : \mathsf{V} \to \mathsf{D} \mid \mathsf{s}(\mathsf{x}) < \max \wedge \mathsf{s}(\mathsf{m}) = 0 \wedge \mathsf{s}(\mathsf{y}) = \mathtt{OK}\} \\
\mathsf{S_1} &= \{\mathsf{s} : \mathsf{V} \to \mathsf{D} \mid \mathsf{s}(\mathsf{x}) = \max \wedge \mathsf{s}(\mathsf{m}) = 1 \wedge \mathsf{s}(\mathsf{y}) = \mathtt{OK}\} \\
\mathsf{S_2} &= \{\mathsf{s} : \mathsf{V} \to \mathsf{D} \mid \mathsf{s}(\mathsf{x}) > \max - \delta \wedge \mathsf{s}(\mathsf{m}) = 2 \wedge \mathsf{s}(\mathsf{y}) = \mathtt{ALARM}\}
\end{aligned}
$$

The transient states are defined by

$$
\begin{aligned}
S_T &= S_3 \cup S_4 \cup S_5 \cup S_6 \\
S_3 &= \{s : V \to D \mid s(x) = \max \wedge s(m) = 0 \wedge s(y) = \texttt{OK}\} \\
S_4 &= \{s : V \to D \mid s(x) > \max \wedge s(m) = 0 \wedge s(y) = \texttt{OK}\} \\
S_5 &= \{s : V \to D \mid s(x) < \max \wedge s(m) = 1 \wedge s(y) = \texttt{OK}\} \\
S_6 &= \{s : V \to D \mid s(x) \leq \max - \delta \wedge s(m) = 2 \wedge s(y) = \texttt{ALARM}\} \\
S_7 &= \{s : V \to D \mid s(x) > \max \wedge s(m) = 1 \wedge s(y) = \texttt{OK}\}
\end{aligned}
$$

The transition relation is defined by

$$
\begin{aligned}
R &= \bigcup_{i=0}^{2}(S_i \times S_i) \cup (S_0 \times S_3) \cup (S_0 \times S_4) \cup \\
&\qquad (S_1 \times S_5) \cup (S_1 \times S_7) \cup (S_2 \times S_6) \cup R_3 \cup R_4 \cup R_5 \cup R_6 \cup R_7 \\
R_3 &= \{(s, s') \in S_3 \times (S_1 \cup S_2) \mid s'(x) = s(x)\} \\
R_4 &= \{(s, s') \in S_4 \times S_2 \mid s'(x) = s(x)\} \\
R_5 &= \{(s, s') \in S_5 \times S_0 \mid s'(x) = s(x)\} \\
R_6 &= \{(s, s') \in S_6 \times S_0 \mid s'(x) = s(x)\} \\
R_7 &= \{(s, s') \in S_7 \times S_2 \mid s'(x) = s(x)\}
\end{aligned}
$$

□

Conformance Relations We are interested in two types of conformance relations between RIOSTS representing reference models and systems under test.

- *I/O-equivalence* expresses the fact that reference model and SUT can perform the same set of observable input/output sequences.
- *Reduction* expresses the property that the observable SUT behaviours form a subset of the behaviours observable in the reference model.

Let $\mathcal{S}$ be an RIOSTS. Two quiescent states $s_1, s_2 \in S_Q$ are called *I/O-equivalent* (written $s_1 \sim s_2$), if and only if $L(s_2) = L(s_1)$. State s_2 is called a *reduction* of s_1 (written $s_2 \preccurlyeq s_1$), if and only if $L(s_2) \subseteq L(s_1)$. Two RIOSTS $\mathcal{S}_1$ and $\mathcal{S}_2$ are called *I/O-equivalent* (written $\mathcal{S}_2 \sim \mathcal{S}_1$), if and only if $L(\mathcal{S}_2) = L(\mathcal{S}_1)$. $\mathcal{S}_2$ is called a *reduction* of $\mathcal{S}_1$ (written $\mathcal{S}_2 \preccurlyeq \mathcal{S}_1$), if and only

if $\mathsf{L}(\mathcal{S}_2) \subseteq \mathsf{L}(\mathcal{S}_1)$. It is easy to see that $\mathcal{S}_2 \preccurlyeq \mathcal{S}_1$ implies $\mathcal{S}_2 \sim \mathcal{S}_1$, if both have the same defined input sequences and $\mathcal{S}_1$ is deterministic. In particular, if $\mathcal{S}_1$ is deterministic and $\mathcal{S}_2$ is completely specified, $\mathcal{S}_1$ is also completely specified and $\mathcal{S}_2 \preccurlyeq \mathcal{S}_1$ implies $\mathcal{S}_2 \sim \mathcal{S}_1$.

Chapter 7

The Model Map

In general, an RIOSTS cannot be mapped to an FSM without losing significant behaviour-related information, because RIOSTS operate on variables with potentially infinite domains, while FSM are restricted to finite state spaces and finite input/output alphabets. In this section, a sub-domain of completely defined RIOSTS is identified, whose members $\mathcal{S}$ can be mapped to finite state machines M, such that the latter reflect the behaviour of the former in a "loss-less" way. This is achieved by way of input domain partitions. The domains of internal state variables and outputs are supposed to be finite for the RIOSTS sub-class under consideration.

7.1 Set Partitions

Recall that a *partition* of some set N is a subset $\mathcal{I}$ of N's power set, such that

$$\forall \mathsf{X} \in \mathcal{I} : \mathsf{X} \neq \varnothing, \quad \forall \mathsf{X}, \mathsf{X}' \in \mathcal{I} : \mathsf{X} = \mathsf{X}' \vee \mathsf{X} \cap \mathsf{X}' = \varnothing, \quad \text{and} \quad \bigcup_{\mathsf{X} \in \mathcal{I}} \mathsf{X} = \mathsf{N}$$

If $\mathcal{I}, \mathcal{I}'$ are two partitions of N, $\mathcal{I}'$ is called a *refinement* of $\mathcal{I}$ if and only if

$$\forall \mathsf{X}' \in \mathcal{I}' : \exists \mathsf{X} \in \mathcal{I} : \mathsf{X}' \subseteq \mathsf{X}$$

Given two partitions $\mathcal{I}, \mathcal{I}'$ of N, the *intersection*

$$\mathcal{I} \cap \mathcal{I}' = \{\mathsf{X} \cap \mathsf{X}' \neq \varnothing \mid \mathsf{X} \in \mathcal{I}, \mathsf{X}' \in \mathcal{I}'\}$$

is the coarsest partition of N that refines both $\mathcal{I}$ and $\mathcal{I}'$.

A partition $\mathcal{I}$ of N induces an equivalence relation $\sim_{\mathcal{I}}$ on N by defining $a \sim_{\mathcal{I}} a'$ if and only if a, a' reside in the same partition element $X \in \mathcal{I}$. Conversely, the classes of any equivalence relation on N build a partition of N. Partitions $\mathcal{I}$ of N induce equivalence relations on the set N^* of finite sequences over N-elements by defining

$$a_1 \dots a_k \sim_{\mathcal{I}} b_1 \dots b_{k'} \equiv k = k' \wedge (\forall i = 1, \dots, k : \exists X_i \in \mathcal{I} : a_i, b_i \in X_i)$$

for sequences $a_1 \dots a_k, b_1 \dots b_{k'} \in N^*$.

7.2 State Equivalence Class Partitions

Given the state space S of an RIOSTS $\mathcal{S} = (S, \underline{s}, R)$, a natural equivalence relation $\sim$ on S is introduced by defining

$$s \sim r \equiv L(s) = L(r)$$

The set of $\sim$-equivalence classes is denoted by $S/_\sim$.

Now let $\mathcal{S}$ be a completely defined RIOSTS such that D_M and D_O are finite, while the input vector domain D_I may be infinite. Define an enumeration of all internal state values combined with all output values by

$$\{(\mathbf{d}_1, \mathbf{e}_1), \dots, (\mathbf{d}_n, \mathbf{e}_n)\} = \\ \{(\mathbf{d}, \mathbf{e}) \in D_M \times D_O \mid \exists s \in S_Q : \ s \text{ is reachable and } s(\mathbf{m}, \mathbf{y}) = (\mathbf{d}, \mathbf{e})\}.$$

In the above definition, recall that $s(\mathbf{m}, \mathbf{y})$ is an abbreviation for the tuple $(s(m_1), \dots, s(m_k), s(y_1), \dots, s(y_p))$, where $M = \{m_1, \dots, m_k\}$ is the set of internal variable symbols, and $O = \{y_1, \dots, y_p\}$ is the set of output variables.

For $i = 1, \dots, n$, let A_i, be the set of all quiescent states s with $s(\mathbf{m}, \mathbf{y}) = (\mathbf{d}_i, \mathbf{e}_i)$. By definition, all elements of A_i coincide in their valuations of internal variables and output variables, but they differ in their input valuations. We call $\mathcal{A} = \{A_1, \dots, A_n\}$ the *MO-partition* of S_Q. For any $s, r \in A_i$, $s \oplus \{\mathbf{x} \mapsto \mathbf{c}\} = r \oplus \{\mathbf{x} \mapsto \mathbf{c}\}$, and consequently $s/\mathbf{c} = r/\mathbf{c}$, for any input vector $\mathbf{c} \in D_I$. Hence $L(s) = L(r)$, so s and r are I/O-equivalent. As a consequence, the MO-partition $\mathcal{A}$ is a refinement of the partition $S/_\sim$.

7.3 Input Equivalence Class Partitions

Definition 7.1 *Given an RIOSTS* $\mathcal{S} = (S, \underline{s}, R)$*, two input vectors* $\mathbf{c}, \mathbf{c}' \in D_I$ *are called* equivalent*,* $(\mathbf{c} \sim \mathbf{c}')$*, if and only if*

$$\forall s \in S_Q, \overline{\mathbf{x}} \in D_I^* : L(s, \mathbf{c}.\overline{\mathbf{x}}) = L(s, \mathbf{c}'.\overline{\mathbf{x}})$$

The set $D_I/_\sim = \{[\mathbf{c}] \mid \mathbf{c} \in D_I \wedge [\mathbf{c}] = \{\mathbf{c}' \mid \mathbf{c}' \sim \mathbf{c}\}\}$ *is an* input equivalence class partition (IECP) *of* $\mathcal{S}$*.*

Recall that $L(s, \mathbf{c}.\overline{\mathbf{x}})$ contains all output traces $\overline{\mathbf{y}}$ satisfying $\mathbf{c}.\overline{\mathbf{x}}/\overline{\mathbf{y}} \in L(s)$. So the intuition behind Definition 7.1 is that equivalent inputs, when extended in an arbitrary way by input sequences $\overline{\mathbf{x}}$, will always lead to the same sets of possible outputs. For deterministic systems, this output is uniquely determined, so $L(s, \mathbf{c}.\overline{\mathbf{x}}) = L(s, \mathbf{c}'.\overline{\mathbf{x}})$ contains a single output trace.

It is easy to see that $\sim$ is indeed an equivalence relation on D_I. The following lemma shows that $\sim$-equivalent input traces produce outputs from the same set, when applied to the same quiescent state. To this end, the equivalence relation is extended in the natural way to input sequences as described in Section 7.1: $\overline{\mathbf{x}} \sim \overline{\mathbf{x}}'$ if and only if $\overline{\mathbf{x}}$ and $\overline{\mathbf{x}}'$ have the same length, and each pair of corresponding elements in $\overline{\mathbf{x}}$ and $\overline{\mathbf{x}}'$ is equivalent according to Definition 7.1.

Lemma 7.1 *Given an RIOSTS* $\mathcal{S} = (S, \underline{s}, R)$*, let* $\overline{\mathbf{x}} \sim \overline{\mathbf{x}}'$*. Then*

$$\forall s \in S_Q : L(s, \overline{\mathbf{x}}) = L(s, \overline{\mathbf{x}}') \tag{7.1}$$

Proof. Let $\overline{\mathbf{x}}, \overline{\mathbf{x}}'$ be two equivalent input traces of length $n \geq 1$. In the case $n = 1$, there are $\mathbf{c}, \mathbf{c}' \in D_I$, such that $\overline{\mathbf{x}} = \mathbf{c}$, $\overline{\mathbf{x}}' = \mathbf{c}'$ and $\mathbf{c} \sim \mathbf{c}'$, so the statement holds according to Definition 7.1. Let $k \geq 1$ be a positive integer. Suppose that Statement (7.1) holds for any $n \leq k$. Now let $n = k + 1$ and suppose that $\overline{\mathbf{x}} = \mathbf{c}_1.\overline{\mathbf{x}}_1.\mathbf{c}_{k+1}$ and $\overline{\mathbf{x}}' = \mathbf{c}'_1.\overline{\mathbf{x}}'_1.\mathbf{c}'_{k+1}$. Then $\mathbf{c}_1 \sim \mathbf{c}'_1$, and $\overline{\mathbf{x}}_1.\mathbf{c}_{k+1} \sim \overline{\mathbf{x}}'_1.\mathbf{c}'_{k+1}$ are of length $n - 1 = k$. Let s be any quiescent state. Let

$\overline{\mathbf{y}} = \boldsymbol{e}_1.\overline{\mathbf{y}}_1 \in \mathsf{D}_\mathsf{O}^*$ be any nonempty output sequence. Then

$$
\begin{aligned}
\overline{\mathbf{x}}/\overline{\mathbf{y}} \in \mathsf{L}(\mathsf{s}) \quad \Leftrightarrow \quad & [\overline{\mathbf{x}} = \mathbf{c}_1.\overline{\mathbf{x}}_1.\mathbf{c}_{\mathsf{k}+1} \wedge \overline{\mathbf{y}} = \boldsymbol{e}_1.\overline{\mathbf{y}}_1] \\
& (\mathbf{c}_1.\overline{\mathbf{x}}_1.\mathbf{c}_{\mathsf{k}+1})/(\boldsymbol{e}_1.\overline{\mathbf{y}}_1) \in \mathsf{L}(\mathsf{s}) \\
\Leftrightarrow \quad & [\mathbf{c}_1 \sim \mathbf{c}_1'] \\
& (\mathbf{c}_1'.\overline{\mathbf{x}}_1.\mathbf{c}_{\mathsf{k}+1})/(\boldsymbol{e}_1.\overline{\mathbf{y}}_1) \in \mathsf{L}(\mathsf{s}) \\
\Leftrightarrow \quad & [\text{Definition of } \mathsf{L}(\mathsf{s})] \\
& \exists \mathsf{s}_1' \in \mathsf{s}\text{-}\underline{\text{after}}\text{-}\mathbf{c}_1'/\boldsymbol{e}_1 \wedge (\overline{\mathbf{x}}_1.\mathbf{c}_{\mathsf{k}+1})/\overline{\mathbf{y}}_1 \in \mathsf{L}(\mathsf{s}_1') \\
\Leftrightarrow \quad & [\overline{\mathbf{x}}_1.\mathbf{c}_{\mathsf{k}+1} \sim \overline{\mathbf{x}}_1'.\mathbf{c}_{\mathsf{k}+1}' \text{ are of length } \mathsf{k} \text{ and induction hypothesis}] \\
& \exists \mathsf{s}_1' \in \mathsf{s}\text{-}\underline{\text{after}}\text{-}\mathbf{c}_1'/\boldsymbol{e}_1 \wedge (\overline{\mathbf{x}}_1'.\mathbf{c}_{\mathsf{k}+1}')/\overline{\mathbf{y}}_1 \in \mathsf{L}(\mathsf{s}_1') \\
\Leftrightarrow \quad & [\text{Definition of } \mathsf{L}(\mathsf{s})] \\
& (\mathbf{c}_1'.\overline{\mathbf{x}}_1'.\mathbf{c}_{\mathsf{k}+1}')/(\boldsymbol{e}_1.\overline{\mathbf{y}}_1) \in \mathsf{L}(\mathsf{s}) \\
\Leftrightarrow \quad & [\overline{\mathbf{x}}' = \mathbf{c}_1'.\overline{\mathbf{x}}_1'.\mathbf{c}_{\mathsf{k}+1}' \wedge \overline{\mathbf{y}} = \boldsymbol{e}_1.\overline{\mathbf{y}}_1] \\
& \overline{\mathbf{x}}'/\overline{\mathbf{y}} \in \mathsf{L}(\mathsf{s})
\end{aligned}
$$

This proves that $\mathsf{L}(\mathsf{s}, \overline{\mathbf{x}}) = \mathsf{L}(\mathsf{s}, \overline{\mathbf{x}}')$. □

The following Lemma 7.2 states that – up to refinement – $\mathsf{D}_\mathsf{I}/_\sim$ is the *only* way to partition D_I, such that Property (7.1) is satisfied.

Lemma 7.2 *Given an RIOSTS $\mathcal{S} = (\mathsf{S}, \underline{\mathsf{s}}, \mathsf{R})$, let $\mathcal{I}$ be any partition of D_I. Then the following statements are equivalent.*

1. *$\mathcal{I}$ is a refinement of $\mathsf{D}_\mathsf{I}/_\sim$, with $\sim$ specified in Definition 7.1.*
2. *Property (7.1) holds for all $\overline{\mathbf{x}}, \overline{\mathbf{x}}' \in \mathsf{D}_\mathsf{I}^*$ with $\overline{\mathbf{x}} \sim_\mathcal{I} \overline{\mathbf{x}}'$, that is, $\forall \mathsf{s} \in \mathsf{S}_\mathsf{Q} : \mathsf{L}(\mathsf{s}, \overline{\mathbf{x}}) = \mathsf{L}(\mathsf{s}, \overline{\mathbf{x}}')$.*

Proof. To show that Statement 1 implies Statement 2, let $\mathcal{I}$ be a refinement of $\mathsf{D}_\mathsf{I}/_\sim$. Then $\overline{\mathbf{x}} \sim_\mathcal{I} \overline{\mathbf{x}}'$ implies $\overline{\mathbf{x}} \sim \overline{\mathbf{x}}'$, for any pair of input traces $\overline{\mathbf{x}}, \overline{\mathbf{x}}' \in \mathsf{D}_\mathsf{I}^*$. Then Lemma 7.1 implies the validity of Property (7.1).

To show that Statement 2 implies Statement 1, let $\mathcal{I}$ be a partition of D_I satisfying the Property (7.1) for all $\overline{\mathbf{x}} \sim_\mathcal{I} \overline{\mathbf{x}}'$. We show that for any $\mathbf{c}, \mathbf{c}' \in \mathsf{D}_\mathsf{I}$, $\mathbf{c} \sim_\mathcal{I} \mathbf{c}'$ implies $\mathbf{c} \sim \mathbf{c}'$. To this end, let $\overline{\mathbf{x}} \in \mathsf{D}_\mathsf{I}^*$ be any input trace and $\overline{\mathbf{y}} \in \mathsf{D}_\mathsf{O}^*$ be any output trace and $\mathsf{s} \in \mathsf{S}_\mathsf{Q}$ be any quiescent state. Then $\mathbf{c}.\overline{\mathbf{x}} \sim_\mathcal{I} \mathbf{c}'.\overline{\mathbf{x}}$ and from Property (7.1) we have $\mathsf{L}(\mathsf{s}, \mathbf{c}.\overline{\mathbf{x}}) = \mathsf{L}(\mathsf{s}, \mathbf{c}'.\overline{\mathbf{x}})$. □

7.4 The Transition Index Function

For calculating input equivalence class partitions, it is helpful to introduce partitions of the state space and an auxiliary function indicating how members of one state class may be mapped under given input vectors to certain target classes.

Let $\mathcal{S} = (S, s_0, R)$ be a completely defined RIOSTS such that D_M and D_O are finite, while the input vector domain D_I may be infinite. Define an enumeration of all internal state values combined with all output values by

$$\{(\mathbf{d}_1, \mathbf{e}_1), \ldots, (\mathbf{d}_n, \mathbf{e}_n)\} = \\ \{(\mathbf{d}, \mathbf{e}) \in D_M \times D_O \mid \exists s \in S_Q : \ s \text{ is reachable and } s(\mathbf{m}, \mathbf{y}) = (\mathbf{d}, \mathbf{e})\}$$

Let A_i, $i = 1, \ldots, n$, be the set consisting of all quiescent states s with $s(\mathbf{m}, \mathbf{y}) = (\mathbf{d}_i, \mathbf{e}_i)$. Then for any $s, r \in A_i$, $s \oplus \{\mathbf{x} \mapsto \mathbf{c}\} = r \oplus \{\mathbf{x} \mapsto \mathbf{c}\}$, and consequently $s/\mathbf{c} = r/\mathbf{c}$, for any input vector $\mathbf{c} \in D_I$. Hence $L(s) = L(r)$, so s and r are I/O-equivalent. We call $\mathcal{A}$ the *MO-partition* of S_Q. The *transition index function*

$$\delta : D_I \to (\mathbb{P}(\{1, \ldots, n\}))^n; \quad \mathbf{c} \mapsto (\delta_1(\mathbf{c}), \ldots, \delta_n(\mathbf{c}))$$

is specified by

$$j \in \delta_i(\mathbf{c}) \text{ if and only if } (s/\mathbf{c}) \cap A_j \neq \varnothing \text{ for some } s \in A_i.$$

Intuitively speaking, each component function δ_i of δ maps an input vector $\mathbf{c}$ to the indexes j of all MO-partition elements A_j that may be reached from A_i-states when applying $\mathbf{c}$. For deterministic RIOSTS $\delta(\mathbf{c})$ always contains exactly one element. δ is well-defined: for any $i = 1, \ldots, n$, any $s_1, s_2 \in A_i$, $s_1(\mathbf{m}, \mathbf{y}) = s_2(\mathbf{m}, \mathbf{y})$ and therefore $s_1 \oplus \{\mathbf{x} \mapsto \mathbf{c}\} = s_2 \oplus \{\mathbf{x} \mapsto \mathbf{c}\}$. Hence $s_1/\mathbf{c} = s_2/\mathbf{c}$, consequently $(s_1/\mathbf{c}) \cap A_j \neq \varnothing \Leftrightarrow (s_2/\mathbf{c}) \cap A_j \neq \varnothing$.

Lemma 7.3 *The set* $\mathcal{I} = \{X(\mathbf{c}) \mid \mathbf{c} \in D_I\}$*, where*

$$X(\mathbf{c}) = \{\mathbf{c}' \in D_I \mid \delta(\mathbf{c}') = \delta(\mathbf{c})\},$$

is a finite partition of input domain D_I *and satisfies the Property (7.1).*

Proof. From the definition of δ and $\mathcal{I}$ we conclude that $|\mathcal{I}|$ is bounded by $|\delta(D_I)| \leq 2^{(n^n)}$ and therefore finite. Each $X(\mathbf{c}) \in \mathcal{I}$ is non-empty because

it contains at least $\mathbf{c}$. The definition of $\mathcal{I}$ further implies that the union of all $X \in \mathcal{I}$ equals D_I. Moreover, $\mathbf{c}, \mathbf{c}' \in D_I$ are contained in the same $X \in \mathcal{I}$ if and only if $\delta(\mathbf{c}) = \delta(\mathbf{c}')$. Therefore $X(\mathbf{c})$ and $X(\mathbf{c}')$ are either identical or disjoint. Hence $\mathcal{I}$ is a finite partition of input domain D_I. Let $\mathbf{c}, \mathbf{c}' \in D_I$ be any input vectors with $\mathbf{c} \sim_{\mathcal{I}} \mathbf{c}'$. Let $s \in S_Q$ be any quiescent state. Then there is a unique $i \in \{1, \ldots, n\}$ with $s \in A_i$ and

$$s/\mathbf{c} = \bigcup_{j \in \delta_i(\mathbf{c})} \{\{x \mapsto \mathbf{c}, m \mapsto \mathbf{d}_j, y \mapsto \mathbf{e}_j\}\}$$

and

$$s/\mathbf{c}' = \bigcup_{j \in \delta_i(\mathbf{c}')} \{\{x \mapsto \mathbf{c}', m \mapsto \mathbf{d}_j, y \mapsto \mathbf{e}_j\}\}$$

Since $\mathbf{c} \sim_{\mathcal{I}} \mathbf{c}'$, we have $\delta_i(\mathbf{c}) = \delta_i(\mathbf{c}')$, so $s/\mathbf{c}$ and $s/\mathbf{c}'$ have the same cardinality, and their elements only differ in the input valuations $\mathbf{c}$ and $\mathbf{c}'$, respectively. As a consequence, for any $s_1 \in s/\mathbf{c}'$ and any $\mathbf{c}_1 \in D_I$, we find an $s_2 \in s/\mathbf{c}'$ such that $s_1 \oplus \{x \mapsto \mathbf{c}_1\} = s_2 \oplus \{x \mapsto \mathbf{c}_1\}$ and vice versa. This shows that $(s/\mathbf{c}.\mathbf{c}_1)(m, y) = (s/\mathbf{c}'.\mathbf{c}_1)(m, y)$, and this process can be repeated with additional arbitrary input vectors $\mathbf{c}_2, \mathbf{c}_3, \ldots$. Now we have shown that $(s/\mathbf{c}.\overline{x})(y) = (s/\mathbf{c}'.\overline{x})(y)$ holds for all $\overline{x} \in D_I^*$. According to Definition 7.1, this proves $\mathbf{c} \sim \mathbf{c}'$, so $\sim_{\mathcal{I}}$ is a refinement of the partition $D_I/_\sim$. Applying Lemma 7.2, this shows that $\mathcal{I}$ satisfies Property (7.1). This completes the proof. □

In [31], an algorithm is presented showing how to calculate input equivalence class partitions with the help of the transition index function. The basic principle of this calculation is illustrated in the following example.

Example 8. Consider again the alarm system from Example 7. Its reachable pairs of internal states m and outputs y are

$$\{(0, \texttt{OK}), (1, \texttt{OK}), (2, \texttt{ALARM})\} \subseteq D_m \times D_y.$$

From the specification of quiescent states S_Q given in Example 7, the following MO-partition is derived.

$$\begin{aligned}
\mathcal{A} &= \{S_0, S_1, S_2\} \\
S_0 &= \{s : V \to D \mid s(x) < \max \wedge s(m) = 0 \wedge s(y) = \texttt{OK}\} \\
S_1 &= \{s : V \to D \mid s(x) = \max \wedge s(m) = 1 \wedge s(y) = \texttt{OK}\} \\
S_2 &= \{s : V \to D \mid s(x) > \max - \delta \wedge s(m) = 2 \wedge s(y) = \texttt{ALARM}\}
\end{aligned}$$

The transition index function $\delta(\mathbf{c}) = (\delta_0(\mathbf{c}), \delta_1(\mathbf{c}), \delta_2(\mathbf{c}))$ is constant on the input intervals

$$X_1 = [0, \max - \delta], \quad X_2 = (\max - \delta, \max), \quad X_3 = \{\max\}, \quad X_4 = (\max, \infty)$$

and has the following function table

	X_1	X_2	X_3	X_4
δ_0	$\{0\}$	$\{0\}$	$\{1, 2\}$	$\{2\}$
δ_1	$\{0\}$	$\{0\}$	$\{1\}$	$\{2\}$
δ_2	$\{0\}$	$\{2\}$	$\{2\}$	$\{2\}$

Therefore $\mathcal{I} = \{X_1, X_2, X_3, X_4\}$ is a finite partitioning of D_I which satisfies Property (7.1), and hence it is a refinement of the IECP $D_I/_\sim$. □

7.5 State Machine Abstraction of Equivalence Class Partitions

Let $\mathcal{A}, \mathcal{I}$ be refinements of $S_Q/_\sim, D_I/_\sim$, respectively. We call $(\mathcal{A}, \mathcal{I})$ an *equivalence class partition pair* of $\mathcal{S}$. If $\mathcal{A}$ and $\mathcal{I}$ are both finite, $(\mathcal{A}, \mathcal{I})$ is called a *finite equivalence class partition pair*.

Definition 7.2 *Let $\mathcal{S} = (S, \underline{s}, R)$ be an RIOSTS with finite D_O and D_M. Let $(\mathcal{A}, \mathcal{I})$ be a finite equivalence class partition pair of $\mathcal{S}$. Then $M = (\mathcal{A}, \underline{A}, \mathcal{I}, D_O, h_M)$ with initial state $\underline{A} \in \mathcal{A}$ containing $\underline{s}$, transition relation $h_M \subseteq \mathcal{A} \times \mathcal{I} \times D_O \times \mathcal{A}$, such that*

$$(A, X, \boldsymbol{e}, A') \in h_M \Leftrightarrow \exists s \in A, s' \in A', \mathbf{c} \in X : (s' \in s/\mathbf{c} \wedge s'(\mathbf{y}) = \boldsymbol{e})$$

is called the FSM induced by $\mathcal{S}$ and $(\mathcal{A}, \mathcal{I})$.

Example 9. For the equivalence class partition pair $(\mathcal{A}, \mathcal{I})$ elaborated for the alarm indication system $\mathcal{S}$ in Example 8, the FSM induced by $\mathcal{S}$ and $(\mathcal{A}, \mathcal{I})$ is shown in Fig. 7.1. Its transitions can be derived from the transition index function shown in Example 8. □

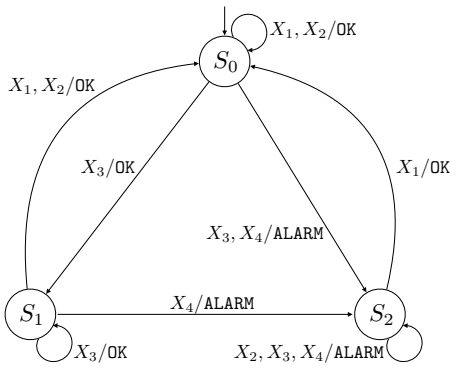


Figure 7.1: FSM induced by $\mathcal{S}$ and $(\mathcal{A}, \mathcal{I})$ from Example 7 and 8.

