

# libfsmtest

## A Library for Model-based Testing With Finite State Machines User Manual

Jan Peleska, Moritz Bergenthal, Niklas Krafczyk, and  
Robert Sachtleben  
peleska@uni-bremen.de

Issue 1.8  
2021-08-27

All rights reserved © 2021 Jan Peleska, Moritz Bergenthal, Niklas Krafczyk, and Robert Sachtleben
---

# Change History

Issue	Date	Change description
1.8	2021-08-27	Add table specifying application conditions for each test generation method implemented in the library (Section ‘Available Test Generation Methods’)
1.7	2021-08-24	Described extended functionality of CSV reader and CSV writer (Section 3.1.2, Chapter 6) : these operate now on arbitrary FSMs (before, it was only DFSMs).  Extend documentation of test generation with abstraction (Section 4.3)
1.6	2021-07-28	Extend documentation of the checker (Section 5.3)
1.5	2021-07-27	Add Example 1 about generator usage (Section 5.2). Section 5.4 (Using the Test Harness) added. Chapter 6 (saving FSMs to Disk) added.
1.4	2021-07-21	Small documentation updates. Section 3.3 on random creation and mutation of FSMs added. Section 5.3 (Using the Checker) added.
1.3	2021-07-18	Start description of test generation with abstraction FSMs in Section 4.3. Chapter 5 (Using Generator ...): description of the generator added.
1.2	2021-07-17	Update documentation of from-file creators. Completed documentation of test generation visitors.
1.1	2021-07-07	Extended documentation.
1.0	2021-06-15	Initial issue.

# Preface

This manual contains the documentation of the `libfsmtest`. This library has been programmed in C<sup>++</sup>. Its classes contain algorithms for instantiating, evaluating, transforming, and creating test suites from **finite state machines (FSM)** representing **Mealy Automata**. The underlying theory has been summarised in the lecture notes [14]. We will give additional references where needed.

The `libfsmtest` class library is a re-factored, novel implementation of its predecessor `fsmlib-cpp`<sup>1</sup>, an open source library whose maintenance is now discontinued. The new `libfsmtest` is open source as well<sup>2</sup>, but it differs from the `fsmlib-cpp` with respect to the following aspects.

- `libfsmtest` is licensed under the MIT license <https://opensource.org/licenses/MIT>.
- The main classes of the `libfsmtest` class library have been reduced to methods for evaluating FSMs, simulating them step by step or with pre-defined input traces, and for checking whether given input/output traces are contained in the FSM's language.
- The creation of FSMs from different file formats, as well as their transformation and random creation, has been moved to factory methods.
- The test generation algorithms are now implemented in separate **visitors** [4], facilitating the addition of new algorithms without having to change the main classes.

---

<sup>1</sup><https://github.com/agbs-uni-bremen/fsmlib-cpp.git>

<sup>2</sup><https://bitbucket.org/JanPeleska/libfsmtest>

The main objective of this re-design was to simplify the library API for users and to facilitate library extension and maintenance. In comparison to other existing libraries supporting FSM-based testing, the `libfsmtest` has the following unique selling points.

- It supports both complete and partial FSMs, and both deterministic and non-deterministic FSMs.
- It provides test generation algorithms for different variants of conformance relations, such as language equivalence, safety-equivalence [9] and several variants of reduction.
- It provides ready-to-use main programs for **test generation** with all the available methods and for running test suites generated from a ‘reference FSM’ against ‘implementation FSMs’. The latter program corresponds to an **FSM model checker**.
- It provides a **test harness** framework which supports the execution of test suites generated from FSMs against software under test programmed as `C++` class libraries.

# Contents

<b>Preface</b>	<b>ii</b>
<b>List of Figures</b>	<b>vi</b>
<b>1 Download and Installation</b>	<b>1</b>
<b>2 Library Overview</b>	<b>2</b>
2.1 Top-level Directory . . . . .	2
2.2 Source Directory src/ . . . . .	2
2.3 Source Directory src/libfsmtest . . . . .	3
<b>3 FSM Creation</b>	<b>5</b>
3.1 FSM Creation from Files . . . . .	5
3.1.1 Raw File Format . . . . .	5
3.1.2 CSV File Format for FSMs . . . . .	11
3.1.3 Reading FSMs from Files With Automated Format Identification . . . . .	14
3.2 FSM Creation by Transformation . . . . .	16
3.2.1 Auto-Completion Transformers . . . . .	16
3.2.2 Transformation to Prime Machines . . . . .	19
3.3 Random Creation and Mutation of FSMs . . . . .	24
3.3.1 Creation of Random Completely Specified FSMs. . .	24
3.3.2 Random Removal of Transitions . . . . .	26
<b>4 Test Suite Generation</b>	<b>28</b>