Definition 7.3 *Let $\mathcal{I}$ be a refinement of $\mathsf{D}_\mathrm{I}/_\sim$. Let $\overline{\mathbf{X}} \in \mathcal{I}^*$ be an input equivalence class sequence and $\overline{\mathbf{c}} \in \mathsf{D}_\mathrm{I}^*$ an input vector sequence.* $\overline{\mathbf{x}}$ is incident to $\overline{\mathbf{X}}$ *(written $\overline{\mathbf{x}}\,\mathrm{I}\,\overline{\mathbf{X}}$) if and only if $|\overline{\mathbf{x}}| = |\overline{\mathbf{X}}|$ and $\overline{\mathbf{x}}(\mathfrak{i}) \in \overline{\mathbf{X}}(\mathfrak{i})$, for $\mathfrak{i} = 1, \ldots, |\overline{\mathbf{x}}|$.*

Let $\mathcal{I}$ be a refinement of $\mathsf{D}_\mathrm{I}/_\sim$. By definition, two input sequences $\overline{\mathbf{x}}, \overline{\mathbf{x}}'$ are $\mathcal{I}$-equivalent, ($\overline{\mathbf{x}} \sim_\mathcal{I} \overline{\mathbf{x}}'$) if and only if they are incident to the same sequence of of partition elements from $\mathcal{I}$. Since $\mathcal{I}$ is a refinement of $\mathsf{D}_\mathrm{I}/_\sim$, $\overline{\mathbf{x}} \sim_\mathcal{I} \overline{\mathbf{x}}'$ implies $\overline{\mathbf{x}} \sim \overline{\mathbf{x}}'$.

Theorem 7.1 *Let $(\mathcal{A}, \mathcal{I})$ be a finite equivalence class partition pair of $\mathcal{S}$ and M the induced FSM. Let $\overline{\mathbf{x}} \in \mathsf{D}_\mathrm{I}^*$ and $\overline{\mathbf{X}} \in \mathcal{I}^*$, such that $\overline{\mathbf{x}}\,\mathrm{I}\,\overline{\mathbf{X}}$. Then for any output vector sequence $\overline{\mathbf{y}} \in \mathsf{D}_\mathrm{O}^*$:*

$$\overline{\mathbf{X}}/\overline{\mathbf{y}} \in \mathsf{L}(\mathsf{M}) \Leftrightarrow \overline{\mathbf{x}}/\overline{\mathbf{y}} \in \mathsf{L}(\mathcal{S})$$

Proof. Let $\overline{\mathbf{X}} = \mathsf{X}_1 \ldots \mathsf{X}_n \in \mathcal{I}^*$, $\overline{\mathbf{x}} = \mathbf{c}_1 \ldots \mathbf{c}_n \in \mathsf{D}_\mathrm{I}^*$ with $\mathbf{c}_\mathfrak{i} \in \mathsf{X}_\mathfrak{i}, \mathfrak{i} = 1, \ldots, n$. Let $\overline{\mathbf{y}} = \boldsymbol{e}_1 \ldots \boldsymbol{e}_n \in \mathsf{D}_\mathrm{O}^*$. Suppose $\overline{\mathbf{X}}/\overline{\mathbf{y}} \in \mathsf{L}(\mathsf{M})$. Then, by definition of h_M, there exists a state sequence $s_0 \ldots s_n \in S^*$ with $s_0 = \underline{s}$ and an input sequence $\overline{\mathbf{x}}' = \mathbf{c}'_1 \ldots \mathbf{c}'_n \in \mathsf{D}_\mathrm{I}^*$, such that $\mathbf{c}'_\mathfrak{i} \in \mathsf{X}_\mathfrak{i}, \mathfrak{i} = 1, \ldots, n$ and $s_\mathfrak{i} \in (s_{\mathfrak{i}-1}\text{-}\underline{\text{after}}\text{-}\mathbf{c}'_\mathfrak{i}/\boldsymbol{e}_\mathfrak{i})$ for all $\mathfrak{i} = 1, \ldots, n$. By definition of $\mathsf{L}(\mathcal{S})$, this implies that $\overline{\mathbf{x}}'/\overline{\mathbf{y}} \in \mathsf{L}(\mathcal{S})$. Since $\overline{\mathbf{x}} \sim_\mathcal{I} \overline{\mathbf{x}}'$, by Lemma 7.1, $\overline{\mathbf{x}}/\overline{\mathbf{y}} \in \mathsf{L}(\mathcal{S})$.

Now suppose $\overline{\mathbf{x}}/\overline{\mathbf{y}} \in \mathsf{L}(\mathcal{S})$. By definition of $\mathsf{L}(\mathcal{S})$, there exists a state sequence $s_0 \ldots s_n \in S^*$ with $s_0 = \underline{s}$, such that $s_1(\boldsymbol{x}, \boldsymbol{y}) \ldots s_n(\boldsymbol{x}, \boldsymbol{y}) = \overline{\mathbf{x}}/\overline{\mathbf{y}}$ and

then $s_i \in (s_{i-1}\text{-}\underline{\text{after}}\text{-}\mathbf{c}_i/\boldsymbol{e}_i)$ for all $i = 1, \ldots, n$. Since $\overline{\mathbf{x}}\, I\, \overline{\mathbf{X}}$, by definition of h_M, we have $\overline{\mathbf{X}}/\overline{\mathbf{y}} \in L(M)$. □

Lemma 7.4 *Let $(\mathcal{A}_1, \mathcal{I})$, $(\mathcal{A}_2, \mathcal{I})$ be two finite equivalence class partition pairs of $\mathcal{S}$. Then the induced FSMs M_1, M_2 are I/O-equivalent.*

Proof. Let $\overline{\mathbf{X}} \in \mathcal{I}^*$ and $\overline{\mathbf{x}} \in D_I^*$, such that $\overline{\mathbf{x}}\, I\, \overline{\mathbf{X}}$. Then by Theorem 7.1, for any output vector sequence $\overline{\mathbf{y}} \in D_O^*$, we have $\overline{\mathbf{X}}/\overline{\mathbf{y}} \in L(M_1) \Leftrightarrow \overline{\mathbf{x}}/\overline{\mathbf{y}} \in L(\mathcal{S}) \Leftrightarrow \overline{\mathbf{X}}/\overline{\mathbf{y}} \in L(M_2)$. This proves $M_1 \sim M_2$. □

Lemma 7.5 *Let $(\mathcal{A}_1, \mathcal{I})$, $(\mathcal{A}_2, \mathcal{I})$ be two finite equivalence class partition pairs of $\mathcal{S}$ and M_1, M_2 the induced FSMs. Let $\overline{M}_1, \overline{M}_2$ be the prime machines associated with M_1, M_2, respectively. Then $\overline{M}_1, \overline{M}_2$ are isomorphic and I/O-equivalent.*

Proof. This follows directly from Lemma 7.4 and Lemma 3.1. □

Hence, for a given $\mathcal{I}$, finite refinement of input equivalence class partition $D_I/_\sim$, there is, up to isomorphism, a unique minimal, observable FSM induced by $\mathcal{S}$. This machine is obtained by constructing the prime machine associated with any FSM induced by $\mathcal{S}$ and some finite equivalence class partition pair $(\mathcal{A}, \mathcal{I})$. This FSM is denoted by $\overline{M}(\mathcal{S}, \mathcal{I})$, since it is independent on $\mathcal{A}$.

Lemma 7.6 *Suppose $\mathcal{S}$ is deterministic. Then for any finite refinement $\mathcal{I}$ of $D_I/_\sim$, the FSM $\overline{M}(\mathcal{S}, \mathcal{I})$ is also deterministic.*

Proof. Let $\mathcal{I}$ be a finite refinement of $D_I/_\sim$. Let M be the FSM induced by $(\mathcal{A} = S_Q/_\sim, \mathcal{I})$. Suppose $(A, X, \boldsymbol{e}_1, A_1), (A, X, \boldsymbol{e}_2, A_2) \in h_M$. By definition, there exist $s, r \in A, \mathbf{c}_1, \mathbf{c}_2 \in X, s_1 \in A_1, s_2 \in A_2$, such that $s_1 \in s/\mathbf{c}_1 \wedge s_1(\mathbf{y}) = \boldsymbol{e}_1$ and $s_2 \in r/\mathbf{c}_2 \wedge s_2(\mathbf{y}) = \boldsymbol{e}_2$. Since $\mathcal{S}$ is deterministic, $s_1 = s/\mathbf{c}_1$ and $s_2 = r/\mathbf{c}_2$. Since $\mathbf{c}_1, \mathbf{c}_2 \in X$, $\mathbf{c}_1 \sim \mathbf{c}_2$.

$\forall \overline{\mathbf{x}} \in \mathsf{D}_\mathsf{I}^*, \overline{\mathbf{y}} \in \mathsf{D}_\mathsf{O}^*$:

$$
\begin{array}{lll}
\mathsf{s}_1(\mathbf{y}) = \boldsymbol{e}_1 \wedge \overline{\mathbf{x}}/\overline{\mathbf{y}} \in \mathsf{L}(\mathsf{s}_1) & \Leftrightarrow & \mathbf{c}_1/\boldsymbol{e}_1 \in \mathsf{L}(\mathsf{s}) \wedge (\mathbf{c}_1.\overline{\mathbf{x}})/(\boldsymbol{e}_1.\overline{\mathbf{y}}) \in \mathsf{L}(\mathsf{s}) \quad [\mathsf{s}_1 = \mathsf{s}/\mathbf{c}_1] \\
& \Leftrightarrow & \mathbf{c}_2/\boldsymbol{e}_1 \in \mathsf{L}(\mathsf{s}) \wedge (\mathbf{c}_2.\overline{\mathbf{x}})/(\boldsymbol{e}_1.\overline{\mathbf{y}}) \in \mathsf{L}(\mathsf{s}) \quad [\mathbf{c}_1 \sim \mathbf{c}_2] \\
& \Leftrightarrow & \mathbf{c}_2/\boldsymbol{e}_1 \in \mathsf{L}(\mathsf{r}) \wedge (\mathbf{c}_2.\overline{\mathbf{x}})/(\boldsymbol{e}_1.\overline{\mathbf{y}}) \in \mathsf{L}(\mathsf{r}) \quad [\mathsf{L}(\mathsf{s}) = \mathsf{L}(\mathsf{r})] \\
& \Leftrightarrow & (\mathsf{r}/\mathbf{c}_2)(\mathbf{y}) = \boldsymbol{e}_1 \wedge \overline{\mathbf{x}}/\overline{\mathbf{y}} \in \mathsf{L}(\mathsf{r}/\mathbf{c}_2) \quad [\mathcal{S} \text{ is deterministic}] \\
& \Leftrightarrow & \mathsf{s}_2(\mathbf{y}) = \boldsymbol{e}_1 \wedge \overline{\mathbf{x}}/\overline{\mathbf{y}} \in \mathsf{L}(\mathsf{s}_2) \quad [\mathsf{s}_2 = \mathsf{r}/\mathbf{c}_2] \\
& \Leftrightarrow & \boldsymbol{e}_1 = \boldsymbol{e}_2 \wedge \overline{\mathbf{x}}/\overline{\mathbf{y}} \in \mathsf{L}(\mathsf{s}_2) \quad [\mathsf{s}_2(\mathbf{y}) = \boldsymbol{e}_2]
\end{array}
$$

Since $\mathcal{A} = \mathsf{S}_\mathsf{Q}/_\sim$, $\mathsf{L}(\mathsf{s}_1) = \mathsf{L}(\mathsf{s}_2)$ implies $\mathsf{A}_1 = \mathsf{A}_2$. Since $\boldsymbol{e}_1 = \boldsymbol{e}_2$ and $\mathsf{A}_1 = \mathsf{A}_2$, M is deterministic and therefore observable. The minimal FSM equivalent to M is also deterministic. □

The following theorem shows that I/O-equivalence and reduction between RIOSTSs can be determined by checking these properties for their FSM abstractions.

Theorem 7.2 *Let $\mathcal{S}_\mathsf{i}, \mathsf{i} = 1, 2$ be RIOSTSs with the same signature. Let $\mathcal{I}$ be a joint input equivalence class partitioning.*[1] *Then*

$$\mathcal{S}_1 \sim \mathcal{S}_2 \Leftrightarrow \overline{\mathsf{M}}(\mathcal{S}_1, \mathcal{I}) \sim \overline{\mathsf{M}}(\mathcal{S}_2, \mathcal{I}) \quad \textit{and} \quad \mathcal{S}_1 \preceq \mathcal{S}_2 \Leftrightarrow \overline{\mathsf{M}}(\mathcal{S}_1, \mathcal{I}) \preceq \overline{\mathsf{M}}(\mathcal{S}_2, \mathcal{I})$$

Proof. The theorem is a consequence of Theorem 7.1. For $\overline{\mathbf{x}} \in \mathsf{D}_\mathsf{I}^*$, let $\overline{\mathbf{X}}(\overline{\mathbf{x}}) \in \mathcal{I}^*$ denote the uniquely determined sequence of input equivalence

[1] For example, $\mathcal{I} = \mathcal{I}_1 \cap \mathcal{I}_2$, where $\mathcal{I}_1$ and $\mathcal{I}_2$ have been constructed for $\mathcal{S}_\mathsf{i}$ according to Lemma 7.3.

classes satisfying $\overline{\mathbf{x}}\,\mathrm{I}\,\overline{\mathbf{X}}$. Then

$$
\begin{aligned}
\mathcal{S}_1 \sim \mathcal{S}_2 \quad &\Leftrightarrow \quad \mathsf{L}(\mathcal{S}_1) = \mathsf{L}(\mathcal{S}_2) \\
&\quad\ \text{[Definition of I/O-equivalence]} \\
&\Leftrightarrow \quad \forall \overline{\mathbf{x}} \in \mathsf{D}_\mathsf{I}^*, \overline{\mathbf{y}} \in \mathsf{D}_\mathsf{O}^* : \overline{\mathbf{x}}/\overline{\mathbf{y}} \in \mathsf{L}(\mathcal{S}_1) \Leftrightarrow \overline{\mathbf{x}}/\overline{\mathbf{y}} \in \mathsf{L}(\mathcal{S}_2) \\
&\quad\ \text{[Definition of L]} \\
&\Leftrightarrow \quad \forall \overline{\mathbf{x}} \in \mathsf{D}_\mathsf{I}^*, \overline{\mathbf{y}} \in \mathsf{D}_\mathsf{O}^* : \overline{\mathbf{X}}(\overline{\mathbf{x}})/\overline{\mathbf{y}} \in \mathsf{L}(\overline{\mathsf{M}}(\mathcal{S}_1, \mathcal{I})) \Leftrightarrow \overline{\mathbf{X}}(\overline{\mathbf{x}})/\overline{\mathbf{y}} \in \mathsf{L}(\overline{\mathsf{M}}(\mathcal{S}_2, \mathcal{I})) \\
&\quad\ \text{[Theorem 7.1]} \\
&\Leftrightarrow \quad \forall \overline{\mathbf{X}} \in \mathcal{I}^*, \overline{\mathbf{y}} \in \mathsf{D}_\mathsf{O}^* : \overline{\mathbf{X}}/\overline{\mathbf{y}} \in \mathsf{L}(\overline{\mathsf{M}}(\mathcal{S}_1, \mathcal{I})) \Leftrightarrow \overline{\mathbf{X}}/\overline{\mathbf{y}} \in \mathsf{L}(\overline{\mathsf{M}}(\mathcal{S}_2, \mathcal{I})) \\
&\quad\ [\mathcal{I}^* = \{\overline{\mathbf{X}}(\overline{\mathbf{x}}) \mid \overline{\mathbf{x}} \in \mathsf{D}_\mathsf{I}^*\}] \\
&\Leftrightarrow \quad \mathsf{L}(\overline{\mathsf{M}}(\mathcal{S}_1, \mathcal{I})) = \mathsf{L}(\overline{\mathsf{M}}(\mathcal{S}_2, \mathcal{I})) \\
&\quad\ \text{[Definition of L]} \\
&\Leftrightarrow \quad \overline{\mathsf{M}}(\mathcal{S}_1, \mathcal{I}) \sim \overline{\mathsf{M}}(\mathcal{S}_2, \mathcal{I}) \\
&\quad\ \text{[Definition of FSM I/O-equivalence]}
\end{aligned}
$$

The proof for $\preceq$ is performed analogously. □

7.6 RIOSTS Sub-domains and Associated Model Maps – Proof of SC1

Fixing a signature $\mathtt{Sig} = (\mathsf{I}, \mathsf{O}, \mathsf{D}_\mathsf{I}, \mathsf{D}_\mathsf{O})$ such that D_O is finite, we restrict the set of RIOSTS over $\mathtt{Sig}$ to those operating on finitely many internal states. Given an arbitrary input partitioning $\mathcal{I}$, let $\mathcal{D}(\mathtt{Sig}, \mathcal{I})$ denote the subdomain of these RIOSTS $\mathcal{S}$ for which a state partition $\mathcal{A}$ can be found such that $(\mathcal{A}, \mathcal{I})$ is a finite equivalence partition pair of $\mathcal{S}$. Then the mapping $\overline{\mathsf{M}}(\mathcal{S}, \mathcal{I})$ constructed above induces the total function

$$\mathsf{T} : \mathcal{D}(\mathtt{Sig}, \mathcal{I}) \to \mathrm{FSM}(\mathcal{I}, \mathsf{D}_\mathsf{O}); \quad \mathcal{S} \mapsto \overline{\mathsf{M}}(\mathcal{S}, \mathcal{I})$$

We call T the *model map* from RIOSTS subdomain $\mathcal{D}(\mathtt{Sig}, \mathcal{I})$ to $\mathrm{FSM}(\mathcal{I}, \mathsf{D}_\mathsf{O})$. Re-written in terms of the model map, and referring to the satisfcation condition defined in Section 2.6, Theorem 7.2 can be re-formulated as follows.

Theorem 7.3 (Satisfaction Condition SC1) *Given an RIOSTS subdomain* $\mathcal{D}(\mathtt{Sig}, \mathcal{I})$*,*

$$\mathcal{S}_1 \sim \mathcal{S}_2 \Leftrightarrow \mathsf{T}(\mathcal{S}_1) \sim \mathsf{T}(\mathcal{S}_2) \quad \textit{and} \quad \mathcal{S}_1 \preceq \mathcal{S}_2 \Leftrightarrow \mathsf{T}(\mathcal{S}_1) \preceq \mathsf{T}(\mathcal{S}_2)$$

holds for all $\mathcal{S}_1, \mathcal{S}_2 \in \mathcal{D}(\mathtt{Sig}, \mathcal{I})$*, so* T *fulfils the satisfaction condition* **SC1***, introduced in Section 2.6.* □

7.7 Practical Calculation of the Model Map

7.7.1 Objectives

In this section an algorithm is introduced that allows for practical calculation of the model map, so that the input equivalence classes of an RIOSTS can be mechanically identified and represented as propositions. This algorithm requires mathematical constraint solvers, such as, for example, an SMT solver as the one described in [44, 53]. Such a constraint solver is needed again for calculating concrete representatives of the input equivalence classes represented by these propositions.

The material presented here is based on the algorithm described in [31], but here it is extended to nondeterministic RIOSTS, while [31] only covered the deterministic case. Applications of the algorithm for deterministic RIOSTS have been described in [5, 4]; in this section the nondeterministic alarm indication system introduced in Example 7 is used for illustrating the algorithm.

The algorithm addresses three non-trivial realisation aspects. (1) The transition relations of RIOSTS require to abstract consecutive transitions between transient states into a single one from the first transient state of this sequence to its quiescent post state. The effect of the consecutive transitions has to be aggregated by the abstracted one. (2) The I/O-equivalence classes have to be determined. (3) The initial IECP has to be calculated.

Note that (1) requires the original test models to be livelock free, because livelocks would imply the existence of infinite sequences of transitions between transient states, and these could not be abstracted to a single transient RIOSTS state. Indeed, only livelock free models are generally admissible for automated test generation, because otherwise an unbounded number of actions could be conceptually performed in zero time, that is, without waiting for another change of inputs.

7.7.2 DNF transformation

We start with an arbitrary first order representation $\mathcal{R}_1$ of the STS model's transition relation R_1 (the construction of such representations is described in more detail in [10, 2.1.1])): $\mathcal{R}_1$ has free variables in V referring to the pre-state of a transition, and variables in $V' = \{v' \mid v \in V\}$ referring to its post-state. The transition relation R_1 is determined by $\mathcal{R}_1$ through

$$\mathsf{R}_1 = \{(s_1, s_2) \in S \times S \mid \mathcal{R}_1[s_1(v)/v, s_2(v)/v' | v \in V, v' \in V']\}$$

where $\mathcal{R}_1[s_1(v)/v, s_2(v)/v' | v \in V, v' \in V']$ denotes $\mathcal{R}_1$ with every unprimed variable v replaced by its pre-state value $s_1(v)$ and every primed variable v' replaced by the post-state value $s_2(v)$.

Example 10. The transition relation of the alarm indication system in Example 7 can be re-written in first order form as

$$\begin{aligned}
\mathcal{R}_1 &\equiv \bigvee_{i=0}^{8} \eta_i \\
\eta_0 &\equiv x < \max \wedge (m, y) = (0, \mathtt{OK}) \wedge (m', y') = (m, y) \\
\eta_1 &\equiv x = \max \wedge (m, y) = (1, \mathtt{OK}) \wedge (m', y') = (m, y) \\
\eta_2 &\equiv x > \max - \delta \wedge (m, y) = (2, \mathtt{ALARM}) \wedge (m', y') = (m, y) \\
\eta_3 &\equiv x = \max \wedge (m, y) = (0, \mathtt{OK}) \wedge (m', y') = (1, \mathtt{OK}) \wedge x' = x \\
\eta_4 &\equiv x = \max \wedge (m, y) = (0, \mathtt{OK}) \wedge (m', y') = (2, \mathtt{ALARM}) \wedge x' = x \\
\eta_5 &\equiv x > \max \wedge (m, y) = (0, \mathtt{OK}) \wedge (m', y') = (2, \mathtt{ALARM}) \wedge x' = x \\
\eta_6 &\equiv x \leq \max - \delta \wedge (m, y) = (2, \mathtt{ALARM}) \wedge (m', y') = (0, \mathtt{OK}) \wedge x' = x \\
\eta_7 &\equiv x > \max \wedge (m, y) = (1, \mathtt{OK}) \wedge (m', y') = (2, \mathtt{ALARM}) \wedge x' = x \\
\eta_8 &\equiv x < \max \wedge (m, y) = (1, \mathtt{OK}) \wedge (m', y') = (0, \mathtt{OK}) \wedge x' = x
\end{aligned}$$

□

The initial transition relation in first order form $\mathcal{R}_1$ is first transformed into disjunctive normal form (DNF) with atomic propositions p_i that are Boolean expressions over free variables from V. In each disjunct, the atomic propositions $p_1 \wedge \cdots \wedge p_k$ are arranged in such a way that $p_1, \ldots, p_\ell$ have free variables in V only and $p_{\ell+1}, \ldots, p_k$ have free variables in V and V'. This means that $p_1 \wedge \cdots \wedge p_\ell$ represent the pre-condition for the disjunct to be applied, and $p_{\ell+1} \wedge \cdots \wedge p_k$ specifies the effect of this disjunct on the post-state represented by primed variables. Summarising, we gain an equivalent

representation

$$\mathcal{R}_2 \equiv \bigvee_{i=0}^{q} (\varphi_i \wedge \psi_i) \tag{7.2}$$

of $\mathcal{R}_1$, such that free(φ_i) $\subseteq V$ and free(ψ_i) $\subseteq V \cup V'$, and each atomic proposition in ψ_i contains at least one primed variable from V'.

Example 11. Continuing with Example 10, we construct

$$\mathcal{R}_2 \equiv \bigvee_{i=0}^{8} (\varphi_i \wedge \psi_i)$$

such that $(\varphi_i \wedge \psi_i) \equiv \eta_i$ for $i = 0, \ldots, 8$. This results in

i	φ_i	ψ_i
0	$x < max \wedge (m, y) = (0, \texttt{OK})$	$(m', y') = (m, y)$
1	$x = max \wedge (m, y) = (1, \texttt{OK})$	$(m', y') = (m, y)$
2	$x > max - \delta \wedge (m, y) = (2, \texttt{ALARM})$	$(m', y') = (m, y)$
3	$x = max \wedge (m, y) = (0, \texttt{OK})$	$(m', y') = (1, \texttt{OK}) \wedge x' = x$
4	$x = max \wedge (m, y) = (0, \texttt{OK})$	$(m', y') = (2, \texttt{ALARM}) \wedge x' = x$
5	$x > max \wedge (m, y) = (0, \texttt{OK})$	$(m', y') = (2, \texttt{ALARM}) \wedge x' = x$
6	$x \leq max - \delta \wedge (m, y) = (2, \texttt{ALARM})$	$(m', y') = (0, \texttt{OK}) \wedge x' = x$
7	$x > max \wedge (m, y) = (1, \texttt{OK})$	$(m', y') = (2, \texttt{ALARM}) \wedge x' = x$
8	$x < max \wedge (m, y) = (1, \texttt{OK})$	$(m', y') = (0, \texttt{OK}) \wedge x' = x$

□

7.7.3 Identification of quiescent states

With the first order representation from Equation (7.2) it is easy to identify the quiescent states of the RIOSTS: if ψ_i leaves some freedom to change inputs (i.e., does not contain propositions equivalent to $\mathbf{x}' = \mathbf{x}$) and does not allow changes of internal model variables and outputs, then φ_i specifies a subset of quiescent states. This identification can be mechanised by using a constraint solver and checking whether

$$\varphi_i \wedge \psi_i \wedge \mathbf{x}' \neq \mathbf{x}$$

has a solution. As a consequence, we can re-arrange the disjuncts of $\mathcal{R}_2$ such that the first q_0 specify transitions from quiescent states, and the ones with

indexes $q_0 < i \leq q$ specify transitions from transient states. As a result, the transition relation is represented by

$$\mathcal{R}_3 \equiv \bigvee_{i=1}^{q_0} (\varphi_i \wedge \psi_i) \vee \bigvee_{i=q_0+1}^{q} (\varphi_i \wedge \psi_i) \tag{7.3}$$

Example 12. Continuing with Example 11, we construct

$$\mathcal{R}_3 \equiv \bigvee_{i=0}^{2} (\varphi_i \wedge \psi_i) \vee \bigvee_{i=3}^{8} (\varphi_i \wedge \psi_i)$$

with φ_i, ψ_i specified as in Example 11. □

7.7.4 Rewriting the representation

Observing that the variable domains of $v \in M \cup O$ are finite, propositions γ with free variables in V, V' occurring in $\mathcal{R}_3$ can be re-written as

$$\gamma \equiv \bigvee_{\mathbf{d},\mathbf{d}' \in D_M, \mathbf{e},\mathbf{e}' \in D_O} \big(\gamma[\mathbf{d}/\mathbf{m}, \mathbf{e}/\mathbf{y}, \mathbf{d}'/\mathbf{m}', \mathbf{e}'/\mathbf{y}'] \wedge (\mathbf{m}, \mathbf{y}) = (\mathbf{d}, \mathbf{e}) \wedge (\mathbf{m}', \mathbf{y}') = (\mathbf{d}', \mathbf{e}')\big)$$

Observing that input changes are unconstrained when transiting from quiescent states and disallowed when transiting from transient states, the $\gamma[\mathbf{d}/\mathbf{m}, \ldots]$ can be assumed to have free variables in I only. This induces another representation of the transition relation in first order form.

$$\mathcal{R}_4 \equiv \bigvee_{i=0}^{a_0} \big(\alpha_i \wedge (\mathbf{m}, \mathbf{y}) = (\mathbf{d}_i, \mathbf{e}_i) \wedge (\mathbf{m}', \mathbf{y}') = (\mathbf{m}, \mathbf{y})\big) \vee \bigvee_{i=a_0+1}^{a} \big(\beta_i \wedge (\mathbf{m}, \mathbf{y}) = (\mathbf{d}_i, \mathbf{e}_i) \wedge (\mathbf{m}', \mathbf{y}') = (\mathbf{d}'_i, \mathbf{e}'_i) \wedge \mathbf{x}' = \mathbf{x}\big)$$

where propositions α_i and β_i are disjunctions of input constraints. The number a_0 enumerates the pairs $(\mathbf{d}_i, \mathbf{e}_i)$ of internal state and output values. The propositions α_i denote quiescent state conditions, sorted over the possible

$(\mathbf{d}_i, \boldsymbol{e}_i)$-values. Each β_i specifies a condition for a transient state s_1 satisfying $\beta_i[s_1(\mathbf{x})/\mathbf{x}]$, $s_1(\mathbf{m}) = \mathbf{d}_i$ and $s_1(\mathbf{y}) = \boldsymbol{e}_i$ to switch to a quiescent post state s_2 satisfying $s_2 = \{\mathbf{x} \mapsto s_1(\mathbf{x}), \mathbf{m} \mapsto \mathbf{d}'_i, \mathbf{y} \mapsto \boldsymbol{e}'_i\}$.

Example 13. The table specifying φ_i, ψ_i in Example 11 is already structured in such a way that the propositions α_i, β_i required for the representation $\mathcal{R}_4$ can be directly identified:

α_0	$x < \max$
α_1	$x = \max$
α_2	$x > \max - \delta$

β_3	$x = \max$
β_4	$x = \max$
β_5	$x > \max$
β_6	$x \leq \max - \delta$
β_7	$x > \max$
β_8	$x < \max$

□

7.7.5 Final RIOSTS Transition Relation

Evaluating $\mathcal{R}_4$ by means of a reachability analysis, the final proposition $\mathcal{R}$ conforming to an RIOSTS transition relation is produced as follows. A *reachability trace* is a finite sequence of indexes $i_0.i_1 \dots i_n,\ n > 1$, such that

1. $i_0, i_n \in \{1, \dots, a_0\}$ and $\{i_1, \dots, i_{n-1}\} \subseteq \{a_0 + 1, \dots, a\}$ (recall that a_0, a are indexes denoting the last elements of disjunctions in the definition of $\mathcal{R}_4$).
2. $\mathbf{d}_{i_0} = \mathbf{d}_{i_1} \wedge \boldsymbol{e}_{i_0} = \boldsymbol{e}_{i_1}$.
3. $\mathbf{d}'_{i_j} = \mathbf{d}_{i_{j+1}} \wedge \boldsymbol{e}'_{i_j} = \boldsymbol{e}_{i_{j+1}}$ for $j = 1, \dots, n-1$.
4. The formula
$$\bigwedge_{j=1}^{n-1} \beta_{i_j} \wedge \alpha_{i_n}$$
is satisfiable, that is, there exists $\mathbf{c} \in D_I$ such that
$$\bigwedge_{j=1}^{n-1} \beta_{i_j}[\mathbf{c}/\mathbf{x}] \wedge \alpha_{i_n}[\mathbf{c}/\mathbf{x}].$$

A reachability trace captures a set of state traces starting in any quiescent state s_0 satisfying $s_0(\mathbf{m}) = \mathbf{d}_{i_0}$, $s_0(\mathbf{y}) = \boldsymbol{e}_{i_0}$ and ending in a quiescent state s_n satisfying $s_n(\mathbf{m}) = \mathbf{d}_{i_n}$, $s_n(\mathbf{y}) = \boldsymbol{e}_{i_n}$, such that $s_1, \ldots, s_{n-1}$ are transient. The trace conforms to the transition relation: the input defined when transiting from s_0 to s_n is never changed until s_n is reached, and the changes produced by the transition from transient state s_i are consistent with the valuation in state s_{i+1}. Since a_0, a are finite, the number of reachability traces is always finite. Note that this fact holds for both deterministic and nondeterministic state transition systems. Consequently, the sets RTR_{i_0,i_n} of reachability traces starting in $i_0 \in \{1, \ldots, a_0\}$ and ending in $i_n \in \{1, \ldots, a_0\}$, and the sets RTR_{i_0} of reachability traces starting in $i_0 \in \{1, \ldots, a_0\}$ and ending at any i_n are finite as well.

Note that – since we are dealing with potentially infinite input domains – it may be the case that two quiescent states in the original test model are connected by an infinite number of different path segments containing transient states only. Each segment, however, will be finite (that is, a trace segment), because the model is assumed to be livelock free. Each of these traces will cover exactly one finite sequence of state classes which is identified by a reachability trace $i_0 \ldots i_n$. Therefore the potentially infinite number of finite traces connecting to quiescent states by transient states can be abstracted by the finite number of associated reachability traces.

Starting from the initial state s_0, the reachable quiescent states are explored according to the following algorithm.

1. **Inputs.** State valuation $s_0 : V \to D$ that associates an initial value to every model variable. Constants $a_0, a \in \mathbb{N}$, propositions $\alpha_1, \ldots, \alpha_{a_0}$ and $\beta_{a_0+1}, \ldots, \beta_a$, as well as associated value vectors $(\mathbf{d}_i, \boldsymbol{e}_i), i = 1, \ldots, a_0, a_0 + 1, \ldots, a$ and $(\mathbf{d}'_i, \boldsymbol{e}'_i), i = a_0 + 1, \ldots, a$, extracted from the transition relation in first order representation $\mathcal{R}_4$, as specified in Section 7.7.4.

2. **Outputs.** Transition relation $\mathcal{R}$ in first order form, conforming to the RIOSTS requirement that transient states are always followed by quiescent states, while quiescent states may transit to other quiescent or to transient states.

3. Initialise $\mathcal{R}$ with
$$\mathcal{R} \equiv \bigvee_{i=1}^{a_0} \left(\alpha_i \wedge (\mathbf{m}, \mathbf{y}) = (\mathbf{d}_i, \boldsymbol{e}_i) \wedge (\mathbf{m}', \mathbf{y}') = (\mathbf{d}_i, \boldsymbol{e}_i) \right)$$

4. Let $i_0 \in \{1, \ldots, a_0\}$ be the uniquely determined index, such that the initial state s_0 fulfils

$$s_0(\alpha_{i_0}) \wedge s_0(\mathbf{m}) = \mathbf{d}_{i_0} \wedge s_0(\mathbf{y}) = \boldsymbol{e}_{i_0}$$

5. Initialise an index queue T with this index i_0.

6. Initialise index sets IDX $:= \varnothing$ and $J := \varnothing$.

7. While T is not empty

   (a) Set $i = \text{head}(T)$ and remove head from T.

   (b) Set IDX $:=$ IDX $\cup \{i\}$.

   (c) Calculate RTR_i as the set of all reachability traces starting with index i. (From the discussion above we know that RTR_i is finite and can be automatically calculated by checking the satisfiability of formulas of the type $\bigwedge_{j=1}^{n-1} \beta_{i_j} \wedge \alpha_{i_n}$.)

   (d) For every reachability trace $i.i_1 \ldots i_n \in \text{RTR}_i$

      i. Set

$$g_{i.i_1 \ldots i_n} := \Big(\bigwedge_{j=1}^{n-1} \beta_{i_j} \wedge \alpha_{i_n} \Big)$$

      ii. Extend $\mathcal{R}$ by setting

$$\mathcal{R} := \mathcal{R} \vee \big(g_{i.i_1 \ldots i_n} \wedge (\mathbf{m}, \mathbf{y}) = (\mathbf{d}_i, \boldsymbol{e}_i) \wedge (\mathbf{m}', \mathbf{y}') = (\mathbf{d}_{i_n}, \boldsymbol{e}_{i_n}) \big).$$

      iii. Insert (i, i_n) into J.

      iv. If $i_n \notin$ IDX
         - Insert i_n into IDX.
         - Append i_n to T.

   (e) Continue with Step 7.

8. For all $i \in \{1, \ldots, a_0\} -$ IDX remove disjunct i from $\mathcal{R}$, because the associated quiescent pre-states are unreachable.

9. For each $(i, j) \in J$, collect all disjuncts

$$g_{i'.i'_1 \ldots i'_{n'}} \wedge (\mathbf{m}, \mathbf{y}) = (\mathbf{d}_{i'}, \boldsymbol{e}_{i'}) \wedge (\mathbf{m}', \mathbf{y}') = (\mathbf{d}_{i'_{n'}}, \boldsymbol{e}_{i'_{n'}})$$

satisfying $i'.i'_1 \ldots i'_{n'} \in \mathrm{RTR}_{i,j}$ and consequently $i' = i$, $i'_{n'} = j$ and merge them into a single disjunct

$$g_{i,j} \wedge (\mathbf{m}, \mathbf{y}) = (\mathbf{d}_i, \boldsymbol{e}_i) \wedge (\mathbf{m}', \mathbf{y}') = (\mathbf{d}_j, \boldsymbol{e}_j)$$

where

$$g_{i,j} \equiv \bigvee_{i'.i'_1 \ldots i'_{n'} \in \mathrm{RTR}_{i,j}} g_{i'.i'_1 \ldots i'_{n'}}$$

10. Terminate by returning $\mathcal{R}$.

Observe that the algorithm always terminates because the number of reachability traces is finite, and therefore the nested loop in Step 7 terminates. Furthermore, since $\mathcal{R}$ has been constructed using all reachability traces induced by $\mathcal{R}_4$, the set of $\mathcal{R}_4$-state traces is I/O-equivalent to the set of $\mathcal{R}$-state traces (recall that I/O-equivalence abstracts from intermediate transient states and only compares quiescent ones). Furthermore, since the transformations from $\mathcal{R}_1$ to $\mathcal{R}_4$ are all equivalence transformations, this proves that the set of $\mathcal{R}_1$-state traces in the original model is I/O-equivalent to the set of $\mathcal{R}$-state traces. This establishes termination and correctness of the algorithm.

The resulting proposition

$$\mathcal{R} \equiv \mathcal{R}_1 \vee \mathcal{R}_2 \tag{7.4}$$

$$\mathcal{R}_1 \equiv \bigvee_{i \in \mathrm{IDX}} \big(\alpha_i \wedge (\mathbf{m}, \mathbf{y}) = (\mathbf{d}_i, \boldsymbol{e}_i) \wedge (\mathbf{m}', \mathbf{y}') = (\mathbf{d}_i, \boldsymbol{e}_i)\big) \tag{7.5}$$

$$\mathcal{R}_2 \equiv \bigvee_{(i,j) \in J} \big(g_{i,j} \wedge (\mathbf{m}, \mathbf{y}) = (\mathbf{d}_i, \boldsymbol{e}_i) \wedge (\mathbf{m}', \mathbf{y}') = (\mathbf{d}_j, \boldsymbol{e}_j)\big) \tag{7.6}$$

completely specifies the RIOSTS representation we are looking for. The set of reachable quiescent states s is specified by

$$S_Q = \big\{ s : V \to D \mid s\big(\bigvee_{i \in \mathrm{IDX}} (\alpha_i \wedge (\mathbf{m}, \mathbf{y}) = (\mathbf{d}_i, \boldsymbol{e}_i))\big)\big\}$$

The set of transient states is specified by

$$S_T = \big\{ s : V \to D \mid s\big(\bigvee_{(i,j) \in J} (g_{i,j} \wedge (\mathbf{m}, \mathbf{y}) = (\mathbf{d}_i, \boldsymbol{e}_i))\big)\big\}$$

The transition relation $R \subseteq S \times S$ is given by

$$R = \{(s_1, s_2) \mid \mathcal{R}[s_1(\boldsymbol{v})/\boldsymbol{v}, s_2(\boldsymbol{v})/\boldsymbol{v}']\}$$

Example 14. For the alarm indication system, Example 13 shows that there is not much to do in the algorithm above, because the transition relation $\mathcal{R}_4$ already has the required target form required for the final proposition $\mathcal{R}$: every transient state specified by $\beta_i \wedge (m, y) = (d_i, e_i)$ has one or two immediate quiescent post-states. Therefore it is not necessary to calculate reachability traces. In [31], a more complex example is presented where reachability traces need to be constructed. For our example here, $\mathcal{R}$ is constructed from $\mathcal{R}_4$ by setting

$$\begin{aligned}\mathcal{R} \;\equiv\; & \bigvee_{i=0}^{2} g_{i,i} \wedge (m, y) = (d_i, e_i) \wedge (m', y') = (m, y) \vee \\ & \bigvee_{(i,j)\in J} g_{i,j} \wedge (m, y) = (d_i, e_i) \wedge (m', y') = (d_j, e_j)\end{aligned}$$

with

$$J = \{(0,1), (0,2), (1,0), (1,2), (2,0)\}$$

and

$$(d_0, e_0) = (0, \mathrm{OK}), (d_1, e_1) = (1, \mathrm{OK}), (d_2, e_2) = (2, \mathrm{ALARM}),$$

and

$$\begin{aligned} g_{0,0} &\equiv x < \max \\ g_{1,1} &\equiv x = \max \\ g_{2,2} &\equiv x > \max - \delta \end{aligned}$$

$$\begin{aligned} g_{0,1} &\equiv x = \max \\ g_{0,2} &\equiv x \geq \max \\ g_{1,0} &\equiv x < \max \\ g_{1,2} &\equiv x > \max \\ g_{2,0} &\equiv x \leq \max - \delta \end{aligned}$$

□

7.7.6 IECP Identification

From the discussion in Section 7.4 we know that the propositions α_i induce sets

$$A_i = \{s \in S \mid s(\alpha_i) \wedge s(\boldsymbol{m}) = \boldsymbol{d}_i \wedge s(\boldsymbol{y}) = \boldsymbol{e}_i\}, \quad i \in \mathrm{IDX}$$

of I/O-equivalent quiescent states. Setting $g_{i,i} \equiv \alpha_i$ for $i \in IDX$, we observe that all input changes $s \oplus \{\boldsymbol{x} \mapsto \boldsymbol{c}\}$ satisfying $g_{i,i}[\boldsymbol{c}/\boldsymbol{x}]$ lead to post-states in A_i.

By construction, all input changes $s \oplus \{\boldsymbol{x} \mapsto \boldsymbol{c}\}$ applied to states s in A_i satisfying $g_{i,j}[\boldsymbol{c}/\boldsymbol{x}]$ with $(i,j) \in J$ (J defined in the algorithm in Section 7.7.5 and used in Equation (7.6)) may lead to post-states in A_j. If the RIOSTS is deterministic, then this is the only possibility. For nondeterministic RIOSTS, however, several conditions $g_{i,j_1}[\boldsymbol{c}/\boldsymbol{x}]$, $g_{i,j_2}[\boldsymbol{c}/\boldsymbol{x}], j_1 \neq j_2$ may evaluate to `true`. Then, as the structure of $\mathcal{R}$ in Equation (7.6) shows, the transient state $s \oplus \{\boldsymbol{x} \mapsto \boldsymbol{c}\}$ may transit nondeterministically to one of the post-states

$$\{\boldsymbol{x} \mapsto \boldsymbol{c}, \boldsymbol{m} \mapsto \boldsymbol{d}_{j_1}, \boldsymbol{y} \mapsto \boldsymbol{e}_{j_1}\} \quad \text{or} \quad \{\boldsymbol{x} \mapsto \boldsymbol{c}, \boldsymbol{m} \mapsto \boldsymbol{d}_{j_2}, \boldsymbol{y} \mapsto \boldsymbol{e}_{j_2}\}.$$

This possibility is, of course, not restricted to two possible post-states, but it is always restricted to a finite number of choices, because the number $|J|$ of reachable pairs $(\boldsymbol{d}_j, \boldsymbol{e}_j)$ of internal state values and output values is finite. To analyse these finite variants of nondeterministic choices further, consider a fixed index $i \in IDX$ and set $J_i = \{i\} \cup \{j \in IDX \mid (i,j) \in J\}$. Let $K_{\neq\varnothing} \subseteq J_i$ and define

$$\Phi_{i,K} \equiv \bigwedge_{j \in K} g_{i,j} \wedge \bigwedge_{j \in J_i \setminus K} \neg g_{i,j}$$

and

$$K_i = \{K_{\neq\varnothing} \subseteq J_i \mid \Phi_{i,K} \text{ is satisfiable}\}.$$

For each function $f : IDX \to 2^{IDX}$ where the proposition

$$\Phi_f \equiv \bigwedge_{i \in IDX} \Phi_{i,f(i)}$$

is satisfiable, the following property holds:

$$\forall i \in IDX, \boldsymbol{c} \in D_I : \big((s \in A_i \wedge \Phi_f[\boldsymbol{c}/\boldsymbol{x}]) \Rightarrow (\forall j \in f(i) : s/\boldsymbol{c} \cap A_j \neq \varnothing)\big)$$

This means that any input change $\boldsymbol{c}$ satisfying Φ_f, when applied to any $s \in A_i$ leads to a quiescent post-state in some A_j with $j \in f(i)$. Moreover, for any $j' \in f(i)$, a transition from s to an element of $A_{j'}$ is possible: if $|f(i)| > 1$, this reflects the presence of nondeterministic transitions.

The following lemma shows that the Φ_f introduce a finite partitioning of D_I, because every $\boldsymbol{c} \in D_I$ fulfils exactly one Φ_f.

Lemma 7.7 *The set*

$$\mathcal{I} = \{\mathsf{X}_\mathsf{f} \mid \Phi_\mathsf{f} \textit{ is satisfiable and } \mathsf{X}_\mathsf{f} = \{\mathbf{c} \in \mathsf{D}_\mathrm{I} \mid \Phi_\mathsf{f}[\mathbf{c}/\boldsymbol{\chi}]\}\}$$

is an IECP according to Definition 7.1 or a refinement thereof.

Proof. For any $\mathtt{i} \in \mathrm{IDX}$, every input vector $\mathbf{c}$ fulfils the disjunction $\bigvee_{\mathsf{j}\in\mathsf{J}_\mathtt{i}} \mathsf{g}_{\mathtt{i},\mathsf{j}}[\mathbf{c}/\boldsymbol{\chi}]$, because our RIOSTS are completely specified, that is,

$$\Big\{\mathbf{c} \in \mathsf{D}_\mathrm{I} \mid \bigvee_{\mathsf{j}\in\mathsf{J}_\mathtt{i}} \mathsf{g}_{\mathtt{i},\mathsf{j}}[\mathbf{c}/\boldsymbol{\chi}]\Big\} = \mathsf{D}_\mathrm{I}. \tag{7.7}$$

For arbitrary $\mathbf{c} \in \mathsf{D}_\mathrm{I}$, define $\mathsf{C}_\mathtt{i}(\mathbf{c}) = \{\mathsf{j} \in \mathsf{J}_\mathtt{i} \mid \mathsf{g}_{\mathtt{i},\mathsf{j}}[\mathbf{c}/\boldsymbol{\chi}]\}$. Then $\mathsf{C}_\mathtt{i}(\mathbf{c})$ is never empty according to Equation (7.7), and

$$\bigwedge_{\mathsf{j}\in\mathsf{C}_\mathtt{i}(\mathbf{c})} \mathsf{g}_{\mathtt{i},\mathsf{j}}[\mathbf{c}/\boldsymbol{\chi}] \wedge \bigwedge_{\mathsf{j}\in\mathsf{J}_\mathtt{i}\setminus\mathsf{C}_\mathtt{i}(\mathbf{c})} \neg\mathsf{g}_{\mathtt{i},\mathsf{j}}[\mathbf{c}/\boldsymbol{\chi}]$$

always evaluates to `true`. Let $\sim_\mathsf{C}$ be a binary relation on D_I, where

$$\mathbf{c}' \sim_\mathsf{C} \mathbf{c} \quad \text{if and only if} \quad \forall \mathtt{i} \in \mathrm{IDX} : \mathsf{C}_\mathtt{i}(\mathbf{c}') = \mathsf{C}_\mathtt{i}(\mathbf{c}).$$

Obviously, $\sim_\mathsf{C}$ is an equivalence relation, IDX and $\{\mathsf{C}_\mathtt{i}(\mathbf{c}) \mid \mathbf{c} \in \mathsf{D}_\mathrm{I}\} \subseteq 2^{\mathsf{J}_\mathtt{i}}$ are finite sets. Hence $\mathsf{D}_\mathrm{I}/_{\sim_\mathsf{C}}$ is a finite partitioning of D_I.

For any $\mathtt{i} \in \mathrm{IDX}$ and $\mathsf{K} \subseteq \mathsf{J}_\mathtt{i}$,

$$\Phi_{\mathtt{i},\mathsf{K}} \equiv \bigwedge_{\mathsf{j}\in\mathsf{K}} \mathsf{g}_{\mathtt{i},\mathsf{j}} \wedge \bigwedge_{\mathsf{j}\in\mathsf{J}_\mathtt{i}\setminus\mathsf{K}} \neg\mathsf{g}_{\mathtt{i},\mathsf{j}}$$

is satisfiable if and only if there is $\mathbf{c} \in \mathsf{D}_\mathrm{I}$, $\Phi_{\mathtt{i},\mathsf{K}}[\mathbf{c}/\boldsymbol{\chi}]$, i.e., $\mathsf{K} = \mathsf{C}_\mathtt{i}(\mathbf{c})$.

For each function $\mathsf{f} : \mathrm{IDX} \to 2^{\mathrm{IDX}}$, the proposition

$$\Phi_\mathsf{f} \equiv \bigwedge_{\mathtt{i}\in\mathrm{IDX}} \Phi_{\mathtt{i},\mathsf{f}(\mathtt{i})}$$

is satisfiable, if and only if there exists $\mathbf{c} \in \mathsf{D}_\mathrm{I}$, such that $\Phi_{\mathtt{i},\mathsf{f}(\mathtt{i})}[\mathbf{c}/\boldsymbol{\chi}]$ holds for all $\mathtt{i} \in \mathrm{IDX}$. This means that $\mathsf{f}(\mathtt{i}) = \mathsf{C}_\mathtt{i}(\mathbf{c})$ holds for all $\mathtt{i} \in \mathrm{IDX}$.

For any $\mathbf{c} \in \mathsf{D}_\mathrm{I}$, let $\mathsf{f}_\mathbf{c} : \mathrm{IDX} \to 2^{\mathrm{IDX}}$, $\mathsf{f}_\mathbf{c}(\mathtt{i}) = \mathsf{C}_\mathtt{i}(\mathbf{c})$. Then for any $\mathsf{f} : \mathrm{IDX} \to 2^{\mathrm{IDX}}$, the proposition Φ_f is satisfiable if and only if $\mathsf{f} = \mathsf{f}_\mathbf{c}$, for

some $\mathbf{c} \in \mathrm{D_I}$. Hence

$$\begin{aligned}
\mathrm{D_I}/_{\sim_C} &= \bigcup_{\mathbf{c}\in \mathrm{D_I}} \{\mathbf{c}' \in \mathrm{D_I} \mid \mathrm{C_i}(\mathbf{c}') = \mathrm{C_i}(\mathbf{c}), \forall \mathfrak{i} \in \mathrm{IDX}\} \\
&= \bigcup_{\mathbf{c}\in \mathrm{D_I}} \{\mathbf{c}' \mid \Phi_{\mathrm{f_c}}[\mathbf{c}'/\mathbf{x}]\} \\
&= \bigcup_{\Phi_\mathrm{f} \text{ is satisfiable}} \{\mathbf{c}' \mid \Phi_\mathrm{f}[\mathbf{c}'/\mathbf{x}]\}
\end{aligned}$$

□

Example 15. With $\mathcal{R}$ obtained for the alarm indication system in Example 14 the input equivalence classes are calculated as follows:

$$\mathrm{J_0} = \{0, 1, 2\},\ \mathrm{J_1} = \{0, 1, 2\},\ \mathrm{J_2} = \{0, 2\}$$

and

$$\mathrm{K_0} = \{\{0\}, \{2\}, \{1, 2\}\},\ \mathrm{K_1} = \{\{0\}, \{1\}, \{2\}\},\ \mathrm{K_2} = \{\{0\}, \{2\}\}$$

$$\begin{aligned}
\Phi_{0,\{0\}} &\equiv x < \max & \Phi_{1,\{1\}} &\equiv x = \max \\
\Phi_{0,\{2\}} &\equiv x > \max & \Phi_{1,\{2\}} &\equiv x > \max \\
\Phi_{0,\{1,2\}} &\equiv x = \max & \Phi_{2,\{0\}} &\equiv x \leq \max - \delta \\
\Phi_{1,\{0\}} &\equiv x < \max & \Phi_{2,\{2\}} &\equiv x > \max - \delta
\end{aligned}$$

The conjunctions Φ_f that have a solution are constructed by enumerating all possible mappings $\mathrm{f} : \{0, 1, 2\} \to 2^{\{0,1,2\}}$. The following conjunctions are feasible. In these definitions the notation $(\mathrm{k_0}, \mathrm{k_1}, \mathrm{k_2})$, $\mathrm{k_i} \in \mathrm{K_i}$, denotes the function $\mathrm{f} = \{0 \mapsto \mathrm{k_0}, 1 \mapsto \mathrm{k_1}, 2 \mapsto \mathrm{k_2}\}$

$$\begin{aligned}
\Phi_{(\{0\},\{0\},\{0\})} &\equiv x < \max \wedge x \leq \max - \delta \equiv x \in [0, \max - \delta] \\
\Phi_{(\{0\},\{0\},\{2\})} &\equiv x < \max \wedge x > \max - \delta \equiv x \in (\max - \delta, \max) \\
\Phi_{(\{1,2\}\{1\}\{2\})} &\equiv x = \max \\
\Phi_{(\{2\}\{2\}\{2\})} &\equiv x > \max
\end{aligned}$$

This induces the input equivalence classes

$$\mathcal{I} = \{\ [0, \max - \delta], (\max - \delta, \max), [\max, \max], (\max, \infty)\ \}.$$

□

Chapter 8

Test Case Map – From FSM Test Cases to RIOSTS Test Cases

8.1 RIOSTS Test Cases

In analogy to the introduction of abstract FSM test cases in Section 4.1, we introduce RIOSTS test cases as subsets of the I/O-languages involved.

Definition 8.1 *Given an RIOSTS signature* $(\mathsf{I}, \mathsf{O}, \mathsf{D}_\mathsf{I}, \mathsf{D}_\mathsf{O})$*, a* test case W (for signature $(\mathsf{I}, \mathsf{O}, \mathsf{D}_\mathsf{I}, \mathsf{D}_\mathsf{O})$) *is a pair of disjoint sets*

$$W = (W_{\text{pass}}, W_{\text{fail}}), \quad W_{\text{pass}} \cup W_{\text{fail}} \subseteq (\Sigma_\mathsf{I} \times \Sigma_\mathsf{O})^*.$$

For an RIOSTS $\mathcal{S}$ *and a test case* $W = (W_{\text{pass}}, W_{\text{fail}})$*, the following pass-criteria are defined.*

1. $\mathcal{S}$ $\underline{\text{pass}}_{\preceq}$ $W \equiv \mathsf{L}(\mathcal{S}) \cap W_{\text{fail}} = \varnothing$.
2. $\mathcal{S}$ $\underline{\text{pass}}_{\sim}$ $W \equiv W_{\text{pass}} \subseteq \mathsf{L}(\mathcal{S}) \wedge \mathsf{L}(\mathcal{S}) \cap W_{\text{fail}} = \varnothing$.

The set of all RIOSTS test cases for the given signature is denoted by $TC(\mathsf{I}, \mathsf{O}, \mathsf{D}_\mathsf{I}, \mathsf{D}_\mathsf{O})$. □

Since Theorem 4.1 only refers to the existence of a language associated with each model, and to the definitions of pass-traces and fail-traces, this theorem holds in analogous form for RIOSTS:

Theorem 8.1 *Let $\mathcal{S}_1, \mathcal{S}_2$ be any RIOSTS with signature $(\mathrm{I}, \mathrm{O}, \mathrm{D}_\mathrm{I}, \mathrm{D}_\mathrm{O})$. Then*

1. *$\mathcal{S}_2 \preceq \mathcal{S}_1$ if and only if*

$$\forall W \in TC(\mathrm{I}, \mathrm{O}, \mathrm{D}_\mathrm{I}, \mathrm{D}_\mathrm{O}) : \mathcal{S}_1 \underline{\mathrm{pass}}_{\preceq} W \Rightarrow \mathcal{S}_2 \underline{\mathrm{pass}}_{\preceq} W$$

2. *$\mathcal{S}_2 \sim \mathcal{S}_1$ if and only if*

$$\forall W \in TC(\mathrm{I}, \mathrm{O}, \mathrm{D}_\mathrm{I}, \mathrm{D}_\mathrm{O}) : \mathcal{S}_1 \underline{\mathrm{pass}}_{\sim} W \Rightarrow \mathcal{S}_2 \underline{\mathrm{pass}}_{\sim} W$$

□

8.2 The Test Case Map

Given an RIOSTS signature $\mathtt{Sig} = (\mathrm{I}, \mathrm{O}, \mathrm{D}_\mathrm{I}, \mathrm{D}_\mathrm{O})$ and a sub-domain $\mathcal{D}(\mathtt{Sig}, \mathcal{I})$ as introduced in Section 7.6, this associates a signature $\mathrm{FSM}(\mathcal{I}, \mathrm{D}_\mathrm{O})$ of FSMs and FSM test cases $\mathrm{TC}(\mathcal{I}, \mathrm{D}_\mathrm{O})$. We will now construct the *test case map*

$$\mathsf{T}^* : \mathrm{TC}(\mathcal{I}, \mathrm{D}_\mathrm{O}) \to \mathrm{TC}(\mathrm{I}, \mathrm{O}, \mathrm{D}_\mathrm{I}, \mathrm{D}_\mathrm{O})$$

which translates FSM test cases to RIOSTS test cases.

Let $W = (W_{\mathrm{pass}}, W_{\mathrm{fail}}) \in \mathrm{TC}(\mathrm{I}, \mathrm{O}, \mathrm{D}_\mathrm{I}, \mathrm{D}_\mathrm{O})$. $W = (W_{\mathrm{pass}}, W_{\mathrm{fail}})$ is called a *representative* of $\mathsf{U} = (\mathsf{U}_{\mathrm{pass}}, \mathsf{U}_{\mathrm{fail}}) \in \mathrm{TC}(\mathcal{I}, \mathrm{D}_\mathrm{O})$, if there exists an injective mapping $\mathsf{f}_\mathsf{U} : \mathsf{U}_{\mathrm{pass}} \cup \mathsf{U}_{\mathrm{fail}} \to (\mathrm{D}_\mathrm{I} \times \mathrm{D}_\mathrm{O})^*$ satisfying:

1. $W = (\mathsf{f}_\mathsf{U}(\mathsf{U}_{\mathrm{pass}}), \mathsf{f}_\mathsf{U}(\mathsf{U}_{\mathrm{fail}}))$.[1]
2. $\mathsf{f}_\mathsf{U}(\varepsilon) = \varepsilon$.
3. $\forall \overline{\mathbf{X}}/\overline{\mathbf{y}} \in \mathsf{U}_{\mathrm{pass}} \cup \mathsf{U}_{\mathrm{fail}} : \mathsf{f}_\mathsf{U}(\overline{\mathbf{X}}/\overline{\mathbf{y}}) = \overline{\mathbf{x}}/\overline{\mathbf{y}}$ with some $\overline{\mathbf{x}} \in \mathrm{D}_\mathrm{I}^*$ satisfying $\overline{\mathbf{x}}\, \mathrm{I}\, \overline{\mathbf{X}}$.