4.1	Test Generation Frame, Test Generation Method, and Test Suite . . . . .	28
4.2	Available Test Generation Methods . . . . .	30
4.3	Test Generation Methods Using Abstraction . . . . .	33
<b>5</b>	<b>Using Generator, Checker, and Test Harness – the Workflow</b>	<b>40</b>
5.1	A Sample Test Campaign . . . . .	40
5.1.1	Reference Model Description . . . . .	40
5.1.2	GDC System Under Test . . . . .	44
5.2	Using the Test Generator . . . . .	46
5.3	Using the Checker . . . . .	50
5.4	Using the Test Harness . . . . .	54
<b>6</b>	<b>Visitors for Saving FSMs to Disk</b>	<b>62</b>
<b>7</b>	<b>Bibliography</b>	<b>64</b>

# List of Figures

3.1	GraphViz representation of DFSM specified in raw format with the files shown below. . . . .	8
3.2	Tabular format for modelling DFSMs. . . . .	12
4.1	FSM $A$ with different regions: once state $s_2$ has been reached, the FSM will only visit states in $\{s_2, s_3, s_4\}$ ; it will never return to $s_0$ or $s_1$ . . . . .	35
4.2	FSM abstraction $\alpha(A)$ of the original FSM $A$ shown in Fig. 4.1.	36
4.3	Minimised FSM associated with $\alpha(A)$ from Fig. 4.2. . . . .	36
5.1	Garage door controller and its operational environment. . .	41
5.2	Behaviour of the garage door controller, modelled by a DFSM.	42
5.3	Tabular format for modelling DFSMs. . . . .	43
5.4	Minimised, auto-completed DFSM, equivalent to the GDC model from Fig. 5.2. . . . .	44
5.5	Test harness, SUT wrapper, and software under test. . . . .	55

# Listings

3.1	Create a FSM from raw format files. . . . .	10
3.2	Create a FSM from a CSV file. . . . .	13
3.3	Create an FSM from file with automated format identification. . . . .	14
3.4	Use of the auto-completion transformers. . . . .	17
3.5	Use of the auto-completion transformers. . . . .	18
3.6	Transformation to prime machine. . . . .	20
3.7	Transformation to an initially connected FSM. . . . .	21
3.8	Transformation to an observable FSM. . . . .	22
3.9	Transformation to minimised FSM. . . . .	23
3.10	Create a Random Completely Specified FSM. . . . .	25
3.11	Create a Random Completely Specified FSM. . . . .	26
3.12	Transformation to remove random transitions of an FSM. . . . .	27
4.1	Application of test generation frame, method, and suite. . . . .	29
5.1	SUT wrapper source frame. . . . .	57
5.2	SUT wrapper for the garage door controller example. . . . .	59
6.1	Write an FSM to disk in three different formats format. . . . .	63



# Chapter 1

## Download and Installation

The libfsmtest class library can be downloaded from [bitbucket.org](https://bitbucket.org/JanPeleska/libfsmtest), using command

```
1 git clone https://bitbucket.org/JanPeleska/libfsmtest.git
```

For producing the binary code of the library, including the executables, the cmake tool<sup>1</sup> is required and needs to be installed first, if it is not available on your platform. The required CMakeLists.txt have already been prepared in the libfsmtest repository.

The README.md file explains in detail how the cmake configuration, compilation, and linking work under Linux, MacOSX, and Windows.

---

<sup>1</sup><https://cmake.org>

# Chapter 2

## Library Overview

### 2.1 Top-level Directory

The top-level directory `libfsmtest/` of the `libfsmtest` class library is structured as follows.

1. Directory `doc/` contains this manual, named `libfsmtest.pdf`.
2. Directory `resources/` contains sample FSMs, encoded in different file formats.
3. Directory `src/` contains the source code.

### 2.2 Source Directory `src/`

The source directory `src/` contains the complete `libfsmtest` source code, classes and main programs. It is structured as follows.

1. Directory `libfsmtest/` contains all classes, both `cpp`-files and header files.
2. Directory `usage-demo/` contains main program file `usage-demo.cpp`. This file contains many small procedures, each showing how to use a specific feature of the `libfsmtest` classes. If the examples shown in this documentation do not suffice, or if you wish to create your own

code by copying from existing examples, this is the place to go. The file names used in this documentation refer to files that actually exist in the `libfsmtest/resources` directory. In program usage-demo, these files are referenced with their absolute path

`RESOURCEPATH + <filename>`

The `#define`-value of `RESOURCEPATH` is determined while building `libfsmtest` using `cmake`.