Condition 3 to be satisfied by admissible mappings f_U states that every concrete input x_i in an input/output trace $\overline{\mathbf{x}}/\overline{\mathbf{y}}$ is a representative of the corresponding input equivalence class X_i in the FSM test trace $\overline{\mathbf{X}}/\overline{\mathbf{y}}$.

Now we can define the test case map

[1] We apply the usual notation for lifting functions $\mathsf{f} : \mathsf{A} \to \mathsf{B}$ to their set-valued equivalents: for $\mathsf{Z} \subseteq \mathsf{A}$, $\mathsf{f}(\mathsf{Z}) = \{\mathsf{f}(\mathsf{a}) \mid \mathsf{a} \in \mathsf{Z}\}$.

$$\mathsf{T}^* : \mathrm{TC}(\mathcal{I}, \mathsf{D_O}) \to \mathrm{TC}(\mathsf{I}, \mathsf{O}, \mathsf{D_I}, \mathsf{D_O});$$
$$\mathsf{U} = (\mathsf{U}_{\mathrm{pass}}, \mathsf{U}_{\mathrm{fail}}) \mapsto \mathsf{W} = (\mathsf{f_U}(\mathsf{U}_{\mathrm{pass}}), \mathsf{f_U}(\mathsf{U}_{\mathrm{fail}}))$$

as a function mapping each FSM test case $\mathsf{U} \in \mathrm{TC}(\mathcal{I}, \mathsf{D_O})$ to one of its representatives $\mathsf{W} \in \mathrm{TC}(\mathsf{I}, \mathsf{O}, \mathsf{D_I}, \mathsf{D_O})$. Different selections of representatives lead to different variants of T^*, but the proof of the satisfaction condition given below will show that SC2 holds independently on the choice of representatives, as long as the selected $\mathsf{f_U}$ all fulfil the conditions specified above.

Chapter 9

Proof of the Satisfaction Condition SC2

With both model map and test case map available, we are now in the position to prove the second part of the satisfaction condition.

Given an RIOSTS signature $\mathsf{Sig} = (\mathsf{I}, \mathsf{O}, \mathsf{D_I}, \mathsf{D_O})$ and a sub-domain $\mathcal{D}(\mathsf{Sig}, \mathcal{I})$, we have seen in Section 7.6 that this induces a well-defined model map $\mathsf{T} : \mathcal{D}(\mathsf{Sig}, \mathcal{I}) \to \mathrm{FSM}(\mathcal{I}, \mathsf{D_O})$ from RIOSTS models to FSMs. Moreover, from Section 8 it is known that a test case map $\mathsf{T}^* : \mathrm{TC}(\mathcal{I}, \mathsf{D_O}) \to \mathrm{TC}(\mathsf{I}, \mathsf{O}, \mathsf{D_I}, \mathsf{D_O})$ can be constructed by mapping each FSM test case $\mathsf{U} = (\mathsf{U_{pass}}, \mathsf{U_{fail}})$ to a representative $\mathsf{T}^*(\mathsf{U}) = \mathsf{W} = (\mathsf{f_U}(\mathsf{U_{pass}}), \mathsf{f_U}(\mathsf{U_{fail}}))$, where $\mathsf{f_U}$ maps I/O-traces of U to I/O-traces of W and satisfies the condition specified in Section 8.2. It will now be shown that T and T^* fulfil the second part **SC2** of the satisfaction condition introduced in Section 2.6.

Theorem 9.1 (Satisfaction Condition SC2) *With the notation introduced in the previous sections, the following statements hold for all* $\mathsf{U} = (\mathsf{U_{pass}}, \mathsf{U_{fail}}) \in TC(\mathcal{I}, \Sigma_\mathsf{O})$ *and all* $\mathcal{S} \in \mathcal{D}(\mathsf{Sig}, \mathcal{I})$.

1. $\mathsf{T}(\mathcal{S}) \ \underline{\mathrm{pass}}_{\preceq} \ \mathsf{U} \Leftrightarrow \mathcal{S} \ \underline{\mathrm{pass}}_{\preceq} \ \mathsf{T}^*(\mathsf{U})$
2. $\mathsf{T}(\mathcal{S}) \ \underline{\mathrm{pass}}_{\sim} \ \mathsf{U} \Leftrightarrow \mathcal{S} \ \underline{\mathrm{pass}}_{\sim} \ \mathsf{T}^*(\mathsf{U})$

Proof. Since $\mathsf{W} = \mathsf{T}^*(\mathsf{U}) = (\mathsf{f_U}(\mathsf{U_{pass}}), \mathsf{f_U}(\mathsf{U_{fail}}))$ is the representative of U with $\mathsf{f_U}$ satisfying the conditions specified in Section 8.2, application of Theorem 7.1 results in

$$\mathsf{f_U}(\overline{\mathbf{X}}/\overline{\mathbf{y}}) \in \mathsf{L}(\mathcal{S}) \Leftrightarrow \overline{\mathbf{X}}/\overline{\mathbf{y}} \in \mathsf{L}(\mathsf{T}(\mathcal{S})) \tag{9.1}$$

Hence

$$
\begin{aligned}
\mathsf{L}(\mathcal{S}) \cap (W_{\text{pass}} \cup W_{\text{fail}}) &= [\text{Definition of } W] \\
&\quad \mathsf{L}(\mathcal{S}) \cap \mathsf{f_U}(\mathsf{U}_{\text{pass}} \cup \mathsf{U}_{\text{fail}}) \\
&= \{\mathsf{f_U}(\overline{\mathbf{X}}/\overline{\mathbf{y}}) \in \mathsf{L}(\mathcal{S}) \mid \overline{\mathbf{X}}/\overline{\mathbf{y}} \in \mathsf{U}_{\text{pass}} \cup \mathsf{U}_{\text{fail}}\} \\
&= [\text{Equivalence (9.1)}] \\
&\quad \{\mathsf{f_U}(\overline{\mathbf{X}}/\overline{\mathbf{y}}) \mid \overline{\mathbf{X}}/\overline{\mathbf{y}} \in \mathsf{L}(\mathsf{T}(\mathcal{S})) \wedge \overline{\mathbf{X}}/\overline{\mathbf{y}} \in \mathsf{U}_{\text{pass}} \cup \mathsf{U}_{\text{fail}}\} \\
&= \mathsf{f_U}\big(\mathsf{L}(\mathsf{T}(\mathcal{S})) \cap (\mathsf{U}_{\text{pass}} \cup \mathsf{U}_{\text{fail}})\big) \qquad (*)
\end{aligned}
$$

Consequently,

$$
\begin{aligned}
\mathcal{S} \underline{\text{pass}}_{\preceq} \mathsf{T}^*(\mathsf{U}) &\Leftrightarrow [\mathsf{T}^*(\mathsf{U}) = W] \\
&\quad \mathsf{L}(\mathcal{S}) \cap (W_{\text{pass}} \cup W_{\text{fail}}) \subseteq W_{\text{pass}} \\
&\Leftrightarrow [\text{Derivation } (*), W_{\text{pass}} = \mathsf{f_U}(\mathsf{U}_{\text{pass}})] \\
&\quad \mathsf{f_U}\big(\mathsf{L}(\mathsf{T}(\mathcal{S})) \cap (\mathsf{U}_{\text{pass}} \cup \mathsf{U}_{\text{fail}})\big) \subseteq \mathsf{f_U}(\mathsf{U}_{\text{pass}}) \\
&\Leftrightarrow [\mathsf{f_U} \text{ is injective }] \\
&\quad \mathsf{L}(\mathsf{T}(\mathcal{S})) \cap (\mathsf{U}_{\text{pass}} \cup \mathsf{U}_{\text{fail}}) \subseteq \mathsf{U}_{\text{pass}} \\
&\Leftrightarrow \mathsf{T}(\mathcal{S}) \underline{\text{pass}}_{\preceq} \mathsf{U}
\end{aligned}
$$

and

$$
\begin{aligned}
\mathcal{S} \underline{\text{pass}}_{\sim} \mathsf{T}^*(\mathsf{U}) &\Leftrightarrow [\mathsf{T}^*(\mathsf{U}) = W] \\
&\quad \mathsf{L}(\mathcal{S}) \cap (W_{\text{pass}} \cup W_{\text{fail}}) = W_{\text{pass}} \\
&\Leftrightarrow [\text{Derivation } (*), W_{\text{pass}} = \mathsf{f_U}(\mathsf{U}_{\text{pass}})] \\
&\quad \mathsf{f_U}\big(\mathsf{L}(\mathsf{T}(\mathcal{S})) \cap (\mathsf{U}_{\text{pass}} \cup \mathsf{U}_{\text{fail}})\big) = \mathsf{f_U}(\mathsf{U}_{\text{pass}}) \\
&\Leftrightarrow [\mathsf{f_U} \text{ is injective}] \\
&\quad \mathsf{L}(\mathsf{T}(\mathcal{S})) \cap (\mathsf{U}_{\text{pass}} \cup \mathsf{U}_{\text{fail}}) = \mathsf{U}_{\text{pass}} \\
&\Leftrightarrow \mathsf{T}(\mathcal{S}) \underline{\text{pass}}_{\sim} \mathsf{U}
\end{aligned}
$$

□

Complexity considerations for $(\mathsf{T}, \mathsf{T}^*)$ With Theorem 7.3 and Theorem 9.1, it has been established that the translation of FSM testing theories to theories for RIOSTS is admissible, using the pair $(\mathsf{T}, \mathsf{T}^*)$ for translation of models and test cases. Both mappings are implementable, so the question of their complexity in practical applications arises.

The complexity for calculating an image value $\mathsf{T}(\mathcal{S})$ of the model map mainly depends on the effort to calculate the image of the transition index function introduced in Section 7.4. As shown in the proof of Lemma 7.3, this task has worst-case complexity 2^{n^n}, where n is the number of pairs $(\boldsymbol{d}, \boldsymbol{e})$ that enumerate the possible value combinations of internal states and outputs. The experimental evaluations published in [4, 34, 52] show that this worst-case complexity hardly ever occurs when processing meaningful controller models.

The effort to calculate the RIOSTS test case $W = \mathsf{T}^*(\mathsf{U})$ associated with FSM test case U is obviously linear in the size of the FSM test cases $\mathsf{U} = (\mathsf{U}_{\text{pass}}, \mathsf{U}_{\text{fail}})$. When calculating representatives of input equivalence classes $\mathsf{X} \in \mathcal{I}$, however, an SMT solver is used to solve the defining constraint of X. Depending on the arithmetic formulas contained in this constraint, the effort for solving this constraint may become exponential in the number of bits needed to represent the data items involved.

Chapter 10

Complete Testing Theories for RIOSTS

10.1 Overview

In this section, Theorem 2.1 is "instantiated" for the case of RIOSTSs and FSMs. This allows us to establish novel complete IECP testing theories for RIOSTSs by means of translation of known FSM theories. The new strategies apply to RIOSTS fault models with varying types of reference models, conformance relations, and fault domains; the combinations are shown in Table 10.1.

Throughout this section, a fixed signature $\mathsf{Sig} = (\mathsf{I}, \mathsf{O}, \mathsf{D}_\mathsf{I}, \mathsf{D}_\mathsf{O})$ for RIOSTSs is considered, such that, while D_I may be infinite, D_O is finite. Let $\mathcal{D}(\mathsf{Sig}, \mathcal{I}) \subseteq \mathsf{Sig}$ denote the subdomain of completely defined RIOSTSs in Sig with finitely many internal states, such that $\mathcal{I}$ is an input equivalence partitioning or a refinement thereof for all $\mathcal{S} \in \mathcal{D}(\mathsf{Sig}, \mathcal{I})$. The function $\mathsf{T} : \mathcal{D}(\mathsf{Sig}, \mathcal{I}) \to \mathrm{FSM}(\mathcal{I}, \mathsf{D}_\mathsf{O})$ denotes the model map as introduced in Section 7.6. Recall that $\mathsf{F}(\mathcal{D}(\mathsf{Sig}, \mathcal{I}), \leq)$ denotes the set of all fault models defined on $\mathcal{D}(\mathsf{Sig}, \mathcal{I})$; for RIOSTS, the conformance relation $\leq$ is always $\sim$ (I/O-equivalence) or $\preccurlyeq$ (reduction).

Table 10.1: Complete IECP testing theories for RIOSTSs.

Reference Model	Conformance Relation	Fault Domain	Theorem
deterministic	I/O-equivalence	deterministic	10.2
nondeterministic	reduction	nondeterministic	10.3
nondeterministic	I/O-equivalence	nondeterministic	10.4
nondeterministic	reduction	deterministic	10.5

10.2 Theory Translation Theorem – From FSM Theories to RIOSTS Theories

The following theorem is an instance of the general theory translation Theorem 2.1. It states the conditions for existing complete testing theories about FSMs to induce likewise complete testing theories for RIOSTS. It further explains, how RIOSTS test suites are created from FSM suites by means of translation, using the test case map $\mathsf{T}^* : \mathrm{TC}(\mathcal{I}, \mathsf{D_O}) \to \mathrm{TC}(\mathsf{Sig})$ constructed in Section 8.2.

Theorem 10.1 *Let* $\mathsf{F}_{\mathrm{FSM}}$ *be a set of FSM fault models with conformance relation* $\leq \in \{\sim, \preccurlyeq\}$*. Then every complete FSM testing theory* $\mathbf{TS}_{\mathrm{FSM}} : \mathsf{F}_{\mathrm{FSM}} \to \mathbb{P}(\mathit{TC}(\mathcal{I}, \mathsf{D_O}))$ *induces a complete RIOSTS testing theory, if* F *and* $\mathbf{TS}$ *are defined by* $\mathbf{TS} : \mathsf{F} \to \mathbb{P}(\mathit{TC}(\mathsf{Sig}))$ *with*

1. $\mathsf{F} = \{(\mathcal{S}, \leq, \mathcal{D}) \in \mathsf{F}(\mathcal{D}(\mathsf{Sig}, \mathcal{I}), \leq) \mid \exists (\mathsf{M}, \leq, \mathcal{D}_{\mathrm{FSM}}) \in \mathsf{F}_{\mathrm{FSM}} : \mathsf{T}(\mathcal{S}) = \mathsf{M} \wedge \mathsf{T}(\mathcal{D}) \subseteq \mathcal{D}_{\mathrm{FSM}}\}$,
2. $\mathbf{TS}(\mathcal{S}, \leq, \mathcal{D}) = \mathsf{T}^*(\mathbf{TS}_{\mathrm{FSM}}(\mathsf{T}(\mathcal{S}), \leq, \mathcal{D}_{\mathrm{FSM}}))$*, for some* $(\mathsf{T}(\mathcal{S}), \leq, \mathcal{D}_{\mathrm{FSM}}) \in \mathsf{F}_{\mathrm{FSM}}$ *with* $\mathsf{T}(\mathcal{D}) \subseteq \mathcal{D}_{\mathrm{FSM}}$.

Proof. For the model map $\mathsf{T} : \mathcal{D}(\mathsf{Sig}, \mathcal{I}) \to \mathrm{FSM}(\mathcal{I}, \mathsf{D_O})$ and test case map $\mathsf{T}^* : \mathrm{TC}(\mathcal{I}, \mathsf{D_O}) \to \mathrm{TC}(\mathsf{Sig})$ it has been shown in Theorem 7.3 and Theorem 9.1 that $(\mathsf{T}, \mathsf{T}^*)$ satisfy the satisfaction condition for I/O-equivalence $\sim$ and reduction $\preccurlyeq$. Applying the general theory translation Theorem 2.1 to $(\mathsf{T}, \mathsf{T}^*)$ proves the theorem. □

Recall from Section 7.6, that the model map depends on the choice of the input partition $\mathcal{I}$. The fault models in F contain only reference models $\mathcal{S}$ and

fault domains with members $\mathcal{S}'$, such that $\mathcal{I}$ refines both the IECP of the reference models $\mathcal{S}$ and the potential implementation behaviours $\mathcal{S}'$. Refining $\mathcal{I}$ increases the size of the fault domain, at the cost of having to consider larger alphabets for the associated FSMs. Recall further from Section 8.2, that the test case map $\mathsf{T}^*(\mathsf{U})$ varies with the choice of $\mathsf{f_U}$ which selects concrete representatives from every IEC, each time it is referenced by U as an input to the SUT. Since Theorem 10.1 does not depend on a specific choice of the $\mathsf{f_U}$, this can be interpreted in the way that the theorem still holds if a *random* selection is made from each IEC, whenever it is referenced from some test case.

We are now in the position to transform a variety of testing theories for FSMs to equivalence class testing theories for RIOSTS. The theorems below always state the *existence* of a theory in relation to a set of fault models, as introduced in Section 2.3. In the proofs, however, references to the *concrete construction* of RIOSTS test suites by means of translation from FSM test suites are given.

10.3 Deterministic Reference Model and Deterministic Implementation

The following result has originally been published in [31], but now it can be easily established as one application of Theorem 10.1. Moreover, we extend the existing result with respect to random selection of representatives from input equivalence classes.

Theorem 10.2 *Let $\mathcal{D}_0 = \mathcal{D}_0(\mathsf{Sig}, \mathcal{I}, \mathfrak{m}) \subseteq \mathcal{D}(\mathsf{Sig}, \mathcal{I})$ denote the fault domain of deterministic RIOSTSs, such that for all $\mathcal{S} \in \mathcal{D}_0$, the prime machine $\mathsf{T}(\mathcal{S})$ has at most $\mathfrak{m}$ states.*
Let $\mathsf{F} = \{(\mathcal{S}, \sim, \mathcal{D}_0) \in \mathsf{F}(\mathcal{D}(\mathsf{Sig}, \mathcal{I}), \sim) \mid \mathcal{S} \in \mathcal{D}_0\}$. Then there exist complete finite input equivalence class testing theories $\mathbf{TS} : \mathsf{F} \to \mathbb{P}(TC(\mathsf{Sig}))$ with randomised selection of representatives from each input equivalence class.

Proof. Consider the FSM fault domain $\mathrm{DFSM}(\mathcal{I}, \mathsf{D_O}, \mathfrak{m})$ of completely specified, deterministic FSMs whose prime machines (recall that these are just the minimised deterministic FSM) have at most $\mathfrak{m}$ states. Then the classical W-Method is known to produce finite complete test suites for every

fault model

$$\mathcal{F} \in \mathsf{F}_{\mathrm{DFSM}} = \{(\mathsf{M}, \sim, \mathrm{DFSM}(\mathcal{I}, \mathsf{D}_{\mathsf{O}}, \mathfrak{m})) \mid \mathsf{M} \in \mathrm{DFSM}(\mathcal{I}, \mathsf{D}_{\mathsf{O}}, \mathfrak{m})\}$$

This has been shown in [9, 69]; alternatively, the Wp-Method can be applied, which is known to yield shorter test suites [19]. Let $\mathbf{TS}_{\mathrm{DFSM}} : \mathsf{F}_{\mathrm{DFSM}} \rightarrow \mathbb{P}(\mathrm{TC}(\mathcal{I}, \mathsf{D}_{\mathsf{O}}))$ denote a complete testing theory constructed according to the W-Method or the Wp-Method.

By definition of $\mathcal{D}_0$, and since we know from Lemma 7.6 that for any $\mathcal{S} \in \mathcal{D}_0$, its minimal FSM abstraction $\mathsf{T}(\mathcal{S})$ is also deterministic (and therefore also observable), $\mathsf{T}(\mathcal{D}_0) \subseteq \mathrm{DFSM}(\mathcal{I}, \mathsf{D}_{\mathsf{O}}, \mathfrak{m})$ follows. As a consequence, the general fault model collection

$$\begin{aligned}\mathsf{F} = \{(\mathcal{S}, \leq, \mathcal{D}) \in \mathsf{F}(\mathcal{D}(\mathtt{Sig}, \mathcal{I}), \leq) \mid \\ \exists (\mathsf{M}, \leq, \mathcal{D}_{\mathrm{FSM}}) \in \mathsf{F}_{\mathrm{FSM}} : \mathsf{T}(\mathcal{S}) = \mathsf{M} \wedge \mathsf{T}(\mathcal{D}) \subseteq \mathcal{D}_{\mathrm{FSM}}\}\end{aligned}$$

specified in condition 1 of Theorem 10.1 equals

$$\mathsf{F} = \{(\mathcal{S}, \sim, \mathcal{D}_0) \in \mathsf{F}(\mathcal{D}(\mathtt{Sig}, \mathcal{I}), \sim) \mid \mathcal{S} \in \mathcal{D}_0\}$$

for the deterministic case addressed in the theorem here. Now we can apply Theorem 10.1 to conclude that

$$\begin{aligned}&\mathbf{TS} : \mathsf{F} \rightarrow \mathbb{P}(\mathrm{TC}(\mathtt{Sig})); \\ &(\mathcal{S}, \sim, \mathcal{D}_0) \mapsto \mathsf{T}^*(\mathbf{TS}_{\mathrm{DFSM}}((\mathsf{T}(\mathcal{S}), \sim, \mathrm{DFSM}(\mathcal{I}, \mathsf{D}_{\mathsf{O}}, \mathfrak{m}))))\end{aligned}$$

is a complete testing theory as well. As stated in Section 10.2, random selections may be performed from input classes $\mathsf{X}_{\mathsf{i}} \in \mathcal{I}$, when translating I/O-traces $\overline{\mathbf{x}}/\overline{\mathbf{y}}$ of FSM test cases U to their RIOSTS counterparts in $\mathsf{T}^*(\mathsf{U})$. □

Concrete examples of complete test suites for deterministic RIOSTS have been given in [31, 34, 52].

As discussed in Section 6, $\mathcal{S}' \preccurlyeq \mathcal{S}$ implies $\mathcal{S}' \sim \mathcal{S}$, if $\mathcal{S}$ is deterministic and $\mathcal{S}'$ is completely specified. Therefore no other theories have to be considered if both reference model and implementation model are deterministic and completely specified RIOSTSs.

10.4 Nondeterministic Reference Model and Nondeterministic Implementation

Complete testing assumption The completeness results described below hold under the *complete testing assumption* [26]: this is a fairness hypothesis stating the existence of some $k > 0$, so that, when applying input sequence $\overline{x}$ k times to the SUT, every output sequence $\overline{y}$ that *can* be produced with this $\overline{x}$ *will* be observed. It is then required to execute the complete test suites introduced below at least k times, so that all possible behaviours of the SUT will be observed.

Testing for reduction

Theorem 10.3 *Let $\mathcal{D} = \mathcal{D}(Sig, \mathcal{I}, m) \subseteq \mathcal{D}(Sig, \mathcal{I})$ denote the fault domain of deterministic or nondeterministic RIOSTS, such that for all $\mathcal{S} \in \mathcal{D}$, the prime machine $T(\mathcal{S})$ has at most m states.*
Let $F = \{(\mathcal{S}, \preccurlyeq, \mathcal{D}) \in F(\mathcal{D}(Sig, \mathcal{I}), \preccurlyeq) \mid \mathcal{S} \in \mathcal{D}\}$. Then there exist complete finite input equivalence class testing theories $\mathbf{TS} : F \to \mathbb{P}(TC(Sig))$ with randomised selection of representatives from each input equivalence class.

Proof. We apply the same proof schema as for Theorem 10.2. The image of $\mathcal{D}(Sig, \mathcal{I}, m)$ under the model map T is contained in $\mathrm{FSM}(\mathcal{I}, D_O, m)$, the set of all FSMs whose prime machines have at most m states. It has been shown in [26, 59] that complete finite test suites exist for fault models of the type $\mathcal{F} = (M, \preccurlyeq, \mathrm{FSM}(\mathcal{I}, D_O, m))$ with $M \in \mathrm{FSM}(\mathcal{I}, D_O, m))$. To this end, the authors construct suites of adaptive FSM tests and apply the state counting method. To our best knowledge, the construction method given in [59] results in the shortest complete test suites that are currently known for this problem.

Setting $F_{\mathrm{FSM}} = \{(M, \preccurlyeq, \mathrm{FSM}(\mathcal{I}, \Sigma_O, m)) \mid M \in \mathrm{FSM}(\mathcal{I}, D_O, m)\}$, this induces a complete FSM testing theory $\mathbf{TS}_{\mathrm{FSM}} : F_{\mathrm{FSM}} \to \mathbb{P}(\mathrm{TC}(\mathcal{I}, D_O))$. Since $T(\mathcal{S}) \in \mathrm{FSM}(\mathcal{I}, D_O, m)$ and $T(\mathcal{D}) \subseteq \mathrm{FSM}(\mathcal{I}, D_O, m)$, we can apply Theorem 10.1 to obtain a complete finite testing theory $\mathbf{TS} : F \to \mathbb{P}(\mathrm{TC}(Sig))$ by setting $\mathbf{TS}(\mathcal{S}, \preccurlyeq, \mathcal{D}) = T^*(\mathbf{TS}_{\mathrm{FSM}}(T(\mathcal{S}), \preccurlyeq, \mathrm{FSM}(\mathcal{I}, D_O, m)))$.

As shown in the proof of Theorem 10.2, randomised selection of representatives from each input equivalence class referenced in test cases $U \in \mathbf{TS}_{\mathrm{FSM}}(T(\mathcal{S}), \preccurlyeq, \mathrm{FSM}(\mathcal{I}, D_O, m))$ preserves the completeness property.

□

Table 10.2: Test cases generated for the alarm indication system from Example 7, using the generalised Wp-Method.

Test Case	Inputs	Expected Outputs
1.	100	{OK, ALARM}
2.	90.100	{OK.OK, OK.ALARM}
3.	95.100	{OK.OK, OK.ALARM}
4.	100.100	{OK.OK, ALARM.ALARM}
5.	101.100	{ALARM.ALARM}
6.	100.50.100	{OK.OK.OK, OK.OK.ALARM, ALARM.OK.ALARM}
7.	100.99.100	{OK.OK.OK, OK.OK.ALARM, ALARM.ALARM.ALARM}
8.	100.100.100	{OK.OK.OK, ALARM.ALARM.ALARM}
9.	100.500.100	{OK.ALARM.ALARM, ALARM.ALARM.ALARM}

Testing for I/O-equivalence We can also translate a theory for checking the I/O-equivalence of nondeterministic (or deterministic) FSMs to a theory for testing the I/O-equivalence of nondeterministic (or deterministic) RIOSTS.

Theorem 10.4 *Let $\mathcal{D} = \mathcal{D}(\mathsf{Sig}, \mathcal{I}, \mathfrak{m}) \subseteq \mathcal{D}(\mathsf{Sig}, \mathcal{I})$ denote the fault domain of deterministic or nondeterministic RIOSTS, such that for all $\mathcal{S} \in \mathcal{D}$, the prime machine $\mathsf{T}(\mathcal{S})$ has at most $\mathfrak{m}$ states.*
Let $\mathsf{F} = \{(\mathcal{S}, \sim, \mathcal{D}) \in \mathsf{F}(\mathcal{D}(\mathsf{Sig}, \mathcal{I}), \sim) \mid \mathcal{S} \in \mathcal{D}\}$. Then there exists a complete finite input equivalence class testing theory $\mathbf{TS} : \mathsf{F} \to \mathbb{P}(\mathit{TC}(\mathsf{Sig}))$ with randomised selection of representatives from each input equivalence class.

Proof. For nondeterministic FSM, an extension of the Wp-Method originally described in [19] has been presented in [45]. This extension describes the generation of a complete testing strategy for checking I/O-equivalence in presence of nondeterministic, completely specified reference models. The remainder of the proof is the same as for Theorem 10.3. □

Example 16. We apply the generalised Wp-Method (see Section 4.8.1) to the alarm indication system $\mathcal{S}$ introduced in Example 7, using concrete values $\mathrm{max} = 100$ and $\delta = 10$. For the fault domain $\mathcal{D} = \mathcal{D}(\mathsf{Sig}, \mathcal{I}, \mathfrak{m})$ we assume $\mathfrak{m} = 3$, that is, the prime machines of all FSM abstractions

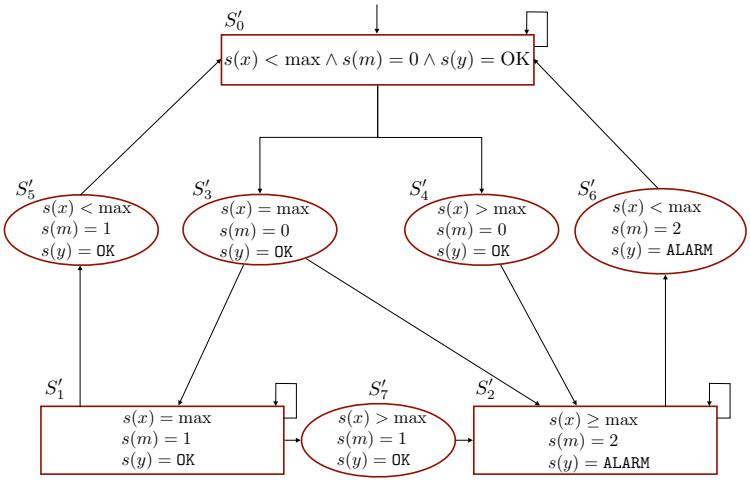


Figure 10.1: Erroneous implementation $\mathcal{S}'$ of the alarm indication system shown in Fig. 6.1.

from potential SUT behaviours contained in the fault domain have at most 3 states. Furthermore, we assume that the IECP $\mathcal{I}$ elaborated for the system in Example 8 is already fine-grained enough to serve as an IECP $\mathcal{I}$ of each possible SUT behaviour in the fault domain. Note that the FSM abstraction M of the alarm indication system shown in Fig. 7.1 is already minimised and observable.

According to the test suite generation algorithm of the Wp-Method, the following input sequences are derived first from the FSM abstraction of the alarm system which was calculated in Example 9 (the FSM is shown in Fig. 7.1).

1. A *state cover* $\mathsf{Q} = \{\varepsilon, \mathsf{X}_3\}$ (see Section 4.2). The input sequences in Q are suitable to reach every state in the reference FSM.

2. A *transition cover* $\mathsf{P} = \{\varepsilon, \mathsf{X}_1, \mathsf{X}_2, \mathsf{X}_3, \mathsf{X}_4, \mathsf{X}_3.\mathsf{X}_1, \mathsf{X}_3.\mathsf{X}_2, \mathsf{X}_3.\mathsf{X}_3, \mathsf{X}_3.\mathsf{X}_4\}$ (see Section 4.2). A transition cover contains the empty input trace and input traces suitable to reach every state of the reference FSM and applying every input to that state. Typically, the transition cover is constructed from the state cover by appending every input to every input trace contained in the latter.

3. A *characterisation set* $\mathsf{W} = \{\mathsf{X}_3\}$ (see Section 3.7). This set is suit-

able to distinguish all states of the reference FSM, in the sense that applying all input traces of W to each state will lead to different sets of output traces. For our example, X_3 applied to S_0 results in output set {OK, ALARM}. When applied to S_1 it yields output {OK}, and applied to S_2 it results in {ALARM}.

4. A set of *state identification sets* $W_1 = W_2 = W_3 = W$ (see Section 3.7). Each W_i is a subset of prefixes from input traces in the characterisation set, suitable to distinguish FSM state i from each other state. In the example here the W_i already coincide with W.

As specified in Section 4.8.1, the complete test suite is constructed from the input trace fragments above, and this results in 9 test cases, after deleting redundant input traces that are prefixes of longer ones contained in the suite.

$$\begin{aligned} P.W = \{&X_3, X_1.X_3, X_2.X_3, X_3.X_3, X_4.X_3, \\ &X_3.X_1.X_3, X_3.X_2.X_3, X_3.X_3.X_3, X_3.X_4.X_3\} \end{aligned}$$

Define FSM abstract test cases (see Section 4.1) for M by

$$U_M(\overline{X}) = (U_{\text{pass}}(\overline{X}), U_{\text{fail}}(\overline{X}))$$

with

$$U_{\text{pass}}(\overline{X}) = Z(\overline{X}) \cap L(M) \quad \text{and} \quad U_{\text{fail}}(\overline{X}) = Z(\overline{X}) \setminus L(M),$$

where $Z(\overline{X})$ is defined by

$$Z(\overline{X}) = \{\overline{X}/y_1 \dots y_{|\overline{X}|} \mid y_1 \dots y_{|\overline{X}|} \in \Sigma_O^*\}.$$

Using this notation, the complete test suite associated with the set $P.W$ of input traces may be expressed as

$$TS_{\text{FSM}} = \{U_M(\overline{X}) \mid \overline{X} \in P.W\}.$$

The translated RIOSTS test cases use arbitrary selections from the IEC referenced in the input traces from TS_{FSM}. For example, the translated RIOSTS test suite could apply the input sequences specified in the second column of Table 10.2, expecting the output sets specified in the third column.

Consider now the erroneous implementation $\mathcal{S}'$ of the alarm indication system, as shown in Fig. 10.1. It performs illegal transitions from quiescent states S'_2 via transient states S'_6 to quiescent states S'_0: the OK-indication is

already given, when after an alarm the input x drops below threshold max, instead of $\max - \delta$. This error is identified, for example, by Test Case 7 in Table 10.2, which is generated by random selection from the FSM test case $\mathsf{U}_{\mathsf{M}}(\mathsf{X}_3.\mathsf{X}_2.\mathsf{X}_3)$ as

$$\begin{aligned} W_{\mathcal{S}}(100.99.100) &= (W_{\text{pass}}, W_{\text{fail}}) \\ W_{\text{pass}} &= \{100.99.100/\texttt{OK.OK.OK}, 100.99.100/\texttt{OK.OK.ALARM}, \\ &\quad 100.99.100/\texttt{ALARM.ALARM.ALARM}\} \\ W_{\text{fail}} &= \{100.99/\texttt{OK.ALARM}, 100.99/\texttt{ALARM.OK}\} \end{aligned}$$

The SUT $\mathcal{S}'$ fails the test case $W_{\mathcal{S}}(100.99.100)$ because $100.99/\texttt{ALARM.OK} \in \mathsf{L}(\mathcal{S}') \cap W_{\text{fail}}$. □

10.5 Nondeterministic Reference Model and Deterministic Implementation

It is interesting to note that the knowledge about SUT determinism can be exploited to obtain shorter and simpler complete test suites.

Theorem 10.5 *Let $\mathcal{D} = \mathcal{D}(\mathsf{Sig}, \mathcal{I}, \mathfrak{m}) \subseteq \mathcal{D}(\mathsf{Sig}, \mathcal{I})$ denote the fault domain of deterministic or nondeterministic RIOSTS, such that for all $\mathcal{S} \in \mathcal{D}$, the prime machine $\mathsf{T}(\mathcal{S})$ has at most $\mathfrak{m}$ states. Let $\mathcal{D}_0$ be the restriction of $\mathcal{D}$ to deterministic RIOSTS as defined in Theorem 10.2. Let $\mathsf{F} = \{(\mathcal{S}, \preccurlyeq, \mathcal{D}_0) \in \mathsf{F}(\mathcal{D}(\mathsf{Sig}, \mathcal{I}), \leq) \mid \mathcal{S} \in \mathcal{D}\}$. Then there exists a complete finite input equivalence class testing theory $\mathbf{TS} : \mathsf{F} \to \mathbb{P}(TC(\mathsf{Sig}))$ with randomised selection of representatives from each input equivalence class.*

Proof. In [54] the authors introduce a complete testing theory for fault models $(\mathsf{M}, \preccurlyeq, \text{DFSM}(\mathcal{I}, \mathsf{D}_{\mathsf{O}}, \mathfrak{m}))$ with $\mathsf{M} \in \text{FSM}(\mathcal{I}, \mathsf{D}_{\mathsf{O}}, \mathfrak{m})$. Since $\mathsf{T}(\mathcal{D}_0) \subseteq \text{DFSM}(\mathcal{I}, \mathsf{D}_{\mathsf{O}}, \mathfrak{m})$ and $\mathsf{T}(\mathcal{D}) \subseteq \text{FSM}(\mathcal{I}, \mathsf{D}_{\mathsf{O}}, \mathfrak{m})$, the proof can now be completed just as for Theorem 10.3. □

The practical relevance of testing deterministic SUTs against nondeterministic reference models with respect to reduction $\preccurlyeq$ is high: even in the case of safety-critical systems, reference models may be nondeterministic, because design options are left open or external components can only be modelled as nondeterministic black boxes. The SUT itself, however, will be usually

deterministic for safety-critical applications. In contrast to this, testing deterministic implementations $\mathcal{S}'$ for I/O-equivalence against nondeterministic reference models $\mathcal{S}$ is *not* of interest: nondeterminism of $\mathcal{S}$ implies the existence of I/O-traces $\overline{\mathbf{x}}/\overline{\mathbf{y}}, \overline{\mathbf{x}}/\overline{\mathbf{y}'} \in \mathsf{L}(\mathcal{S})$ with $\overline{\mathbf{y}} \neq \overline{\mathbf{y}'}$. Since the implementation $\mathcal{S}'$ is deterministic, it can perform at most one of these traces, so $\mathcal{S}' \preccurlyeq \mathcal{S}$ may hold, but never $\mathcal{S}' \sim \mathcal{S}$ if $\mathcal{S}$ is truly nondeterministic.

Analogously, testing nondeterministic SUTs against deterministic reference models is not of interest: if an implementation model $\mathcal{S}'$ is truly nondeterministic, it will contain I/O-traces $\overline{\mathbf{x}}/\overline{\mathbf{y}}, \overline{\mathbf{x}}/\overline{\mathbf{y}'} \in \mathsf{L}(\mathcal{S}')$ with $\overline{\mathbf{y}} \neq \overline{\mathbf{y}'}$, where at least one of those traces is not contained in $\mathcal{S}$ since it is deterministic. Therefore neither $\mathcal{S}' \preccurlyeq \mathcal{S}$, nor $\mathcal{S}' \sim \mathcal{S}$ can hold.

10.6 Weaker Test Strategies: Single Output Fault

We present an example how FSM testing theories with lesser test strength can be transformed into corresponding input equivalence class testing theories. Here "lesser" means that the test suites associated with such a theory can only prove conformance under additional assumptions regarding the admissible fault domains.

Theorem 10.6 *For $\mathcal{S} \in \mathcal{D}_0(\mathtt{Sig}, \mathcal{I})$, let $\mathcal{D}_s(\mathcal{S}, \mathcal{I}) \subseteq \mathcal{D}_0(\mathtt{Sig}, \mathcal{I})$ denote the* single output fault domain *of deterministic RIOSTS $\mathcal{S}'$ whose prime machines $\mathsf{T}(\mathcal{S}')$ are isomorphic to $\mathsf{T}(\mathcal{S})$ after replacing at most one output label at a single transition. Let $\mathsf{F} = \{(\mathcal{S}, \sim, \mathcal{D}_s(\mathcal{S}, \mathcal{I})) \in \mathsf{F}(\mathcal{D}(\mathtt{Sig}, \mathcal{I}), \leq) \mid \mathcal{S} \in \mathcal{D}_0(\mathtt{Sig}, \mathcal{I})\}$. Then there exists a complete finite input equivalence class testing theory $\mathbf{TS} : \mathsf{F} \to \mathbb{P}(\mathit{TC}(\mathtt{Sig}))$ with randomised selection of representatives from each input equivalence class.*

Proof.

The so-called T-Method [47] generates test suites for a given reference model M that perform transition tours on the model, that is, the test cases are selected in such a way that each transition of the reference DFSM is tested at least once. It is easy to see that this induces a complete testing theory $\mathbf{TS}$: $\mathsf{F} \to \mathbb{P}(\mathrm{TC}(\mathcal{I}, \mathsf{D_O}))$ with $\mathsf{F} = \{(\mathsf{M}, \sim, \mathrm{DFSM_s}(\mathsf{M})) \mid \mathsf{M} \in \mathrm{DFSM}(\mathcal{I}, \mathsf{D_O})\}$, where $\mathrm{DFSM_s}(\mathsf{M})$ is the fault domain of DFSMs $\mathsf{M}' \in \mathrm{DFSM}(\mathcal{I}, \mathsf{D_O})$ whose

prime machines are isomorphic to that of M, up to a change of at most one output signal.

As in the proofs of the theorems above, this induces a complete finite testing theory for RIOSTS with random selection of inputs from each IEC. □

10.7 Complexity Considerations

As seen in the proofs of the theorems above, the mapping of complete FSM test suites to complete RIOSTS test suites preserves the number of test cases. Moreover, the lengths of maximal I/O-traces of FSM test cases U are equal to the maximal numbers of test steps to be performed by the translated test case $\mathsf{T}^*(\mathsf{U})$: the functions f_U used in the transformations $\mathsf{T}^*(\mathsf{U})$ map FSM I/O-traces $\overline{\mathsf{X}}/\overline{\mathsf{y}}$ to RIOSTS traces $\overline{\mathsf{x}}/\overline{\mathsf{y}}$ of the same length. As a consequence, complexity results about both number and length of test cases can be directly transferred from the FSM test suite to the associated RIOSTS suite.

Chapter 11

Related Work

The proof strategy applied in the chapters above and described in Chapter 2 is inspired by the theory of institutions introduced in [22]. There the concepts of model maps, sentence translation maps (these are generalisations of the test case map constructed in this article), and the satisfaction condition have originally been introduced for formally establishing the fact that *"truth is invariant under change of notation"* [22, p. 101]. The 'sentences' associated with a signature were typically formulas φ expressed in a specific logic, and the satisfaction condition required that models and formulas could be translated in a way preserving the satisfaction relation, in the sense that $\mathsf{T}(\mathsf{M}) \models \varphi \Leftrightarrow \mathsf{M} \models \mathsf{T}^*(\varphi)$. In our exposition, sentences are represented by test cases, and the $\models$ relationship is replaced by the $\underline{\text{pass}}$ relationship. It is possible to translate the "model passes test" terminology $\mathsf{M}\ \underline{\text{pass}}\ \mathsf{U}$ into the more conventional "model fulfils formula" notation $\mathsf{M} \models \varphi$, but this neither simplifies the exposition presented here, nor does it yield any additional insight into the field of complete testing theories and their translation between different signatures. It should be noted further, that the classical theory of institutions as presented in [22] would only be applicable to signatures $\mathtt{Sig}_1$, $\mathtt{Sig}_2$ belonging to a class of closely related models, such as FSMs over different input/output alphabets or process algebras from the same formalism, but with different alphabets (see, for example [46]). In our application, however, theory translation is performed between quite distinct formalisms (FSMs and Kripke structures). This situation is more complicated, because we map models and formulas between signatures of *different* institutions. This is typically handled by institutions with many-sorted signatures, and these can be conveniently represented by Grothendiek Institutions [11, Chapter 12], if

a formal model-theoretic underpinning is desired.

The Unifying Theories of Programming (UTP) [27] provide an alternative to the institution-based approach for translating theories. The UTP is a relational semantic framework allowing to capture and relate theories for a variety of modelling and programming formalisms. Each formalism and its theories are captured in a lattice of logical formulas, and theory translation is enabled if a Galois Connection between these lattices can be established. An illustration of the UTP-based approach in the context of testing is given in [7], where a passive testing theory elaborated in the context of Kripke structures is translated to a corresponding theory for the CSP process algebra. Further testing theories for modelling formalisms interpreted in UTP semantics have been constructed in [6] for the Circus modelling language, which also allows for the introduction of large and complex data types.

The formal framework for constructing complete test suites in general, and for introducing equivalence class testing methods preserving completeness in particular, has been laid out in [20]. A more restricted completeness notion for process algebras over finite alphabets had been investigated in [25], where it was shown that specific – though infinite – adaptive test suites are suitable to characterise refinement relations between nondeterministic processes. The formal approach to investigate completeness in relation to fault models has originally been introduced in [56, 57].

In the field of (usually nondeterministic) process algebras, the work from [25] has been extended in [50, 49]. This resulted in effectively implementable finite test suites for fault domains specified by an upper bound on the length of traces after which potential deviations of the SUT from the reference process could occur. Similar results have been obtained for labelled transition systems and the ioco conformance relation [67], and for (deterministic) timed I/O-automata in [65]. These results, however, all depended on the finiteness of alphabets. This changed in [18], where the authors use symbolic transitions systems to avoid the enumeration of inputs, outputs, or internal state during test generation. Their completeness results, however, only apply to test suites that are infinite.

In the FSM world, a multitude of complete test strategies has been developed over the decades, starting with Chow's and Vasilevskii's W-Method [9, 69] and its optimisations such as [19] for complete DFSMs and covering a wide variety of test configurations involving nondeterministic FSMs; we name [26, 45, 57, 54, 55] for some of the more prominent results. Extended finite state machines (EFSMs) add variables to the notion

of FSMs, and this leads to investigations about equivalence class partitions in a natural way. In [40] the authors advocate genetic algorithms for constraint solving. In contrast to this, [28] consider test generation for EFSMs as a witness generation problem for global CTL model checking and apply this technique to EFSMs with finite variable domains. EFSMs can be encoded as STS in a straight-forward way. Indeed, the authors of [28] provide such an encoding and extend it to Kripke Structures by adding a labelling function for atomic propositions, as required by the classical CTL model checking algorithms [10]. As a consequence, the equivalence class testing strategy presented here can be directly applied to testing EFSMs with large or infinite input data domains and finite outputs and I/O-equivalence classes. Our general approach to model-based test generation differs from [40, 28] in the way that test objectives are encoded as bounded model checking instances, and we use an SMT solver to calculate concrete test data [53, 51], because – here we follow [2] and the numerous references to model-based testing approaches given there – MBT test generation tasks can be regarded as variants of constraint solving problems. The calculation of representatives of input equivalence classes not only works for integral data types, but for floats as well, because the SMT solver SONOLAR integrated in RT-Tester supports floating point datatypes and operations [43].

An early formalised presentation of equivalence class partitioning algorithms has been given in [12] for testing against VDM models. The authors already point out the role of DNF representations which are also used during an intermediate transformation step in the implementation of our strategy [31, Section 3]. They also use finite state machines to generate the sequences of processing steps which are necessary to apply equivalence class testing in presence of internal state. They do not, however, give any proofs about the completeness of their test suite which, from our understanding, is only ensured when performing grey-box tests allowing to analyse internal SUT states and in case of deterministic operations. The black-box equivalence class testing problem addressed in our article is not covered in [12]. Similar results (also without completeness proofs) have been achieved in [24] for testing against Z specifications. There the automated test generation has been mechanised using theorem prover support.

The authors of [23] investigate refined data abstraction techniques for the purpose of equivalence class definition, where the classes are denoted as *hyperstates*, and the concept is applied to testing against abstract state machine models. They sketch for white box tests, how a complete test suite

could be created [23, Section 4]: the transition cover approach discussed there is applicable for SUT where the internal state (respectively, its abstraction) can be monitored during test execution.

Adaptive random testing [8] focuses on techniques to evenly spread the test cases over the complete input domain. Most research presented by other authors so far concentrates on testing non-reactive software modules, where test cases are specified by single input vectors instead of the input sequences considered in our reactive systems setting. An example of the application of adaptive random testing and search-based testing to realtime embedded systems is given in [3].

Part IV

Fuzz Testing

Chapter 12

Fuzz Testing

12.1 Objectives

There is currently (2020) quite a hype about fuzz testing. At first glance, this may come as a surprise, since fuzzing is just yet another random testing method. The hype, however, is well-justified, because fuzz testing

- has helped to uncover quite a number of infamous bugs in important software packages,
- is simple to use in comparison to, for example, code verification by means of provers, bounded or global model checking,
- can be applied to large software packages that could not be handled by model checkers.

In this chapter, we focus on the practical application of fuzz testing and less on its underlying methods. For tool support, the LLVM libFuzzer library is used, since it seems to be the most mature fuzzing tool currently available and definitely fit for application in an industrial context.

12.2 LLVM libFuzzer – Capabilities

The fuzzing library *libFuzzer* is part of the well-known LLVM compiler framework. It is integrated in the Clang compiler since version 6.0. The library contains an *in-process* fuzzing engine. This means that the SUT must be

linked as (a library of) object files to the fuzzing engine. In particular, the main program for executing the the fuzz tests is provided by libFuzzer: if you wish to test your own main program, the entry point

```
int main(int argc, const char* argv[])
```

needs to be renamed, for example in

```
int my_main(int argc, const char* argv[])
```

The fuzzing process implemented in libFuzzer is *coverage guided*. This means that the random data selection is influenced by the yet uncovered code portions: input data promising to cover new branches of the software is preferred to input data leading to re-execution of branches already explored before. In this sense, the fuzzing method implemented in libFuzzer is a (very special) variant of *adaptive random testing*: the random input data selection is "biased" towards input data promising to drive the software under test into new branches that have been uncovered so far.

The coverage guidance is accelerated by means of a *corpus*. This is a collection of sample inputs to the SUT that help to cover specific portions of the code. By mutating the elements of the corpus, the fuzzer detects new test cases that are able to cover new portions of the code, and can therefore be added to the corpus.

Furthermore, the fuzzing process is *evolutionary*: test cases are produced in successive generations, each new generation produced from the previous ones by application of genetic algorithms. This algorithm lets some test cases of the previous generation "die" in favour of new test cases to become members of the next generation.

12.3 libFuzzer – Interface to the SUT

The libFuzzer requires a C-procedural interface to the SUT. Typically, this is realised as a wrapper calling the SUT provided via the fuzzer interface. The library expects the SUT to be reachable by means of a user-provided interface

```
1 int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
2   map Data to real inputs of the SUT;
3   call the SUT;
4   return 0;
5 }
```

If used for testing C++ code, users need to provide an interface

```
1  extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data,
2                                        size_t Size) {
3    map Data to real inputs of the SUT;
4    call the SUT;
5    return 0;
6  }
```

Typically, this interface is provided by the test harness, so that the SUT code remains unchanged. When the `LLVMFuzzerTestOneInput()` function returns, the LLVM fuzzer calls the function again, but now with a new input data chunk `Data` with random contents and random length `Size`. This goes on until the function terminates (a conditional call to `exit()` can be made after the SUT has been called) or crashes anywhere in the SUT code, for example, due to a segmentation fault, due to an arithmetic exception, or due to a failed assertion[1].

Example 17. Suppose you wish to test the following function `getMin()` which is meant to return the smallest value in a buffer of integers. The length of the buffer (number of int-cells provided) is passed as the second parameter. For the boundary case where the buffer length is 0, we wish to return the maximum signed integer (just for the sake of this example).

```
1  #include <limits.h>
2  #include "mylib.h"
3
4  int getMin(const int* buffer, size_t numIntElements) {
5    if ( numIntElements == 0 ) return INT_MAX;
6    int idx;
7    int min = buffer[0];
8    for ( idx = 1; idx < numIntElements; idx++ ) {
9       if ( buffer[idx] < min ) min = buffer[idx];
10   }
11   return min;
12 }
```

Suppose further that this function resides in file `mylib.c`, and that the prototype is declared in header file `mylib.h`. To separate the test harness from the SUT, we create an additional file `testharness.c` where the fuzzer interface is implemented:

[1]Recall that in C and C++, logical checks can be inserted into code using the `assert` macro (see ASSERT(3) in the manual pages)

```
1  #include "mylib.h"
2
3  int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
4      size_t idx;
5      // Adapt data to SUT input format
6      const int *buffer = (const int*)Data;
7      size_t numIntElements = Size/sizeof(int);
8      // Call the SUT
9      int result = getMin(buffer,numIntElements);
10     return 0;
11 }
```

Here, the mapping from the byte-data buffer `Data` provided by the fuzzer to the int-buffer required by the SUT is straight forward: it just requires a cast to the int-datatype and a transformation of byte-length to int-length of the buffer. In other situations, the following more complex transformations will become necessary:

- Mapping of the `Data` buffer into the components of an input structure of the SUT.
- Padding the rest of a structure with default values if `Data` is shorter than the structure.
- Mapping of the `Data` buffer into several input parameters and global variables representing the SUT input interface.

Moreover, if the `Data` buffer is longer than the SUT input interface, one execution of `LLVMFuzzerTestOneInput()` should lead to *several* SUT calls, each call using a new portion of `Data` as input. □

12.4 Creating a Fuzzer Program With Clang

After having prepared the test harness as described above, the SUT and the harness are compiled and linked with libFuzzer. To this end, `clang` offers the following variants of compiler calls.

Basic fuzzing. In the simplest case, the compiler is called as follows.[2]

[2]It is also possible to use other optimisation options, but `-O0` guarantees that all variables can be inspected in the debugger, after an error has been detected.

```
clang -o <fuzzer-executable-name> \
      -g -O0 -fsanitize=fuzzer \
      <further options> \
      <testharness>.c \
      <sut-file1>.c <sut-file2>.c ...
```

All variants of fuzzer compilation use the option `-fsanitize=fuzzer` to indicate that a "fuzzable" program should be generated. Under `<further options>` additional compiler options that are unrelated to fuzzing are specified, such as `-I` options do indicate paths where to for header files. Next, all C-files to be compiled, including the harness file, are added. The SUT may consist of several files.

Create a fuzzer with address checks. Compiling with options

```
clang -o <fuzzer-executable-name> \
      -g -O0 -fsanitize=fuzzer,address \
      <further options> \
      <testharness>.c \
      <sut-file1>.c <sut-file2>.c ...
```

will produce an executable which performs additional address-related checks and aborts the program execution if one of these checks fail.

1. Out-of-bounds accesses to heap, stack and globals (this includes array boundary checks)
2. Use-after-free
3. Use-after-return
4. Use-after-scope
5. Double-free, invalid free

Create a fuzzer with integer overflow checks. Compiling with options

```
clang -o <fuzzer-executable-name> \
      -g -O0 -fsanitize=fuzzer,signed-integer-overflow \
      <further options> \
      <testharness>.c \
      <sut-file1>.c <sut-file2>.c ...
```

will produce an executable which performs additional checks for overflows in signed integers. All fuzzer-related options can be combined, so that compilation command

```
clang -o <fuzzer-executable-name> \
      -g -O0 -fsanitize=fuzzer,address,signed-integer-overflow \
      <further options> \
      <testharness>.c \
      <sut-file1>.c <sut-file2>.c ...
```

generates an executable where all the checks described above are made.

12.5 Executing a Fuzzer Program Created With Clang

12.5.1 Simple Execution

The fuzzer executable created with one of the clang-commands described above is now executed in the simplest case by starting the program executable without additional parameters, like

```
./<fuzzer-executable-name>
```

This will lead to a program execution where the `LLVMFuzzerTestOneInput()`-function is repeatedly called with varying random inputs, guided by the coverage achieved so far (the coverage is internally monitored during program execution). The program will only stop when it crashes due to a segmentation fault, a failed assertion, or due to address violations or signed integer overflows detected by libFuzzer. If one of these abort situations occur, a file named

```
crash-<identification-string>
```

is created in the working directory where the program had been started. The crash file can then be used as a parameter to a new start of the fuzzer program – potentially in debug mode – and will use exactly the same data leading to the abort situation. The associated fuzzer call looks like

```
./<fuzzer-executable-name> crash-<identification-string>
```

Example 18. Suppose that the small library function `getMin()` introduced in Example 17 would be buggy as follows (we are sure that you can see the programming error at once!).

```
1  #include <limits.h>
2  #include "mylib.h"
3
4  int getMin(const int* buffer, size_t numIntElements) {
5    int idx;
6    int min = buffer[0];
7    for ( idx = 1; idx < numIntElements; idx++ ) {
8       if ( buffer[idx] < min ) min = buffer[idx];
9    }
10   return min;
11 }
```

We create the fuzzer program with command

```
clang -o getMinFuzzer \
      -g -O0 -fsanitize=fuzzer,address,signed-integer-overflow \
      testharness.c \
      mylib.c
```

Then the execution of the fuzzer with command

```
./getMinFuzzer
```

leads to a program abortion with outputs similar to

```
INFO: Seed: 1500934536
INFO: Loaded 1 modules   (13 inline 8-bit counters): 13 [0x10d7b35c8, 0x10d7b35d5),
INFO: Loaded 1 PC tables (13 PCs): 13 [0x10d7b35d8,0x10d7b36a8),
INFO: -max_len is not provided; libFuzzer will not generate inputs larger than 4096 bytes
==================================================================
==13089==ERROR: AddressSanitizer: heap-buffer-overflow
on address 0x602000000150 at pc 0x00010d770e06 bp 0x7ffee2490430 sp 0x7ffee2490428
READ of size 4 at 0x602000000150 thread T0
    #0 0x10d770e05 in getMin mylib.c:6
    #1 0x10d770e75 in LLVMFuzzerTestOneInput testharness.c:15
    . . .
==13089==ABORTING
. . .
artifact_prefix='./'; Test unit written to
./crash-da39a3ee5e6b4b0d3255bfef95601890afd80709
```

12.5.2 Replay Run to Found Bug

As you have guessed already, the error found refers to line 6 of the source code of `getMin()`: there, in the case that the buffer length is zero, the illegal

int-memory cell `buffer[0]` is accessed. If the situation is not quite clear to you, you can re-run the program in the debugger[3], using the crash-file. Of course, you need to use the LLVM debugger `lldb`, since the program has been compiled with `clang` – the `gdb` debugger would not work.

```
% lldb ./getMinFuzzer
(lldb) target create "./getMinFuzzer"
. . .
(lldb) b mylib.c:6
Breakpoint 1: where = getMinFuzzer`getMin + 36 at mylib.c:6:15, address = 0x0000000100001d04
(lldb) r ./crash-da39a3ee5e6b4b0d3255bfef95601890afd80709
Process 13136 launched:
. . .
Running: ./crash-da39a3ee5e6b4b0d3255bfef95601890afd80709
Process 13136 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
    frame #0: 0x0000000100001d04 getMinFuzzer
    `getMin(buffer=0x0000602000000190, numIntElements=0) at mylib.c:6:15 [opt]
   3
   4    int getMin(const int* buffer, size_t numIntElements) {
   5        int idx;
-> 6        int min = buffer[0];
   7        for ( idx = 1; idx < numIntElements; idx++ ) {
   8            if ( buffer[idx] < min ) min = buffer[idx];
   9        }
(lldb) p (size_t)malloc_size(buffer)
(size_t) $1 = 1
(lldb) n
=================================================================
==13136==ERROR: AddressSanitizer: heap-buffer-overflow
on address 0x602000000190 at pc 0x000100001e06 bp 0x7ffeefbff3d0 sp 0x7ffeefbff3c8
. . .
```

In this debugger run, we have set a breakpoint at the offending line 6 of the `getMin()` function and then activated the fuzzer executable with the crash file as parameter. Reaching the breakpoint, we can analyse the situation more closely: the debugger tells us that `getMin()` has been invoked with a non-null buffer pointer, but the value of parameter `numIntElements` is zero, indicating that `buffer` is too short to accommodate an int-value. Indeed, using the Libc function `malloc_size()`[4], we can print the actual length of the `buffer` (which was allocated on the heap by libFuzzer before calling `LLVMFuzzerTestOneInput()`), and we find out that only 1 byte had been allocated. Consequently, reading four bytes from the address of `buffer` in line 6 is a violation of heap memory boundaries. □

[3]Re-running a program with the same test data that has lead to an error before is often called **replay**.