3. Directory `generator/` contains the generator main program `generator.cpp`. This program uses the `libfsmtest` class library to generate test suites. The reference FSMs to be used and the test generation methods to be applied are provided as command line arguments.
4. Directory `checker/` contains the checker main program `checker.cpp`. It takes an implementation FSM file name and a test suite file name as command line arguments and runs the test suite against this FSM.
5. Directory `harness/` contains the main program of the test harness, called `harness.cpp`. The harness needs a wrapper to refine input data to the system under test (SUT) and abstract SUT outputs back to events of the FSM output alphabet. Such a wrapper can be implemented by inserting code into the source frame `sut_wrapper.cpp`, also contained in this directory. In sub-directory `example/` an example is shown, explaining how to configure the wrapper and run the harness with a test suite against a C++ application library.

## 2.3 Source Directory `src/libfsmtest`

The directory containing the `libfsmtest` class library has the following sub-structure.

1. Directory `creators/` contains factory methods for creating FSMs from files in various formats. Moreover, creator classes in `subdirec-`

tory creators/randoms contains generators of random FSMs. Finally, subdirectory creators/transformers contains FSM transformations.<sup>1</sup>

2. Directory fsm/ contains the main classes for FSMs and their basic evaluation and simulation methods. The root class is Fsm, sub-classed by Ofsm for observable (nondeterministic) FSMs, sub-classed by Dfsm (deterministic FSMs).
3. Directory visitors/ contains visitors implementing the different test generation methods provided by the libfsmtest. The virtual visit-methods are pre-defined in header file FsmVisitor.hpp contained in this directory. The concrete method visitors for test generation algorithms, however, will usually subclass from TestGenerationVisitor.hpp (also contained in this directory), because this class extends the FsmVisitor by operations used by most concrete test case generation visitors. The latter are named after the method they implement, such as HMethod.hpp, HMethod.cpp. To study an example before adding you own test generation visitors, see the W-Method visitor WVisitor.hpp, .cpp.

A second type of visitors is intended for writing FSM instances to files in different format. These visitors will subclass from ToFileVisitor.hpp.

4. Directory testsuite/ contains the main class TestSuite.hpp, .cpp for creating test suites, together with auxiliary classes for representing traces in linear or tree structure.

---

<sup>1</sup>For example, a transformation of an arbitrary nondeterministic FSM into an observable FSM.

# Chapter 3

## FSM Creation

Though the main classes `Fsm`, `Ofsm`, and `Dfsm` have their own constructors, these are used only indirectly via factory methods. The `libfsmtest` provides factory methods for creating FSMs by reading them from various file formats, by transformation of existing FSMs, and by random generation. The available creator methods are described in the subsequent sections. Every creation method inherits from the abstract class `FsmCreator`, see header file

```
src/libfsmtest/creators/FsmCreator.hpp
```

If you wish to program your own FSM creation method, you should also create a concrete class inheriting from `FsmCreator`.

### 3.1 FSM Creation from Files

#### 3.1.1 Raw File Format

The most versatile format for reading FSMs from file is the so-called **raw format**.

- It allows for specification of deterministic or nondeterministic FSMs. In the nondeterministic case, the FSM may be observable or unob-

servable. Both deterministic and nondeterministic FSMs can be completely specified or partial.<sup>1</sup>

- 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 the output alphabet.

Additionally, if FSMs are the result of another automated generation process, the raw format is easier to generate automatically than the other formats accepted by the generator.

By convention, FSM definition files in raw 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  $0, 1, 2, \dots, (\text{NumberOfStates} - 1)$ . The inputs are numbered in range  $0, 1, 2, \dots, (\text{SizeOfInputAlphabet} - 1)$ . The outputs are numbered in range  $0, 1, 2, \dots, (\text{SizeOfOutputAlphabet} - 1)$ .

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.

Optionally, the `.fsm`-file can be complemented by three files defining a **presentation layer**; this layer specifies the external names of states, input events, and output events. The first file associates external state names with internal state numbers. The name of the `.fsm`-file without that extension is usually treated as the FSM-Name.

---

<sup>1</sup>Please look up the definitions in [14], if you are not familiar with these terms.

<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 external FSM representations, 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 external FSM representations, 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 external FSM representations, these names will be used as output names.