[4]`size_t malloc_size(void *ptr)` is the function to use on Mac OS platforms; with Linux it's `size_t malloc_usable_size(void *ptr)`.

12.5.3 Using a Corpus

When performing coverage-guided random testing, libFuzzer is able to record information which input data to the SUT is useful to cover certain portions of code. If this information is available, later runs checking for different errors may re-use this to cover these code portions more quickly. The collection of information recorded from previous fuzzer runs is called **corpus**.

If the fuzzer executable is started like

```
./<fuzzer-executable-name> <name-of-existing-directory>
```

with name of an existing empty directory as first parameter, this directory is used to set up a corpus, consisting of different binary files containing coverage information.

When running the fuzzer program again (typically with other verification objectives specified as assertions as explained below), providing the name of corpus directory again has the effect that the new fuzzer run exploits the information stored there about how to reach certain portions of the code. Again, the name of the corpus directory must be given as first parameter.

Using a corpus usually speeds up the fuzzing process in a considerable way. It is, however, only useful if the SUT code remains mostly unchanged for the different fuzzer runs.

12.5.4 Useful Call Parameters

The generated fuzzer program (in our example, this was `getMinFuzz`) can be invoked with many different parameters; these are listed when giving command

```
./<fuzzer-executable-name>   -help=1
```

In the following paragraphs, we describe two very useful parameters, but make sure to inspect the full list, using the help command.

Setting a Timeout. As mentioned above, the fuzzer program runs "forever" when called without a parameter bounding its execution time, and if no crashes or termination commands are encountered. As long as errors are found quickly, this is not a problem. If the SUT is assumed to be correct (or "mostly correct"), however, it is useful to set time bounds making the fuzzer stop after a certain period of fruitless search for further bugs.

When calling the fuzzer program, the time bound is set with command

```
./<fuzzer-executable-name>  -max_total_time=<value-in-seconds> ...
```

The `<value-in-seconds>` needs to be positive, value zero is interpreted as "no time bound". Called like this, the fuzzer program will terminate after the specified value in seconds, unless the detection of an error leads to an earlier termination.

Setting a Seed. Without setting a specific parameter, the fuzzer program uses new random data on every run. To create the random data, however, pseudo random values are used. The seed for creating these values is printed as

```
INFO: Seed: <seed-value>
```

to standard error, when executing the fuzzer program. Calling the fuzzer with command

```
./<fuzzer-executable-name>  -seed=<seed-value> ...
```

leads to re-using the `<seed-value>`, so that a kind of replay is performed, including all the `LLVMFuzzerTestOneInput()` calls that did not lead to uncovering an error and ending at the last `LLVMFuzzerTestOneInput()` call where an error was uncovered or the execution time ran out.

12.5.5 Parallelisation for Finding Multiple Errors

The generated fuzzer program can be started in multi-processing mode, each process executing a copy of the fuzzer and the SUT and hunting for errors concurrently to the others. A job terminates if an error occurred. Meanwhile, the still living jobs continue operating. The multi-process version of the fuzzer program is activated by command

```
./<fuzzer-executable-name>  \
    -jobs=<number-of-jobs> -workers=<number-of-workers> ...
```

If `n = <number-of-workers>` is smaller that `<number-of-jobs>`, only `n` jobs will be executed at a time, the remaining jobs will be processed when a worker becomes free since its previous job has been completed.

The output produced by each job process is written to a file `fuzz-<job-number>.log`. All concurrent jobs use the same corpus, if such a directory exists.

This variant of multi-process execution of the fuzzer program is useful if

- the fuzzer is executed on a multi-core machine, and
- it is useful to hunt for different errors in a concurrent way.

Quite often, one would rather speed up the process for exploring the SUT and stop after having found just one error. This variant of parallelisation is described next.

12.5.6 Parallelisation for Speeding up Error Detection

For speeding up the detection of just one error, it is useful to start concurrent fuzzers as well. In contrast to the parallelisation discussed in the previous section, however, we wish to *terminate* the remaining processes as soon as the first has found an error. This feature is currently not supported by libFuzzer, but it is rather straightforward to implement.

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <stdlib.h>
4  #include <strings.h>
5  #include <signal.h>
6  #include <sys/time.h>
7  #include <sys/wait.h>
8
9  /**
10  * Main program for cuncurrent activation of a fuzzer program. It
11  * expects the following parameters.
12  *   @param argc     number of arguments passed to the main program,
13  *                   must be >= 3
14  *   @param argv[1] path to the fuzzer program to be started
15  *   @param argv[2] number of concurrent fuzzer processes
16  *                   to be activated
17  *   @param argv[3] first option to be passed on to each
18  *                   fuzzer process instance
19  *   @param argv[argc-1] last option to be passed on to each
20  *                       fuzzer process instance
21  *   Defining options is optionl.
22  */
23 int main(int argc, const char * argv[]) {
24
25     if ( argc < 3 ) exit(1);
26
27     int i;
28
```

```
29      const char* theProgramToStart = argv[1];
30      int numberOfCopies = atoi(argv[2]);
31      pid_t pidArray[numberOfCopies];
32
33      // Call the fuzzer with the following arguments:
34      // callArgs[0] name of fuzzer program
35      // callArgs[1..(argc-3)]  argv[3..(argc-1)] if argc > 3
36      // callArgs[argc-2] = NULL
37      char* callArgs[argc-1];
38
39      // First argument is the program name
40      callArgs[0] = strdup(theProgramToStart);
41
42      // Copy other options provided in argv[]
43      for ( i = 3; i < argc; i++ ) {
44          callArgs[i-2] = strdup(argv[i]);
45      }
46      // Terminate option array with NULL pointer
47      callArgs[i-2] = NULL;
48
49      // Start required number of fuzzer processes
50      for ( i = 0; i < numberOfCopies; i++ ) {
51          if ( 0 == (pidArray[i] = fork()) ) {
52              if ( execve(theProgramToStart,callArgs,NULL) < 0 ) {
53                  perror("execve");
54              }
55          }
56      }
57
58      // Wait for first fuzzer process to terminate
59      pid_t firstPid = wait(NULL);
60
61      // Kill all remaining fuzzer processes
62      for ( i = 0; i < numberOfCopies; i++ ) {
63          if ( pidArray[i] != firstPid ) {
64              if ( 0 == kill(pidArray[i],SIGKILL) ) {
65                  fprintf(stderr,"Killed %u\n",(unsigned)pidArray[i]);
66              }
67              else {
68                  perror("kill");
69              }
70          }
71      }
72      return 0;
73  }
```

Chapter 13

Property-Based Fuzz Testing

13.1 Property-Based Software Testing

Property-based testing (also called *property-oriended testing*) checks whether a violation of a desired system or software property can be uncovered by tests. Note that this is different from conformance testing discussed in the previous parts of these lecture notes, where *all* potential deviations of the SUT in relation to a reference model are investigated.

In the previous chapter, we have introduced fuzz testing with the LLVM libFuzzer and explained that – among other conditions – the fuzzing process always stops when an assertion fails. If we specify a desired *program property* in an assertion, we can obviously apply fuzz testing to property-based testing.

The standard approaches to property specifications in software testing are

- pre-conditions and post-conditions,
- invariant specifications,
- observers, and
- temporal logic specifications.

In the remainder of this chapter, we will introduce each of these concepts and explain how the can be applied using the LLVM libFuzzer.

13.2 Pre-Conditions and Post-Conditions

Recall that behaviour of terminating software operations can be specified by means of *pre-conditions* and *post-conditions*. The pre-condition specifies under which conditions concerning global variables, input parameters, and system resources (e.g. semaphores, available files, shared memory, ...) a well-defined result can be expected to be produced by the unit under test (UUT). The post-condition specifies the relation between program pre-state and program post-state, to be established by the UUT, provided that the pre-condition held at the point in time when the UUT was called.

Pre-conditions and post-conditions are Boolean conditions with global variables, operation parameters and return values, and system resource identifiers as free symbols. Representation formalisms for pre-and post-conditions are widely discussed in the literature. We use a very straight-forward approach that is most appreciated by practitioners:

> Pre-conditions and post-conditions are Boolean functions without side effects, evaluating input parameters, global variables and operating system resources that are defined in the scope where they are compiled.

This is best illustrated by an example

Example 19. Consider again the function `getMin()` used in the examples above.

```
1  #include <limits.h>
2  #include "mylib.h"
3
4  int getMin(const int* buffer, size_t numIntElements) {
5    if ( numIntElements == 0 ) return INT_MAX;
6    int idx;
7    int min = buffer[0];
8    for ( idx = 1; idx < numIntElements; idx++ ) {
9       if ( buffer[idx] < min ) min = buffer[idx];
10   }
11   return min;
12 }
```

A more systematic requirements analysis for this function leads to the following desired properties.

Prop. 1 The function shall return `INT_MAX` in the case of a zero-length buffer.

Prop. 2 The function shall never change the contents of the buffer[1]

Prop. 3 If the buffer has positive length, the function shall return a smallest element contained in the buffer.

As pre-condition, we require that buffer must be non-null.

These consideration lead to modified version of the fuzzer interface function which looks as follows.

```
1  #include <stdlib.h>
2  #include "mylib.h"
3
4  int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
5
6      // Don't call the UUT if precondition is not fulfilled
7      if ( ! preCnd(Data) ) return 0;
8
9      size_t idx;
10     // Adapt data to SUT input format
11     const int *buffer = (const int*)Data;
12     size_t numIntElements = Size/sizeof(int);
13
14     // Create a copy of the buffer -
15     // this is needed for checking property 2
16     int* bufferCopy = (int*)malloc(numIntElements*sizeof(int));
17     memcpy(bufferCopy, buffer, numIntElements*sizeof(int));
18
19     // Call the SUT
20     int result = getMin(buffer, numIntElements);
21
22     // Check the post-conditions, using C assertions
23     assert(property1(numIntElements, result));
24     assert(property2(buffer, numIntElements, bufferCopy));
25     assert(property3(buffer, numIntElements, result));
26     // Free the buffer copy
27     free(bufferCopy);
28     return 0;
29 }
```

[1]You might object that this property is not necessary, since `getMin()` declares `buffer` as a *constant* `int` pointer. Unfortunately, this `const` declaration does not protect against pointers of, for example, character type. If we let a `char` pointer `c` point to an address inside `buffer` and then assign a new value to `*c`, this will remain undetected at compile time, though it changes `buffer`. Therefore, Prop. 2 is well-justified and should always be checked for UUTs operating on a buffer that is not allowed to change.

As can be seen in line 7, the data provided by the fuzzer is discarded if the pre-condition (applied by a call to function `preCnd()`) is not fulfilled. As the only input parameter, the pre-condition function needs the `Data` input, so that it can check this parameter to be non-NULL. Note that we do not use a C-assertion here, because the test should not be aborted if the pre-condition is not fulfilled: if `!preCnd(Data)` holds, this just implies that we cannot work with this data on the UUT, but this is not the UUT's fault.

The three post-condition properties, however, are checked using C-assertions calling functions `property1,2,3()` in lines 23—25. The parameters used by these functions depend (of course) on the symbols needed to check the respective property.

- Prop. 1 just needs the buffer length to decide whether `INT_MAX` should be returned and the actual return value to check whether it contains this value if the buffer length is zero.

- Prop. 2 is more subtle: it needs a copy of the original buffer passed on to the UUT, so that it can check whether the original buffer has not been changed by `getMin()`. Consequently, original buffer, its length, and the buffer copy (which has the same length) are passed to `property2()`. The buffer copy is created before the UUT is called in lines 16—17. Note that the buffer copy needs to be freed in line 27, because otherwise your test may fail with an 'out-of-memory' failure which is not caused by the UUT, but by the test engineer inducing a memory leak in the test harness.[2]

- Prop. 3 checks the main requirement. To this end, the buffer, its length, and the return value are need. Based on these data items, the function can check whether the UUT really returns a smallest element from the buffer. Note that this function has nothing to check if the buffer length is zero.

The implementation of the condition functions should be inserted in to the test harness. They are shown in the following listing. Please make sure that you understand why the condition checking code of each function is adequate!

[2]This would lead to a so-called *false positive*: the UUT is ok, but the test environment is buggy. Therefore the test fails, and the test engineers lose their street credibility.

```
1  static int preCnd(const uint8_t* Data) {
2      return (Data != NULL);
3  }
4
5  static int property1(size_t numIntElements,int result) {
6      if ( numIntElements > 0 ) return 1;
7      return (result == INT_MAX);
8  }
9
10 static int property2(const int *buffer,
11                       size_t numIntElements,
12                       const int *bufferCopy) {
13     size_t i;
14     for ( i = 0; i < numIntElements; i++ ) {
15         if ( buffer[i] != bufferCopy[i] ) return 0;
16     }
17     return 1;
18 }
19
20 static int property3(const int *buffer,
21                       size_t numIntElements,
22                       int result) {
23     if ( numIntElements == 0 ) return 1;
24     size_t i;
25     int haveSeenResult = 0;
26     for ( i = 0; i < numIntElements; i++ ) {
27         if ( buffer[i] == result ) haveSeenResult = 1;
28         if ( buffer[i] < result ) return 0;
29     }
30     return haveSeenResult;
31 }
```

□

13.3 Invariants

Recall that *invariants* are Boolean conditions that are required to hold always at the end of logically coherent transactions or at the beginning of each loop body executed during an iteration. Further invariants specify integrity constraints of structured data. The symbols occurring free in invariants used for software testing comprise those occurring in pre-conditions and post-conditions. However, it is often the case that invariants need to be evaluated *inside* the UUT code, since they also refer to symbols (local variables, static

functions) that would be out of scope before the UUT is called or after it has returned.

As a consequence, invariants are typically specified as C-assertions in the UUT code itself. After testing, the C-assertions can be removed at compile time without changing the UUT code: setting the option `-DNDEBUG` (this sets the "no debug" define) instructs the compiler to not produce any assertion code. As a consequence, it is admissible to have assertions inside C production source code.

Example 20. Consider again the function `getMin()` used in the examples above. We can define an invariant that should hold at the beginning of each loop cycle performed by `getMin()`:

> **Inv. 1**. The value of local variable `min` must be a minimum of the buffer section `buffer[0..idx-1]`.

This invariant is implemented in the following UUT file.

```
1  #include <limits.h>
2  #include "mylib.h"
3
4  // Only for invariant testing
5  #ifndef NDEBUG
6  int inv1(const int* buffer, int idx, int min) {
7     int i;
8     int haveSeenMin = 0;
9     for ( i = 0; i < idx; i++ ) {
10       if ( buffer[i] == min ) haveSeenMin = 1;
11       if ( buffer[i] < min ) return 0;
12    }
13    return haveSeenMin;
14 }
15 #endif
16
17 int getMin(const int* buffer, size_t numIntElements) {
18   if ( numIntElements == 0 ) return INT_MAX;
19   int idx;
20   int min = buffer[0];
21   for ( idx = 1; idx < numIntElements; idx++ ) {
22      assert(inv1(buffer,idx,min));
23      if ( buffer[idx] < min ) min = buffer[idx];
24   }
25   return min;
26 }
```

□

Chapter 14

Coverage Analysis

This chapter is new in Issue 5.1.

14.1 Objectives and Limitations of Coverage Analysis

When fuzz testing no longer discovers any errors in the code, it is time to decide whether we have tested enough. The complete testing methods discussed in other parts of the document are able to give mathematically proven guarantees that we have tested enough after having passed a complete test suite, provided that the hypotheses regarding the SUT's number of internal states and granularity of guard conditions are correct.

For fuzz testing, such guarantees cannot be given, since we do not have a formal reference model of the code's expected behaviour. But at least, we can check whether the fuzz tests executed so far have covered all code portions of interest. While this is definitely better than nothing, please keep in mind that complete code coverage is worthless if we did not insert correct and complete assertions about post conditions and invariants into the code.

The LLVM framework offers branch coverage analysis: it is recorded which arcs of the code's control flow graph have been covered. This coverage metric is adequate for standard applications. For safety-critical applications, however, this is insufficient. For avionic software of design assurance level A, for example, software tests need to achieve MC/DC coverage [70].

14.2 Compile options for Fuzzing With Code Coverage Profiling

When intending to measure the branch coverage achieved during fuzz testing, two additional compile options

```
-fprofile-instr-generate -fcoverage-mapping
```

are required, so that the complete compile directive now looks as follows.

```
clang -o <fuzzer-executable-name> \
      -g -O0 -fsanitize=fuzzer,address,signed-integer-overflow \
      -fprofile-instr-generate -fcoverage-mapping \
      <further options> \
      <testharness>.c \
      <sut-file1>.c <sut-file2>.c ...
```

The generated executable is now able to record coverage profiling data while running, provided that the environment variable `LLVM_PROFILE_FILE` has been associated with file name for the profiling raw data.

Command

```
LLVM_PROFILE_FILE="<file>.profraw" \
  ./<fuzzer-executable-name> -max_total_time=<duration>
```

will now result in a fuzzer execution of the UUT for `<duration>` seconds. After successful termination, coverage raw data will be written into binary file `<file>.profraw`.

The raw data file is pre-processed and merged with other data by means of command

```
llvm-profdata merge -sparse <file>.profraw -o <file>.profdata
```

The the coverage achieved is displayed using command

```
llvm-cov show ./<fuzzer-executable-name> \
              -instr-profile=<file>.profdata
```

Note that the coverage recording fails if the fuzzer aborts due to a crash or failed assertion. In such a case, the bug causing the crash first has to

be fixed. Afterwards, the test can be re-run using the crash-file. If the bug fix was successful, the run should now terminate without crashing, and the coverage achieved by this test can be analysed.

Example 21. Consider yet another faulty variant of the UUT `getMin()` discussed in the examples above.

```
1  #include <limits.h>
2  #include "mylib.h"
3
4  int getMin(const int* buffer, size_t numIntElements) {
5      if ( numIntElements == 0 ) return INT_MAX;
6      int idx;
7      int min = buffer[0];
8      for ( idx = 1; idx < numIntElements; idx++ ) {
9          if ( buffer[idx] < min ) min = buffer[idx];
10     }
11     // This is a deliberate bug which corrupts
12     // the input buffer if it is longer than 500 int's.
13     // The cell buffer[400] is corrupted.
14     if ( numIntElements > 500 ) {
15         char* c = (char*)(buffer + 400);
16         *c = 17;
17     }
18     return min;
19 }
```

It contains a bug of the *"time bomb"* type in lines 14 — 17 which might have been inserted by a malicious programmer. The bug exploits the C language specification which states that `char*`-pointers may be placed anywhere, even into buffers declared as `const` as is the case in this example. The compiler will accept this code without warning. The bug only occurs if the buffer is longer than 500 words, and then it will corrupt the input buffer by assigning a constant byte-value to some byte address inside the buffer. The bug has time bomb character, since it does not occur in every run.

This malicious bug is a good example for using parallelisation in fuzz testing, as described in Section 12.5.6: it takes just about one minute or even less to uncover it, if we start 8 concurrent copies of the fuzzer.

Based on the violation of post-condition `property2()` occurring in the fuzzer run, it is fairly straightforward to uncover the cause of the bug by replaying the crash-file in the debugger. Suppose we use a minimal fix for this bug by just commenting out the offending line 16 which corrupts the buffer:

```
1  #include <limits.h>
2  #include "mylib.h"
3
4  int getMin(const int* buffer, size_t numIntElements) {
5      if ( numIntElements == 0 ) return INT_MAX;
6      int idx;
7      int min = buffer[0];
8      for ( idx = 1; idx < numIntElements; idx++ ) {
9          if ( buffer[idx] < min ) min = buffer[idx];
10     }
11     // This is a deliberate bug which corrupts
12     // the input buffer if it is longer than 500 int's.
13     // The cell buffer[400] is corrupted.
14     if ( numIntElements > 500 ) {
15         char* c = (char*)(buffer + 400);
16     // @bugfix     *c = 17;
17     }
18     return min;
19 }
```

If we now-recompile the executable with code coverage profiling enabled and replay the test case using the crashfile, the formatted coverage created by this test alone is shown in Fig. 14.1.

We expect that there are no further bugs to be found in this function. The coverage information, however, shows that the

```
1    return INT_MAX;
```

statement in line 5 has not been executed yet (it is therefore marked in red). Therefore we start the fuzzer again with a time limit of 60s. The accumulated coverage displayed in Fig. 14.2 shows that complete branch coverage has been achieved now.

Since no further bugs have come up and the branch coverage is complete, we stop testing. □

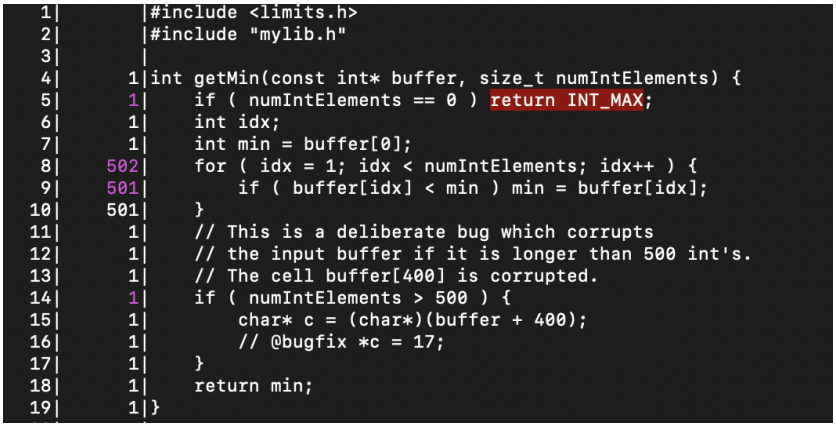


Figure 14.1: Code coverage achieved by replay of crash file for bug from line 16 of UUT `getMin()`.

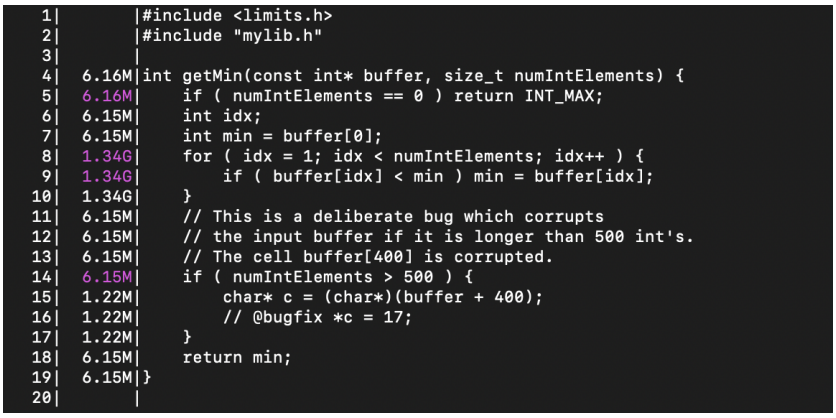


Figure 14.2: Code coverage achieved by an additional fuzz test lasting one minute.

Bibliography

[1] Rui Abreu and Arjan J. C. van Gemund. A low-cost approximate minimal hitting set algorithm and its application to model-based diagnosis. In Vadim Bulitko and J. Christopher Beck, editors, *Eighth Symposium on Abstraction, Reformulation, and Approximation, SARA 2009, Lake Arrowhead, California, USA, 8-10 August 2009*. AAAI, 2009.

[2] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John A. Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil McMinn. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[3] Andrea Arcuri, Muhammad Zohaib Iqbal, and Lionel Briand. Black-box system testing of real-time embedded systems using random and search-based testing. In *Proceedings of the 22nd IFIP WG 6.1 international conference on Testing software and systems*, ICTSS'10, pages 95–110, Berlin, Heidelberg, 2010. Springer-Verlag.

[4] Cécile Braunstein, Anne E. Haxthausen, Wen-ling Huang, Felix Hübner, Jan Peleska, Uwe Schulze, and Linh Vu Hong. Complete model-based equivalence class testing for the ETCS ceiling speed monitor. In S. Merz and J. Pang, editors, *Proceedings of the ICFEM 2014*, number 8829 in Lecture Notes in Computer Science, pages 380–395. Springer Berlin Heidelberg, November 2014.

[5] Cécile Braunstein, Wen-ling Huang, Jan Peleska, Uwe Schulze, Felix Hübner, Anne E. Haxthausen, and Linh Vu Hong. A SysML test model and test suite for the ETCS ceiling speed monitor. Technical report, Embedded Systems Testing Benchmarks Site, 2014-04-30. Available under `http://www.mbt-benchmarks.org`.

[6] Ana Cavalcanti and Marie-Claude Gaudel. Testing for refinement in *circus*. *Acta Inf.*, 48(2):97–147, 2011.

[7] Ana Cavalcanti, Wen-ling Huang, Jan Peleska, and Jim Woodcock. CSP and kripke structures. In Martin Leucker, Camilo Rueda, and Frank D. Valencia, editors, *Theoretical Aspects of Computing - ICTAC 2015 - 12th International Colloquium Cali, Colombia, October 29-31, 2015, Proceedings*, volume 9399 of *Lecture Notes in Computer Science*, pages 505–523. Springer, 2015.

[8] T. Y. Chen, Fei-Ching Kuo, Robert G. Merkel, and T. H. Tse. Adaptive random testing: the art of test case diversity. *JOURNAL OF SYSTEMS AND SOFTWARE*, 83(1):60–66, 2010.

[9] Tsun S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–186, March 1978.

[10] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[11] Razvan Diaconescu. *Institution-independent Model Theory*. Birkhäuser Verlag AG, Basel, Boston, Berlin, 2008.

[12] Jeremy Dick and Alain Faivre. Automating the generation and sequencing of test cases from model-based specifications. In James C. P. Woodcock and Peter G. Larsen, editors, *FME '93: Industrial-Strength Formal Methods*, number 670 in Lecture Notes in Computer Science, pages 268–284. Springer Berlin Heidelberg, January 1993.

[13] Rita Dorofeeva, Khaled El-Fakih, and Nina Yevtushenko. An improved conformance testing method. In Farn Wang, editor, *Formal Techniques for Networked and Distributed Systems - FORTE 2005, 25th IFIP WG 6.1 International Conference, Taipei, Taiwan, October 2-5, 2005, Proceedings*, volume 3731 of *Lecture Notes in Computer Science*, pages 204–218. Springer, 2005.

[14] E.J. McCluskey Jr. Minimization of boolean functions. *Bell System Technical Journal*, 35(6):1417–1444, 1956.

[15] André Takeshi Endo and Adenilso da Silva Simão. Experimental comparison of test case generation methods for finite state machines. In Giuliano Antoniol, Antonia Bertolino, and Yvan Labiche, editors, *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012*, pages 549–558. IEEE, 2012.

[16] European Committee for Electrotechnical Standardization. *EN 50128 – Railway applications – Communications, signalling and processing systems – Software for railway control and protection systems*. CENELEC, Brussels, 2001.

[17] European Railway Agency. ERTMS – System Requirements Specification – UNISIG SUBSET-026, February 2012. Available under http://www.era.europa.eu/Document-Register/Pages/Set-2-System-Requirements-Specification.aspx.

[18] Lars Frantzen, Jan Tretmans, and TimA.C. Willemse. Test generation based on symbolic specifications. In Jens Grabowski and Brian Nielsen, editors, *Formal Approaches to Software Testing*, volume 3395 of *Lecture Notes in Computer Science*, pages 1–15. Springer Berlin Heidelberg, 2005.

[19] S. Fujiwara, G. v Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, 1991.

[20] Marie-Claude Gaudel. Testing can be formal, too. In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach, editors, *TAPSOFT*, volume 915 of *Lecture Notes in Computer Science*, pages 82–96. Springer, 1995.

[21] Arthur Gill. *Introduction to the theory of finite-state machines*. McGraw-Hill, New York, 1962.

[22] Joseph A. Goguen and Rod M. Burstall. Institutions: Abstract Model Theory for Specification and Programming. *J. ACM*, 39(1):95–146, January 1992.

[23] Wolfgang Grieskamp, Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Generating finite state machines from abstract state machines. *ACM SIGSOFT Software Engineering Notes*, 27(4):112–122, July 2002.

[24] Steffen Helke, Thomas Neustupny, and Thomas Santen. Automating test case generation from z specifications with isabelle. In Jonathan P. Bowen, Michael G. Hinchey, and David Till, editors, *ZUM '97: The Z Formal Specification Notation*, number 1212 in Lecture Notes in Computer Science, pages 52–71. Springer Berlin Heidelberg, January 1997.

[25] Matthew Hennessy. *Algebraic Theory of Processes*. MIT Press, Cambridge, MA, USA, 1988.

[26] Rob M. Hierons. Testing from a nondeterministic finite state machine using adaptive state counting. *IEEE Transactions on Computers*, 53(10):1330–1342, 2004.

[27] C. A. R. Hoare and H. Jifeng. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[28] Hyoung Seok Hong, Insup Lee, Oleg Sokolsky, and Hasan Ural. A temporal logic based theory of test coverage and generation. In Joost-Pieter Katoen and Perdita Stevens, editors, *TACAS*, volume 2280 of *Lecture Notes in Computer Science*, pages 327–341. Springer, 2002.

[29] Wen-ling Huang, Sadik Özoguz, and Jan Peleska. Safety-complete test suites. *Software Quality Journal*, Oct 2018. https://doi.org/10.1007/s11219-018-9421-y.

[30] Wen-ling Huang and Jan Peleska. Exhaustive model-based equivalence class testing. In Hüsnü Yenigün, Cemal Yilmaz, and Andreas Ulrich, editors, *Testing Software and Systems*, volume 8254 of *Lecture Notes in Computer Science*, pages 49–64. Springer Berlin Heidelberg, 2013.

[31] Wen-ling Huang and Jan Peleska. Complete model-based equivalence class testing. *STTT*, 18(3):265–283, 2016.

[32] Wen-ling Huang and Jan Peleska. Complete model-based equivalence class testing for nondeterministic systems. *Formal Aspects of Computing*, pages 1–30, 2016. Available under `http://dx.doi.org/10.1007/s00165-016-0402-2`.

[33] Wen-ling Huang and Jan Peleska. Safety-complete test suites. In Nina Yevtushenko, Ana Rosa Cavalli, and Hüsnü Yenigün, editors, *Testing Software and Systems - 29th IFIP WG 6.1 International Conference, ICTSS 2017, St. Petersburg, Russia, October 9-11, 2017, Proceedings*, volume 10533 of *Lecture Notes in Computer Science*, pages 145–161. Springer, 2017.

[34] Felix Hübner, Wen-ling Huang, and Jan Peleska. Experimental evaluation of a novel equivalence class partition testing strategy. In Jasmin Christian Blanchette and Nikolai Kosmatov, editors, *Tests and Proofs - 9th International Conference, TAP 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 22-24, 2015. Proceedings*, volume 9154 of *Lecture Notes in Computer Science*, pages 155–172. Springer, 2015.

[35] ISO/IEC/IEEE 29119-1:2013(e): Software and systems engineering – software testing – part 1: Concepts and definitions, September 2013.

[36] ISO/IEC/IEEE 29119-2:2013(e): Software and systems engineering – software testing – part 2: Test processes, September 2013.

[37] ISO/IEC/IEEE 29119-3:2013(e): Software and systems engineering – software testing – part 3: Test documentation, September 2013.

[38] ISO/IEC/IEEE DIS 29119-4.2: Software and systems engineering – software testing – part: 4 test techniques, February 2014.

[39] Paul C. Jorgensen. *The Craft of Model-Based Testing*. CRC Press, Boca Raton, 2017.

[40] Abdul Salam Kalaji, Robert M. Hierons, and Stephen Swift. Generating feasible transition paths for testing from an extended finite state machine (efsm). In *ICST*, pages 230–239. IEEE Computer Society, 2009.

[41] R. Karp. Reducibility among combinatorial problems. In R.E. Miller and J.W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York NY, 1972.

[42] Donald E. Knuth. *The Art of Computer Programming – Volume 1 – Fundamental Algorithms*. Addison Wesley Longman, Reading, MA, 1997.

[43] Florian Lapschies. SONOLAR homepage, June 2014. http://www.informatik.uni-bremen.de/agbs/florian/sonolar/.

[44] Florian Lapschies. *The SONOLAR SMT Solver*. PhD thesis, University of Bremen, 2014. To be submitted in November 2016.

[45] G. Luo, G.V. Bochmann, and A. Petrenko. Test selection based on communicating nondeterministic finite-state machines using a generalized wp-method. *IEEE Transactions on Software Engineering*, 20(2):149–162, 1994.

[46] Till Mossakowski and Markus Roggenbach. Structured CSP - A process algebra as an institution. In José Luiz Fiadeiro and Pierre-Yves Schobbens, editors, *Recent Trends in Algebraic Development Techniques, 18th International Workshop, WADT 2006, La Roche en Ardenne, Belgium, June 1-3, 2006, Revised Selected Papers*, volume 4409 of *Lecture Notes in Computer Science*, pages 92–110. Springer, 2006.

[47] S. Naito and M. Tsunoyama. Fault detection for sequential machines by transition tours. In *Proc. IEEE Fault Tolerant Comput. Conf.*, pages 162–178, 1981.

[48] Object Management Group. OMG Systems Modeling Language (OMG SysML), Version 1.4. Technical report, Object Management Group, 2015. http://www.omg.org/spec/SysML/1.4.

[49] J. Peleska and M. Siegel. Test automation of safety-critical reactive systems. *South African Computer Jounal*, 19:53–77, 1997.

[50] Jan Peleska. *Formal Methods and the Development of Dependable Systems*. Number 9612 in Institut für Informatik und Praktische Mathematik, Berichte. Christian-Albrechts-Universität Kiel, December 1996. Habilitationsschrift.

[51] Jan Peleska. Industrial-strength model-based testing - state of the art and current challenges. In Alexander K. Petrenko and Holger Schlingloff, editors, Proceedings Eighth Workshop on *Model-Based Testing,* Rome, Italy, 17th March 2013, volume 111 of *Electronic Proceedings in Theoretical Computer Science*, pages 3–28. Open Publishing Association, 2013.

[52] Jan Peleska, Wen-ling Huang, and Felix Hübner. A novel approach to HW/SW integration testing of route-based interlocking system controllers. In Thierry Lecomte, Ralf Pinger, and Alexander Romanovsky, editors, *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification - First International Conference, RSSRail 2016, Paris, France, June 28-30, 2016, Proceedings*, volume 9707 of *Lecture Notes in Computer Science*, pages 32–49. Springer, 2016.

[53] Jan Peleska, Elena Vorobev, and Florian Lapschies. Automated test case generation with SMT-solving and abstract interpretation. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *Nasa Formal Methods, Third International Symposium, NFM 2011*, volume 6617 of *LNCS*, pages 298–312, Pasadena, CA, USA, April 2011. Springer.

[54] A. Petrenko and N. Yevtushenko. Adaptive testing of deterministic implementations specified by nondeterministic fsms. In *Testing Software and Systems*, number 7019 in Lecture Notes in Computer Science, pages 162–178, Berlin, Heidelberg, 2011. Springer.

[55] A. Petrenko and N. Yevtushenko. Adaptive testing of nondeterministic systems with fsm. In *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering (HASE)*, pages 224–28, 2014.

[56] A. Petrenko, N. Yevtushenko, and G. v. Bochmann. Fault models for testing in context. In Reinhard Gotzhein and Jan Bredereke, editors, *Formal Description Techniques IX – Theory, application and tools*, pages 163–177. Chapman&Hall, 1996.

[57] A. Petrenko, N. Yevtushenko, and G. V. Bochmann. Testing deterministic implementations from nondeterministic FSM specifications. In *In Testing of Communicating Systems, IFIP TC6 9th International Workshop on Testing of Communicating Systems*, pages 125–141. Chapman and Hall, 1996.

[58] Alexandre Petrenko and Nina Yevtushenko. Conformance tests as checking experiments for partial nondeterministic FSM. In Wolfgang Grieskamp and Carsten Weise, editors, *Formal Approaches to Software Testing, 5th International Workshop, FATES 2005, Edinburgh, UK, July*

11, 2005, Revised Selected Papers, volume 3997 of *Lecture Notes in Computer Science*, pages 118–133. Springer, 2005.

[59] Alexandre Petrenko and Nina Yevtushenko. Adaptive testing of nondeterministic systems with FSM. In *15th International IEEE Symposium on High-Assurance Systems Engineering, HASE 2014, Miami Beach, FL, USA, January 9-11, 2014*, pages 224–228. IEEE Computer Society, 2014.

[60] Alexandre Petrenko, Nina Yevtushenko, Alexandre Lebedev, and Anindya Das. Nondeterministic state machines in protocol conformance testing. In Omar Rafiq, editor, *Protocol Test Systems, VI, Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test systems, Pau, France, 28-30 September, 1993*, volume C-19 of *IFIP Transactions*, pages 363–378. North-Holland, 1993.

[61] W. V. Quine. The problem of simplifying truth functions. *The American Mathematical Monthly*, 59(8):521–531, 1952.

[62] A. W. Roscoe. *Understanding Concurrent Systems*. Springer, London, Dordrecht Heidelberg, New York, 2010.

[63] Adenilso Simão, Alexandre Petrenko, and Nina Yevtushenko. On reducing test length for FSMs with extra states. *Software Testing, Verification and Reliability*, 22(6):435–454, September 2012.

[64] Andreas Spillner, Tilo Linz, and Hans Schaefer. *Software Testing Foundations*. dpunkt.verlag, Heidelberg, 2006.

[65] J.G. Springintveld, F.W. Vaandrager, and P.R. D'Argenio. Testing timed automata. *Theoretical Computer Science*, 254(1-2):225–257, March 2001.

[66] P. H. Starke. *Abstract Automata*. Elsevier, North-Holland, Amsterdam, 1972.

[67] Jan Tretmans. Conformance testing with labelled transition systems: Implementation relations and test generation. *Computer Networks and ISDN Systems*, 29(1):49–79, 1996.

[68] UNISIG, editor. *ERTMS/ETCS System Requirements Specification, Chapter 3, Principles*, volume Subset-026-3, chapter 3. UNISIG, February 2012. Issue 3.3.0.

[69] M. P. Vasilevskii. Failure diagnosis of automata. *Kibernetika (Transl.)*, 4:98–108, July-August 1973.

[70] RTCA SC-205/EUROCAE WG-71. Software Considerations in Airborne Systems and Equipment Certification. Technical Report RTCA/DO-178C, RTCA Inc, 1140 Connecticut Avenue, N.W., Suite 1020, Washington, D.C. 20036, December 2011.

Appendix A

Algorithms for Solving the Minimal Hitting Set Problem

A.1 Problem Statement

Let (W, S) be a *set system* consisting of a *universe* W (which is just a finite set in our case) and a collection $S = \{z_1, \ldots, z_n\}$ of subsets of W, that is, $\forall i \in \{1, \ldots, n\} : z_i \subseteq W$. The *minimal hitting set (MHS)* problem is defined as the task to identify a subset $H \subseteq W$ such that

1. $\forall i \in \{1, \ldots, n\} : z_i \cap H \neq \varnothing$
2. $\forall H' \subseteq H : (H' \neq H \Rightarrow \exists i \in \{1, \ldots, n\} : z_i \cap H' = \varnothing)$

Intuitively speaking, a solution H of the MHS problem has a non-empty intersection with every set z_i contained in the collection S, and H cannot be further reduced without violating this property.

Since we are dealing with finite universes only, we can assume that W consists of the natural numbers $\{1, \ldots, m\}$, and the z_i are always subsets of

$\{1, \ldots, m\}$.

Example 22. Let $W = \{1, \ldots, 10\}$ and $S = \{z_1, \ldots, z_5\}$ with

$$\begin{aligned} z_1 &= \{1, 2, 3\} \\ z_2 &= \{4\} \\ z_3 &= \{1, 4\} \\ z_4 &= \{3, 5\} \\ z_5 &= \{3, 5, 6\} \end{aligned}$$

Then $\{1, 2, 4, 5\}$ is a non-minimal hitting set of the set system (W, S), because it can be reduced to, for example, $\{1, 4, 5\}$ without loosing the hitting set property. The sets $\{1, 4, 5\}$ and $\{3, 4\}$ are minimal subsets; this shows that MHSs may have different cardinality. □

A.2 A Simple Complete Algorithm for Determining Minimal Hitting Sets With Minimal Cardinality

For the purpose of creating state identification sets as described in Section 3.7, we have seen there in Algorithm 2 that this task can be formulated as an MHS problem. It is desirable, however, to identify the smallest state identification sets possible; this can be rephrased as the problem to find an MHS with smallest cardinality. To identify such an MHS, *all* MHSs have to be determined. Unfortunately, this is an NP-hard problem [41].

Experience shows, however, that in practical applications the characterisation sets W have rather small sizes, so that an explicit enumeration of all MHSs is possible. The following algorithm describes how such an enumeration can be performed. If this leads to complexity problems, one of the many existing approximation algorithms should be applied instead; we name [1] as an example.

1. **Input.** Set system (U, S).
2. **Output.** MHS H with minimal cardinality.

3. Initialise $\mathsf{H} := \bigcup_{z \in \mathsf{S}} z$. Obviously, H has a non-empty intersection with every element of S, so it is a hitting set.

4. Initialise tree with root $\mathsf{r} := \mathsf{H}$.

5. Set current tree node $\mathsf{c} := \mathsf{r}$.

6. While the tree can still be expanded, proceed recursively as follows.

   (a) For every $\mathsf{a} \in \mathsf{c}$, proceed as follows.

      i. Set $\mathsf{c}' := \mathsf{c} - \{\mathsf{a}\}$.

      ii. If $\forall z \in \mathsf{S} : \mathsf{c}' \cap z \neq \varnothing$ do

         - Extend tree by making c' a child node of c.
         - Set $\mathsf{c} := \mathsf{c}'$.
         - If $|\mathsf{c}| < |\mathsf{H}|$, set $\mathsf{H} := \mathsf{c}$.
         - Continue recursively with the new value of c at Step 6a (the loop for the old node c will be continued after the subtree of the new node c has been completely constructed).

      iii. Otherwise continue loop at Step 6a (the tree remains unchanged, c remains unchanged).

   (b) After the loop in Step 6a has been completed for a given node c: If a parent node exists, set $\mathsf{c} :=$ 'parent node' and continue with loop in Step 6a of the parent node (which has already been started before recursively commencing the loop over the old value of c). Otherwise continue with Step 7.

7. Return H.

Example 23. Consider again the hitting set problem from Example 22. Applying the algorithm above, the tree depicted in two parts in Fig. A.1 and Fig. A.2 is generated. Scanning the leaves of the tree shows that the minimal hitting sets are

$$\{3,4\},\{1,4,5\},\{2,4,5\},$$

and the minimal hitting set with minimal cardinality is $\{3,4\}$. □

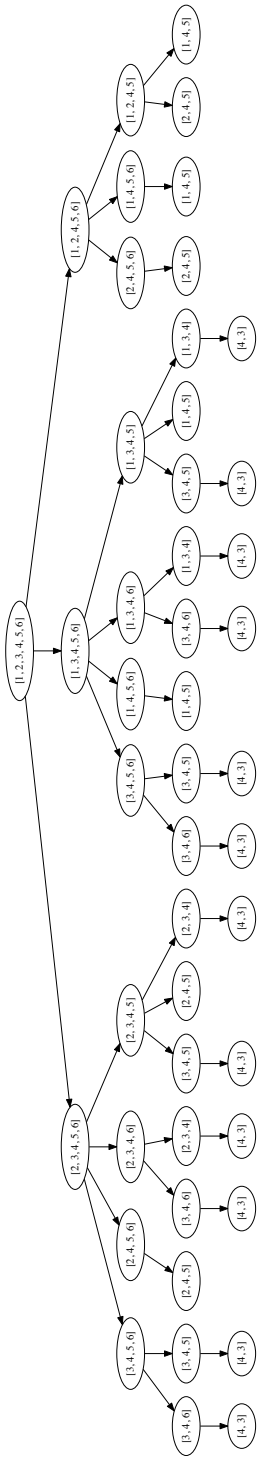


Figure A.1: First part of tree constructed by the algorithm above for solving the minimal hitting set problem from Example 22.

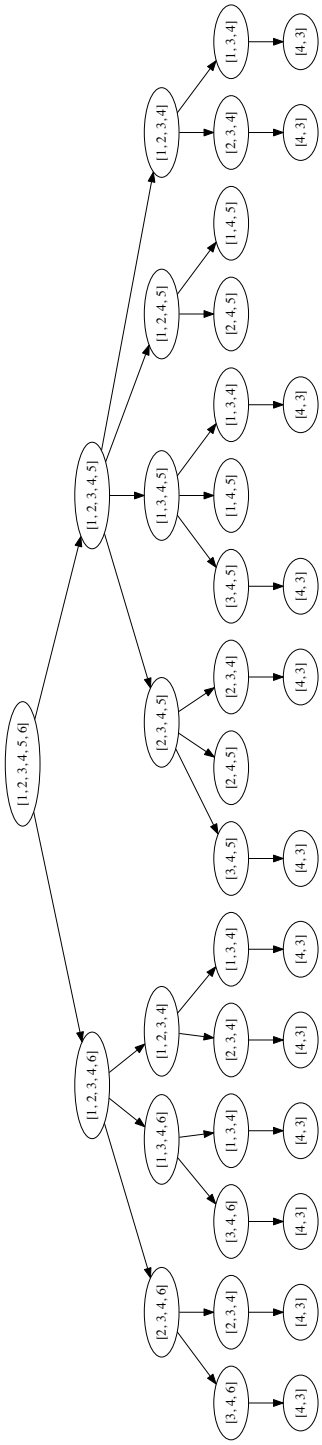


Figure A.2: Second part of tree constructed by the algorithm above for solving the minimal hitting set problem from Example 22.

A.3 The MHS Problem Re-Formulated as a Minimal SAT Model Problem

Recall that the task to find a Boolean valuation function $s : V \to \mathbb{B}$ on variables $V = \{x_1, \ldots, x_n\}$, that is a model for a given Boolean formula φ with free variables in V is called the SAT problem. Being a model for a formula is written as $s \models \varphi$ and defined by

$$s \models \varphi \equiv \varphi[s(x_1)/x_1, \ldots, s(x_n)/x_n].$$

This means that replacing every free variable x_i in φ by its Boolean value assignment $s(x_i)$ results in a term that is equivalent to `true`.

The *Minimal SAT Model Problem* tries to find a minimal subset $A \subseteq V$ and a valuation function $u : A \to \mathbb{B}$ such that u is already a model for φ, regardless of the assignments to the variables in $V - A$. This means that

$$\varphi[s(x)/x \mid x \in A]$$

is a tautology, i.e. it evaluates to `true` in every possible valuation of the remaining variables from $V - A$ occurring on φ.

Let us now reconsider the minimal hitting set problem and encode it as a minimal SAT model problem. Let (W, S) be a set system with universe $W = \{1, \ldots, m\} \subseteq \mathbb{N}$ and $S = \{z_1, \ldots, z_n\}$ with $z_i \subseteq W$ for $i = 1, \ldots, n$. Let $V = \{x_1, \ldots, x_m\}$ be a set of Boolean variable symbols. Any subset $M \subseteq W$ induces a valuation function $s_M : V \to \mathbb{B}$ defined by

$$\forall i \in \{1, \ldots, m\} : s_M(x_i) \Leftrightarrow (i \in M).$$

The hitting set problem

$$\text{Find } H \subseteq W \text{ such that} \quad \forall i \in \{1, \ldots, n\} : z_i \cap H \neq \varnothing$$

can now be re-formulated as the SAT problem in conjunctive form

$$\bigwedge_{i=1}^{n} \bigvee_{j \in z_i} x_j$$

It is easy to see that finding a *minimal* hitting set corresponds to finding a minimal SAT model $u : A \to \mathbb{B}, A \subseteq V$ for φ.

Example 24. Consider again the MHS problem from Example 22 with

$W = \{1, \ldots, 10\}$ and $S = \{z_1, \ldots, z_5\}$ and

$$\begin{aligned} z_1 &= \{1, 2, 3\} \\ z_2 &= \{4\} \\ z_3 &= \{1, 4\} \\ z_4 &= \{3, 5\} \\ z_5 &= \{3, 5, 6\} \end{aligned}$$

Setting $V = \{x_1, \ldots, x_{10}\}$, the hitting set problem can be re-formulated as the SAT problem

$$\begin{aligned} &\varphi \equiv \\ &(x_1 \vee x_2 \vee x_3) \wedge \\ &(x_4) \wedge \\ &(x_1 \vee x_4) \wedge \\ &(x_3 \vee x_5) \wedge \\ &(x_3 \vee x_5 \vee x_6) \end{aligned}$$

Any valuation $s : V \to \mathbb{B}$ satisfying, for example,

$$s(x_1) = s(x_2) = s(x_4) = s(x_5) = \texttt{true}$$

is a model for φ, and this is equivalent to the fact that $H = \{1, 2, 4, 5\}$ is a hitting set for (W, S).

The valuation function $u = \{x_3 \mapsto \texttt{true}, x_4 \mapsto \texttt{true}\}$, for example, is a minimal model for φ, and this corresponds to the minimal hitting set $H = \{3, 4\}$. □

For finding minimal SAT models, the well-known algorithm of Quine and McClusky [61, 14] can be used, for example. It guarantees to find one minimal model, but does not enumerate all minimal models. As a consequence, it is not guaranteed that the solution also has minimal cardinality. On the other hand, the algorithm will generally be significantly faster and need less memory than the complete one presented above.

Appendix B

Introduction to the FSM Library

B.1 Overview

The *Finite State Machine Library (FSM Library)* is a class library containing essential algorithms for manipulating finite state machines and applying model-based test generation methods as described in Part II of these lecture notes. Moreover, the FSM Library contains an executable for test generation by W-Method or Wp-Method. Finally, the library provides a test harness for executing test suites against simple software units written in C.

The FSM Library is provided as open source as explained in Section B.2 and can be used free of charge for non-commercial applications, such as education and research. It is licensed under EUPL V.1.1.[1]

B.2 Download and Installation

The FSM Library is stored on the GitHub; its repository name is `fsmlib-cpp` (there is also a Java implementation of the algorithms contained in the FSM Library). The HTTPS identification is

```
https://github.com/agbs-uni-bremen/fsmlib-cpp.git
```

The source code can be downloaded using the command

[1] see http://ec.europa.eu/idabc/eupl.html

```
git clone https://github.com/agbs-uni-bremen/fsmlib-cpp.git
```

For producing the binary code of the library, including the executable, the cmake tool is used, and the required CMakeLists.txt have already been prepared in the repository. The scripts

```
build-for-linux.sh
build-for-osx.sh
```

provide examples how cmake can be invoked to create makefiles and auxiliary information for compiling and linking the libraries and executables involved. The `README.md` file explains in detail how the cmake configuration, compilation, and linking works under Linux, MacOSX, and Windows.

B.3 Test Generation Support

Users only interested in applying the FSM Library for automated test generation from FSM models find the test generator executable

```
fsm-test-generator
```

in the cmake compilation and linking directory (e.g. Debug.Linux) in subdirectory `generator/`. Its source code is contained in file

```
src/generator/fsm-test-generator.cpp
```

The test generator is invoked with command

```
fsm-generator [-w|-wp|-h|-hsi] [-s] [-n fsmname] \
              [-p infile outfile statefile] \
              [-a additionalstates] [-t testsuitename] \
              modelfile [model abstraction file]
```

The parameters have the following meaning.

`modelfile` Path to the file containing the model; the file extension and the contents of the file need to conform to one of the formats specified in Section B.4. This is the only mandatory parameter to be supplied.

`-w|-wp|-h|-hsi` These optional parameters select the strategy. If none of them is provided, tests are generated according to the Wp-Method (see Section 4.8.1); alternatively, parameter `-wp` can be used for this).

With parameter `-w`, the W-Method [9, 69] is applied, with `-h` the H-Method [13], with `-hsi` the HSI-Method [60].

`-s` This parameter may be selected in combination with `-w, -wp, -h`. It indicates that the test suite should not aim at I/O-equivalence but at the weaker goal *safety equivalence*, as described in [33, 29]. This often leads to fewer test cases, but only guarantees complete fault coverage for safety-related requirements.

If parameter `-s` is selected, then the `model abstraction file` explained in [29] needs to be provided as last parameter.

`-a additionalstates` All selectable test generation methods guarantee to full fault coverage, provided that

- the true behaviour of the SUT can be expressed by an observable, minimised FSM with at most $\mathtt{m}$ states,
- the reference model, when represented in observable, minimised form, has $\mathtt{n}$ states, and
- the difference $(\mathtt{m} - \mathtt{n})$ between implementation FSM states and reference model states fulfils

$$\mathtt{m} - \mathtt{n} \leq \mathtt{additionalstates}.$$

By default, additionalstates is assumed to be zero.

`model abstraction file` Only used when parameter `-s` is set, see explanation for `-s` above.

`-n fsmname` Optional name to be associated with the FSM reference model; this name appears on some graphical representations of the FSM. By default, the name "FSM" is used.

`-t testsuitename` By default, the generated test suite is written to file in the process working directory.

```
testsuite.txt
```

Using the `-t`-option, another path/file name can be selected.

`-p infile outfile statefile` With the low-level FSM format specified in Section B.4.2, optional presentation layer specifications can be provided. These three files contain the external names of FSM states, inputs, and outputs, respectively.

On successful execution, the test generator outputs the following files to the process working directory.

`<testSuiteFileName>` The generated test suite, named `testsuite.txt` by default or `<testSuiteFileName>`, if the `-t` option had been used to specify another name. The format of the test suite is as follows.

1. One test case per line.
2. Each test case consists of a sequence of (input/output) pairs, such as

   `(x0/y0).(x1/y1).(x2/y2)...`

   where `xi` is a member of the input alphabet, and `yi` a member of the output alphabet.

`<FSM-Name>.dot` This text file contains a representation of the FSM model in GraphViz format ("dot-format"), to be displayed by the GraphViz tools.[2] By default, the file name is `FSM.dot`; if another name has been selected for the FSM using the `-n` option, this name is used as `<FSM-Name>.dot`.

`<FSM-Name>.csv` If the FSM model is deterministic, a DFSM representation in CSV-format as described in Section B.4.1 is produced.

B.4 FSM Model Input Formats

B.4.1 Model Input in CSV-Format

Deterministic and completely specified (see Section 3.2) FSMs can be modelled using CSV-format, exported from tools like Excel or LibreOffice.

[2] see http://www.graphviz.org

Fig. B.1 shows a DFSM table for the Garage Door Controller example described below in Section B.6. The rules for filling out such a *DFSM transition table* are as follows.

1. The leftmost/uppermost field (A,1) is empty.
2. The first column, starting with (A,2), contains the state names, starting with the initial state in (A,2).
3. The first row, starting with (B,1), contains the identifiers if the input alphabet.
4. For state $\mathtt{s}$ and input $\mathtt{x}$, field $(\mathtt{s},\mathtt{x})$ has syntax $\mathtt{s}'/\mathtt{y}$. $\mathtt{s}'$ is the post state of the transition from state $\mathtt{s}$ on input $\mathtt{x}$, and $\mathtt{y}$ is the corresponding output. $\mathtt{s}'$ must be a valid state identifier occurring in the first column A.
5. All identifiers conform to C-variable syntax: start with a character or an underscore, only characters, underscores, or numbers may follow, no spaces.
6. If the table field for state $\mathtt{s}$ on input $\mathtt{x}$ is empty, this is interpreted as a self-loop $\mathtt{s}/\mathtt{\_nop}$: a transition with post-state identical to the pre-state, accompanied by a "no operation" output `_nop` which is always inserted into the output alphabet. As a consequence, the DFSMs specified in this way are automatically completely specified.
7. The CSV format needs semicolon ";" as separator. This is important when exporting from a formatted tabular file format (such as .xlsx or .ods) to CSV.

An example of an admissible model CSV-format looks as follows, it corresponds to the DFSM transition table shown in Fig. B.1.

```
;e1;e2;e3;e4
Door_Up;Door_closing/a1;;;
Door_Down;Door_opening/a2;;;
Door_stopped_going_down;Door_closing/a1;;;
Door_stopped_going_up;Door_opening/a2;;;
Door_closing;Door_stopped_going_down/a3;Door_Down/a3;;Door_opening/a4
Door_opening;Door_stopped_going_up/a3;;Door_Up/a3;
```

	A	B	C	D	E
1		**e1**	**e2**	**e3**	**e4**
2	**Door_Up**	Door_closing/a1			
3	**Door_Down**	Door_opening/a2			
4	**Door_stopped_going_down**	Door_closing/a1			
5	**Door_stopped_going_up**	Door_opening/a2			
6	**Door_closing**	Door_stopped_going_down/a3	Door_Down/a3		Door_opening/a4
7	**Door_opening**	Door_stopped_going_up/a3		Door_Up/a3	

Figure B.1: Tabular format for modelling DFSMs.

B.4.2 Model Input in Low-level FSM Format

The low-level format for FSM models is more flexible than the CSV-format.

- It allows for specification of nondeterministic, even non-observable FSMs.
- It is possible to specify FSMs that are not completely specified.
- It is possible to specify larger input alphabets, where not every input is processed by the FSM.
- It is possible to specify larger output alphabets, where the FSM produces only a subset of actual outputs.

Additionally, if FSMs are the result of another automated generation process, the low-level format is easier to generate automatically than the other formats accepted by the generator.

By convention, FSM definition files in low-level format carry the file extension `.fsm`. Each line of an FSM definition file specifies one transition by means of four non-negative numbers

```
<pre-state> <input> <output> <post-state>
```

The interpretation of one transition line is: "Starting in state `<pre-state>`, the FSM may transit with input `<input>` to state `<post-state>`, producing output `<output>`." The states are numbered in range $0, 1, 2, \ldots,$ (NumberOfStates $- 1$). The inputs are numbered in range $0, 1, 2, \ldots,$ (SizeOfInputAlphabet $- 1$). The outputs are numbered in range $0, 1, 2, \ldots,$ (SizeOfOutputAlphabet $- 1$).

For every state, all outgoing transitions must be listed in consecutive lines. The initial FSM state is specified by the `<pre-state>` of the first line in the file. Therefore the pre-state is not necessarily the one with number 0. This is practical when producing different FSMs from the same initial FSM by changing the initial state, but leaving all other specifications unchanged. In such a case, the block of lines starting with the new initial state is just moved to the beginning of the file.

Without further information, the generator will represent the FSM with numbers $0, 1, 2, \ldots$ for states, inputs, and outputs. The optional presentation layer files that can be passed to the test generator using the `-p` option allow to associate proper names with these items.

`<FSM-Name>.state` The state files are usually named by the FSM-Name with file extension `.state`. The file contents consists of a single text column with NumberOfStates entries, such as

```
state0
state1
state2
...
```

This associates name "`state0`" with state number 0, name "`state1`" with state number 1 and so on. In the dot-graph representation, these names will be used as state names.

`<FSM-Name>.in` The input files are usually named by the FSM-Name with file extension `.in`. The file contents consists of a single text column with SizeOfInputAlphabet entries, such as

```
in0
in1
in2
...
```

This associates name "`in0`" with input number 0, name "`in1`" with input number 1 and so on. In the dot-graph representation, these names will be used as input names.

`<FSM-Name>.out` The output files are usually named by the FSM-Name with file extension `.out`. The file contents consists of a single text column with SizeOfOutputAlphabet entries, such as

```
out0
out1
out2
...
```

This associates name "`out0`" with output number 0, name "`out1`" with output number 1 and so on. In the dot-graph representation, these names will be used as output names.

For example, the state machine depicted in transition table format in Fig. B.1 is specified in low-level format as follows.

fsm-file. The state machine with its transitions is specified by

```
0 0 1 4
0 1 0 0
0 2 0 0
0 3 0 0
1 0 2 5
1 1 0 1
1 2 0 1
1 3 0 1
2 0 1 4
2 1 0 2
2 2 0 2
2 3 0 2
3 0 2 5
3 1 0 3
3 2 0 3
3 3 0 3
4 0 3 2
4 1 3 1
4 2 0 4
4 3 4 5
```

```
5 0 3 3
5 1 0 5
5 2 3 0
5 3 0 5
```

state-file. The state file is specified by

```
Door_Up
Door_Down
Door_stopped_going_down
Door_stopped_going_up
Door_closing
Door_opening
```

in-file. The input file is

```
e1
e2
e3
e4
```

out-file. The output file is

```
_nop
a1
a2
a3
a4
```

B.4.3 RTT-MBT-FSM: Model Input in Graphical Format

As an alternative to the FSM input formats described above, a simple graphical tool can be used to specify FSMs as graphs, where the graph nodes represent FSM states and directed, labelled graph edges represent FSM transitions labelled by inputs and outputs. The tool is called RTT-MBT-FSM and has been provided by Verified Systems International GmbH[3]; it does not require any license fees and can be used free of charge. The tool is programmed in Java, and it is provided as jar-files in sub-directory

```
fsm-gui/
```

The fsm-gui requires some platform-specific graphical libraries, therefore different jar-files are provided in this directory for running the tool under MacOSX, Linux, or Windows, respectively.

[3]https://www.verified.de

```
fsm-gui-1.0-0.cocoa.macosx.x86_64.jar
fsm-gui-1.0-0.gtk.linux.x86_64.jar
fsm-gui-1.0-0.win32.x86_64.jar
```

A java runtime with version 1.8.0 or newer is required. The fsm-gui is activated in directory `fsm-gui` with commands

```
java -XstartOnFirstThread -jar fsm-gui-1.0-0.cocoa.macosx.x86_64.jar
```

for MacOSX,

```
java -jar fsm-gui-1.0-0.gtk.linux.x86_64.jar
```

for Linux, and

```
java -jar fsm-gui-1.0-0.win32.x86_64.jar
```

for Windows platforms. RTT-MBT-FSM provides four pull-down menus:

SWT is for quitting the tool and setting preferences in future versions,

File allows to open existing FSM model files (extension *.fsm), creating new FSM model files, and saving changed model files,

Manage is for requirements management – this is not supported by the software of the FSM Library, but only in the commercial version of the model-based test generation tool RTT-MBT[4],

Help shows the version number and will offer help information in future versions.

When invoking the tool, an empty canvas is shown, and a new FSM can be constructed using the four interaction commands on the tool bar above the canvas, as shown in Fig B.2.

1. Activating the state symbol (circle) allows for creating one or more FSM states in a row. A single-click on the canvas will place a new state in the indicated position, and a pop-up menu allows to specify the state name (C-naming conventions, no blanks allowed) and mark the initial state. New states can be created while the state symbol is active.

[4]https://www.verified.de/products/model-based-testing/

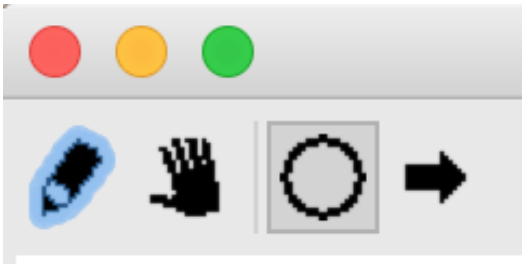


Figure B.2: RTT-MBT-FSM Tool bar for operations on the canvas.

2. Activating the transition symbol (right arrow) allows for creating transitions by single-clicking the source state and then the target state. This also works for self-loops: just click the state twice. A pop-up menu allows to insert input and output associated with this transition.

3. Activating the hand symbol allows to pick a state with the left mouse button pressed and move the state to another position – the connecting transitions are re-arranged correspondingly.

4. Activating the edit symbol (pen) allows to edit states by double-clicking them. The edit window on the right-hand side of the canvas allows to change the state name, as well as the "initial state" marker. By clicking the 'Transition' button in the edit menu, the state's outgoing transitions are listed. By right-clicking such a transition, a context menu pops up and allows to edit or delete the transition.

When creating such an FSM model, it is assumed that the input alphabet Σ_I consists of the union of all inputs specified in any transition on the canvas. Likewise, the output alphabet Σ_O is assumed to be the union of the outputs specified in the transitions. Similar to the FSM model definition by CSV-files, the FSM is automatically completely specified. Consider, for example, the DFSM specified in Fig B.5. In the initial state `Door_Up`, only one outgoing transition has been specified which is triggered by input `e1`. The FSM Library internally adds self-loops for each of the other input events `e2`, `e3`, `e4`. To indicate that "nothing happens" when these events occur in this state, an additional output `_nop` is created which denotes "no operation".

B.5 Test Execution Support

The FSM Library provides a simple *test harness*[5] for executing software unit tests or module tests of libraries. This can be applied to C-Software with interfaces whose input and output values can be enumerated with reasonable effort, so that inputs can be abstracted to events of an input alphabet, and outputs to events of an output alphabet. The source code of the test harness can be found in `harness/fsm-test-harness.c`.

Since a general-purpose test harness cannot anticipate all possible interfaces of the software under test, an *SUT wrapper* needs to be provided, which acts as an adaptor between the test harness and the SUT (see Fig. B.3). The SUT wrapper has to provide three standardised interface functions to the test harness. Their C prototypes, as referenced by the harness, are declared as

```
extern void sut_init();
extern void sut_reset();
extern const char *sut(const char *x);
```

Function `sut()` is used by the test harness to execute the test cases against the SUT. On each test step, the harness calls `sut()` with the external C-string representation of the next input event to be processed by the SUT. The SUT wrapper function has to transform this input string into the concrete input values associated with this event. Note that the concrete input values can also comprise global variables read by the SUT. The `sut()` function then invokes the SUT by its entry function (`sw_under_test()` in Fig. B.3). When the SUT returns, the `sut()` function transforms the return value, output parameters, and global variables written to by the SUT back to an output event in string representation. This string is returned to the harness.

For being able to transform strings of the FSM input alphabet into SUT input data and SUT return data to FSM output alphabet strings, the SUT wrapper has to set up data structures carrying mapping information. This setup has to be performed by function `SUT_init()` which is called once by the test harness before the proper test execution begins. If the software under test needs initialisation actions to be performed before it can be used, these actions also have to be performed in this function.

[5] A software program driving a test against a software system under test.

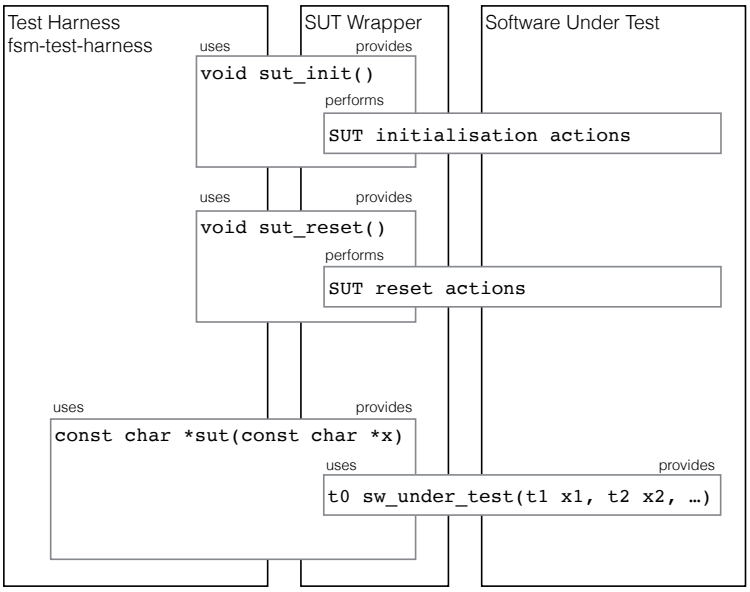


Figure B.3: Test harness interacting with software under test by means of an SUT wrapper.

Between consecutive test case executions, the SUT has to be reset to its initial state, since every test case produced by the FSM test generator starts expects the SUT to be initialised before the first test step is executed. For this purpose, the SUT wrapper needs to provide the `sut_reset()` function, which has to exercise the appropriate reset actions on the SUT.

As soon as the SUT wrapper and the SUT software are available, an executable test harness can be produced by linking the object files of test harness, the SUT wrapper, and the SUT itself into one executable. The test harness is provided by the FSM Library as a sub-library

```
libfsm-harness.a
```

Therefore the executable – to be named `testproc` by default – can be created by using the command

```
cc -o testproc  <path to the libfsm-harness.a> \
                <SUT Wrapper object file> <SUT object files>
```

where 'cc' denotes the local C-compiler. The test procedure is activated by starting `testproc` with parameter `testsuite.txt`, where `testsuite.txt` is the name of the test suite to be executed. The test suite file must be generated by the FSM test generator described in Section B.3 or have the same format as described there.

An example how to create an executable test procedure from test harness, SUT wrapper, and SUT library is presented in Section B.6.

B.6 Example: Garage Door Controller

B.6.1 Problem Description

In this section, the application of the FSM test generator and the test execution by means of the test harness is illustrated, using an example originally introduced by Paul C. Jorgensen in [39].

The *garage door controller (GDC)* is a computer managing the up and down movement of a garage door via an electric motor, as shown in the overview diagram in Fig. B.4. The GDC outputs commands `a1, a2, a3, a4` to the motor, initiating down movement, up movement, stopping the motor, and reversing its down movement into up movement. As inputs, the GDC receives a command "button pressed" (`e1`) from a remote control device,

an event “door reaches position down” (e2), and an event “door reaches position up” (e3). Additionally, a safety device is integrated by means of a light sensor which sends an event “light beam crossed” (e4) when something moves underneath the garage door while the door is closing.

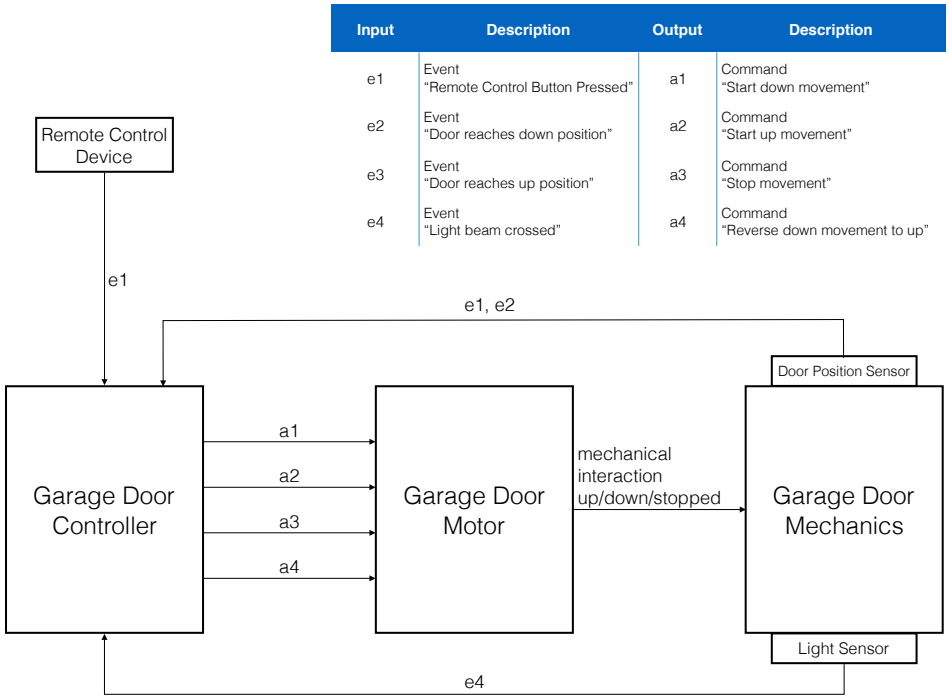


Figure B.4: Garage door controller and its operational environment.

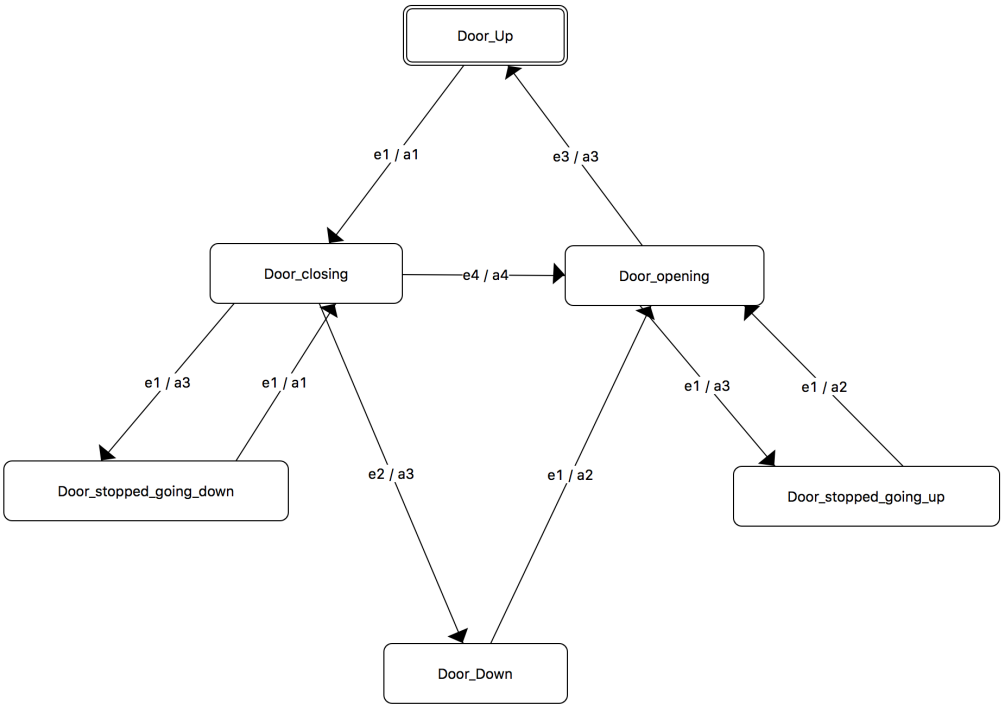


Figure B.5: Behaviour of the garage door controller, modelled by a DFSM.

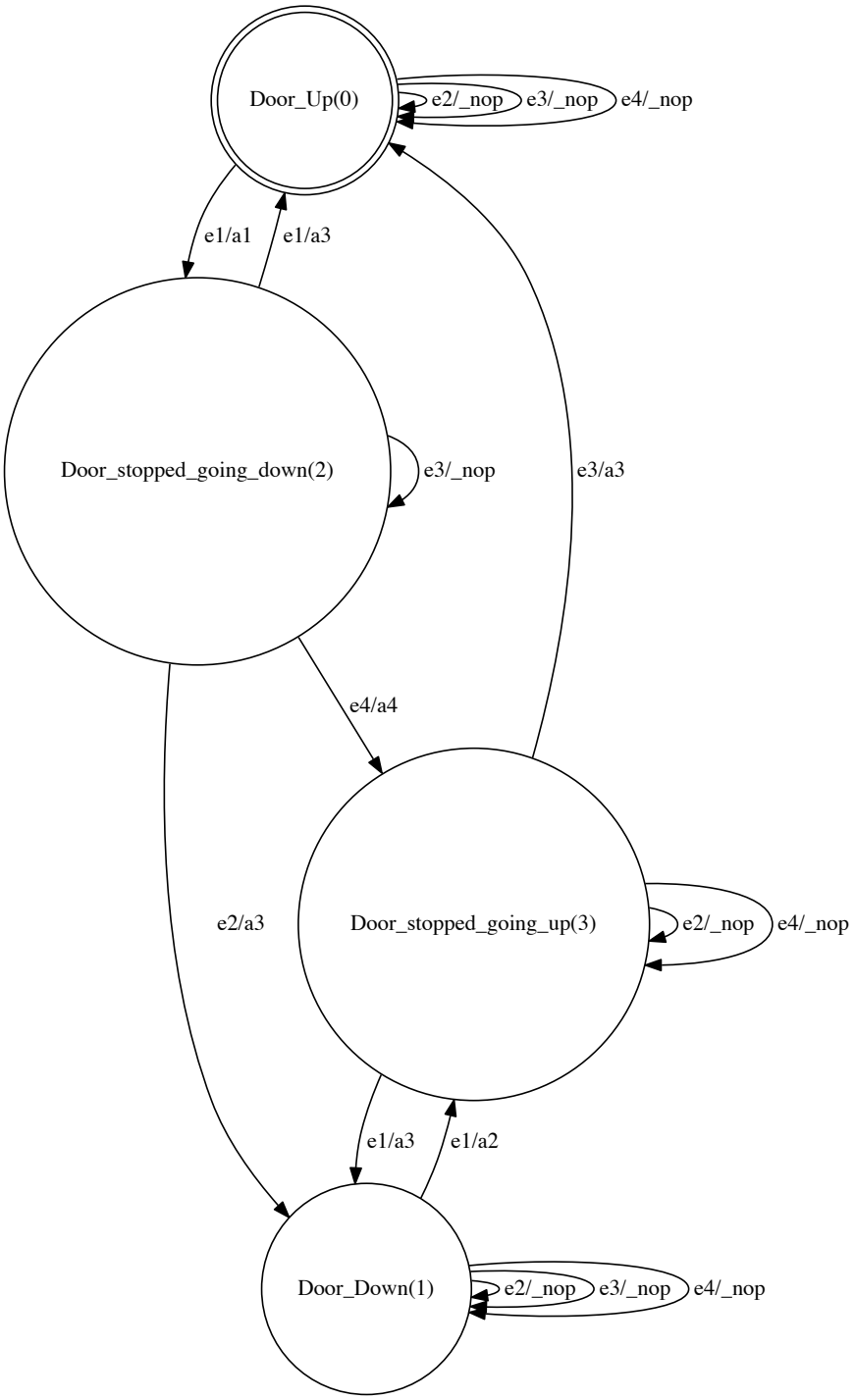


Figure B.6: Minimised DFSM, equivalent to the GDC model from Fig. B.5.

The expected behaviour of the GDC is modelled by the FSM in Fig. B.5. The FSM states, as shown in this figure, have the meaning

State	Description
s1	Initial state Door Up
s2	Door down
s3	Door stopped going down
s4	Door stopped going up
s5	Door closing
s6	Door opening

In the initial state s1, the door is expected to be in the UP position, and the "button pressed" event from the remote control triggers a "Start down movement" command to the motor. The GDC transits to state s5. In this "Door closing" state, an input e4 from the light sensor leads to an a4 command to the motor, with the effect that the down movement of the door is reversed to up movement. This leads to state s6. During down movement in state s5, another occurrence of the e1-event leads to a "Stop movement" command to the motor. Then the downward movement is resumed (output a1), as soon as another e1-command is given. When the door sensor signals that the door has reached the sown position (e2), the motor is stopped with command a3. From the "Door down" state s2, another e1-event triggers the analogous actions for moving the door up, until the UP position is reached. During the UP-movement, inputs from the light sensor do not have any effect.

In Fig. B.1 above, the same DFSM is modelled by means of a transition table. Observe that these two DFSM models are interpreted as completely specified state machines (see Section 3.2), so every input event e1, e2, e3, e4 can occur in every state. The unspecified transitions – for example, occurrence of event e2 in state s1 – are interpreted as having no effect. To this end, an auxiliary DFSM output "no operation" (_nop) is internally introduced when instantiating the DFSM from either of these models. All unspecified transitions are internally defined as self-loops with output _nop.

Note that the DFSM in Fig. B.5 is not minimal; it has been represented in this form to optimise the readability. The equivalent minimised machine is shown in Fig. B.6. This has been calculated using the method minimise() in the FSM Library class Dfsm. The output graph shown in Fig. B.6 has

been created by using the method `toDot()` in the same class, together with the GraphViz[6] tool.

B.6.2 GDC System Under Test

A sample implementation in C is given in the FSM Library, directory `src/example`, in file `gdclib.c`; the public function interfaces are specified in `gdclib.h` as follows.

```
1  typedef enum {
2      e1,
3      e2,
4      e3,
5      e4
6  } gdc_inputs_t;
7
8  typedef enum {
9      nop,
10     a1,
11     a2,
12     a3,
13     a4
14 } gdc_outputs;
15
16 extern void gdc_reset();
17 extern gdc_outputs gdc(gdc_inputs_t x);
```

The GDC processes expects its inputs in enumeration format `gdc_inputs` and returns actions to the motor in format `gdc_outputs`. The implementation in `gdclib.c` follows the state machine programming paradigm and is straightforward, so that no further comments are needed.

B.6.3 Test Generation

To generate test cases using the W-Method or the Wp-Method, we call the the FSM test generator with a model file for the GDC in any of the admissible formats. Prepared files are available in the FSM Library sub-directory `resources`:

`garage-door-controller.csv` contains the GDC model in CSV format, as specified in Section B.4.1. For experimenting with model changes, the

[6] http://www.graphviz.org

file `garage-door-controller.xlsx` can be used, and its content can be exported to CSV format.

`garage.fsm` with presentation layer files `garage.in`, `garage.out`, `garage.state` contains the GDC model in low-level format as described in Section B.4.2.

`garage-door-controller.fsm` contains the model file in JSON format, as produced by the graphical FSM modelling front-end described in Section B.4.3.

Invoking the FSM test generator with command

```
fsm-test-generator garage-door-controller.csv
```

for example, creates a test suite `testsuite.txt` generated using the Wp-Method, with the following content.

```
(e1/a1).(e2/a3).(e1/a2).(e2/_nop)
(e1/a1).(e2/a3).(e1/a2).(e1/a3)
(e1/a1).(e2/a3).(e2/_nop).(e1/a2)
(e1/a1).(e2/a3).(e3/_nop).(e1/a2)
(e1/a1).(e2/a3).(e4/_nop).(e1/a2)
(e1/a1).(e4/a4).(e1/a3).(e1/a2)
(e1/a1).(e4/a4).(e2/_nop).(e2/_nop)
(e1/a1).(e4/a4).(e2/_nop).(e1/a3)
(e1/a1).(e4/a4).(e3/a3).(e1/a1)
(e1/a1).(e4/a4).(e4/_nop).(e2/_nop)
(e1/a1).(e4/a4).(e4/_nop).(e1/a3)
(e1/a1).(e1/a3).(e1/a1)
(e1/a1).(e3/_nop).(e2/a3)
(e2/_nop).(e1/a1)
(e3/_nop).(e1/a1)
(e4/_nop).(e1/a1)
```

The suite contains 16 test cases and has been generated under the assumption that the minimised DFSM representing the unknown implementation behaviour does not have more states than the minimised reference model. Each test case consists of a sequence of pairs (`x/y`) with SUT input `x` and expected output `y`.

B.6.4 Creating the SUT Wrapper

As explained in Section B.5, an SUT wrapper has to be provided before the test suite can be executed using the test harness that comes with the FSM Library. For the GDC example, the wrapper implementation is provided in sub-directory `src/example`, file `sut_wrapper.c`, and it looks as follows.

```
1  #include <string.h>
2  #include "gdclib.h"
3
4  /**
5   *   Helper data structures and functions for
6   *   SUT test wrapper
7   */
8  static const char* outputs[5];
9  static const char* inputs[4];
10
11 static gdc_inputs_t inStr2Enum(const char* input) {
12     int i;
13     for ( i = 0; i < 4; i++ ) {
14         if ( strcmp(inputs[i],input) == 0 ) {
15             return (gdc_inputs_t)i;
16         }
17     }
18     return e1;
19 }
20
21 void sut_init() {
22     outputs[nop] = strdup("_nop");
23     outputs[1] = strdup("a1");
24     outputs[2] = strdup("a2");
25     outputs[3] = strdup("a3");
26     outputs[4] = strdup("a4");
27
28     inputs[0] = strdup("e1");
29     inputs[1] = strdup("e2");
30     inputs[2] = strdup("e3");
31     inputs[3] = strdup("e4");
32 }
33
34 void sut_reset() {
35     gdc_reset();
36 }
37
38
```

```
39 const char* sut(const char* input) {
40     return outputs[gdc(inStr2Enum(input))];
41 }
```

During a test execution, the test harness invokes SUT function `sut()` with an input C-string `"e1"`, `"e2"`, `"e3"`, `"e4"`. The SUT wrapper transforms this into the corresponding enum value expected by the SUT, using auxiliary function `inStr2Enum()`. The SUT function `gdc()` is called with the enum value, and its enum return is transformed into a string as expected by the test harness, using the `output[]` array. To set up the necessary transformation data structures, function `sut_init()` is called once at start-of-test. Between test cases, the SUT is reset using wrapper function `sut_reset()`.

B.6.5 Test Execution

With SUT and SUT wrapper at hand, the executable test program can be built. To this end, use command

```
cc -o testproc <path to test harness library> \
               <path to SUT wrapper and SUT library>
```

This creates an executable `testproc` in the working directory. Using command

```
./testproc <path to test suite file>
```

runs the test cases against the SUT, and the following output is produced.

```
TC-1: (e1,a1).(e2,a3).(e1,a2).(e2,_nop) PASS
TC-2: (e1,a1).(e2,a3).(e1,a2).(e1,a3) PASS
TC-3: (e1,a1).(e2,a3).(e2,_nop).(e1,a2) PASS
TC-4: (e1,a1).(e2,a3).(e3,_nop).(e1,a2) PASS
TC-5: (e1,a1).(e2,a3).(e4,_nop).(e1,a2) PASS
TC-6: (e1,a1).(e4,a4).(e1,a3).(e1,a2) PASS
TC-7: (e1,a1).(e4,a4).(e2,_nop).(e2,_nop) PASS
TC-8: (e1,a1).(e4,a4).(e2,_nop).(e1,a3) PASS
TC-9: (e1,a1).(e4,a4).(e3,a3).(e1,a1) PASS
TC-10: (e1,a1).(e4,a4).(e4,_nop).(e2,_nop) PASS
TC-11: (e1,a1).(e4,a4).(e4,_nop).(e1,a3) PASS
TC-12: (e1,a1).(e1,a3).(e1,a1) PASS
TC-13: (e1,a1).(e3,_nop).(e2,a3) PASS
TC-14: (e2,_nop).(e1,a1) PASS
TC-15: (e3,_nop).(e1,a1) PASS
TC-16: (e4,_nop).(e1,a1) PASS
```

The reader is invited to experiment with different SUT implementations, fault injections into the `gdclib.c` implementation, and different model variants. He or she should keep in mind that some fault injections may increase the number of states in the minimised DFSM corresponding to the true SUT behaviour. If this is suspected, the parameter `-a <additional states>` has to be used for test generation with a suitable estimate (see Section B.3). Otherwise it is not guaranteed that the test suite will uncover every violation of language equivalence between implementation and reference model.

B.7 FSM Library Classes and Methods Overview

In the previous sections it has been shown how the FSM test generator and test harness can be used to create model-based FSM test suites and execute them against an SUT implemented as a C-library with simple interfaces.

Alternatively, users can create their own FSM-related applications in C++, using the static libraries that are part of the FSM Library. When investigating the build directory – for example, `Debug.OSX` or `Release.Linux` – created for constructing the FSM Library with cmake, the following static libraries can be found after a successful build.

```
./example/libfsm-example.a
./externals/jsoncpp-0.10.0/src/libjsoncpp.a
./fsm/libfsm-fsm.a
./harness/libfsm-harness.a
./interface/libfsm-interface.a
./sets/libfsm-sets.a
./trees/libfsm-trees.a
```

The main library is `libfsm-fsm.a` with its main class `Fsm` containing methods for reading, transforming and creating test suites from arbitrary finite state machines. The library also provides a derived class `Dfsm` offering special methods operating on FSMs that are already known to be deterministic. FSMs stored in a file conforming to one of the formats described above are read by means of the various constructors specified in files

```
src/fsm/Fsm.h
src/fsm/Dfsm.h
```

To get an overview over the classes and methods provided by the FSM Library, users may unpack the zip-archive

```
doc/fsmlib-cpp-doc.zip
```

in the `doc/` sub-directory and open the HTML file

```
doc/html/index.html
```

with a browser. There, an overview over the classes, operations, and associated header files is given. Readers will find that most of the algorithms explained in Part II of these lecture notes are implemented in these classes. This documentation has been created using the `Doxygen` tool.[7] For re-generating this documentation from scratch or adapting the documentation style to your preferences, a Doxygen documentation file is provided in

```
doc/Doxyfile
```

Note that also a visitor framework has been set up for traversing FSMs and their states (class `FsmNode`) and transitions (class `FsmTransition`). This framework defines `visit()` methods for these classes in header file `FsmVisitor.h`, and the FSM main classes all provide `accept()` methods as entry points for these visitors. An example showing how the visitor framework can be used is given by the `FsmPrintVisitor` which traverses FSMs using their `accept()` methods and writing the FSM-related data to standard output.

As an example about how the FSM Library can be used from other programs, a C++ main program is provided in

```
src/main/main.cpp
```

This main program instantiates various FSMs from files contained in subdirectory `resources` and performs tests checking the functionality of FSM Library methods.

[7] http://www.stack.nl/~dimitri/doxygen/