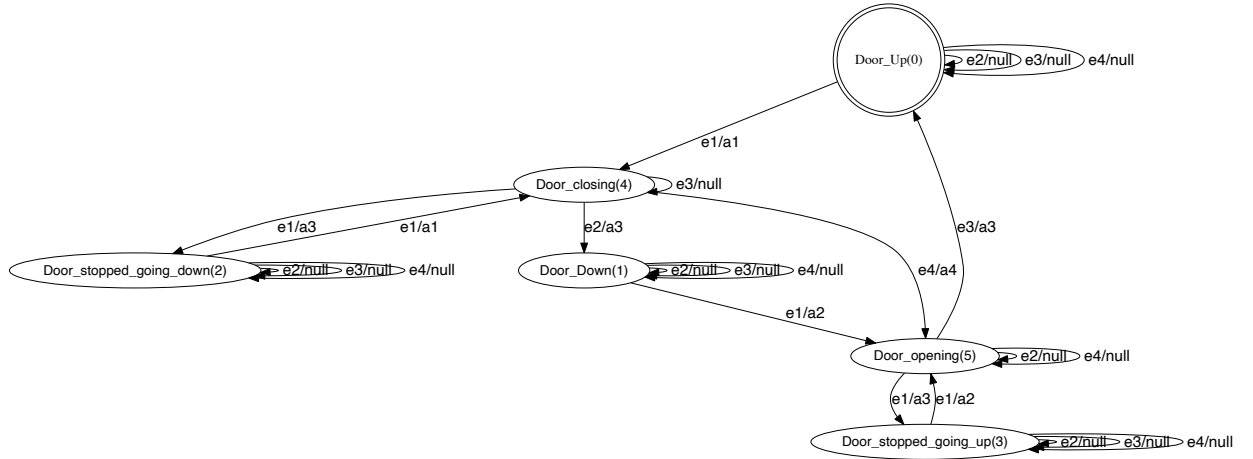


Figure 3.1: GraphViz representation of DFSM specified in raw format with the files shown below.

For example, the state machine depicted in Fig. 3.1 is specified in raw 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

```
null
a1
a2
a3
a4
```

The FSM creator using raw format is implemented as class `FsmFromFileCreator` with header file

```
src/creators/FsmFromRawCreator.hpp
```

Listing 3.1 shows how to create an FSM from a raw input file, together with the three files for the presentation layer.

---

Listing 3.1: Create a FSM from raw format files.

```
1 #include "Fsm.hpp"
2 #include "FsmFromRawCreator.hpp"
3
4 using namespace std;
5 using namespace libfsmtest;
6
7 int main(int argc, char* argv[]) {
8
9     FsmFromRawCreator creator("nonObservable.fsm",
10                             "nonObservable.state",
11                             "nonObservable.in",
12                             "nonObservable.out",
13                             "MY_NONOBSERVABLE_FSM");
14
15     unique_ptr<Fsm> fsm = creator.createFsm();
16
17     // . . .
18
19 }
```

---

All classes implementing the factory methods from header file `FsmCreator.hpp` need to provide one creator method `std::unique_ptr<Fsm> createFsm()`.

Note that the factory methods decide internally, whether to create instances of class `Fsm` or instances of the sub-classes `Ofsm` or `Dfsm`. The creation result is always a unique pointer to `Fsm`. Virtual functions and polymorphism ensure that the proper methods are called. For example, the call `bool b = fsm->isMinimal();` will use Gill's Pk-table algorithm [5] when the FSM is deterministic, a nondeterministic variant thereof [14] if the machine is observable, and it will throw an exception if the FSM is non-observable.

### 3.1.2 CSV File Format for FSMs

FSMs can also be modelled using CSV-format, exported from tools like Excel or LibreOffice. Fig. 3.2 shows a DFSM table for the Garage Door Controller example described below in Chapter 5. The rules for filling out such a *FSM 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 of the input alphabet.
4. (a) For a state  $s$  and input  $x$ , field  $(s, x)$  has syntax  $s'/y$ .  $s'$  is the post state of the transition from state  $s$  on input  $x$ , and  $y$  is the corresponding output.  $s'$  must be a valid state identifier occurring in the first column A.  
(b) There can be multiple outgoing transitions for state  $s$  and input  $x$ . Then, field  $(s, x)$  contains a comma-separated list  $s'_1/y_1, \dots, s'_n/y_n$  of the above syntax to specify more than one transition. For  $s'_i$  and  $y_i$ , the same conditions as above apply.
5. All identifiers for states, inputs, and outputs conform to C-variable syntax: start with a character or an underscore, only characters, underscores, or numbers may follow, no spaces. These identifiers are used to construct the presentation layer: they represent the external names for states, inputs, and outputs. Internal consecutive non-negative numbers for these will be created internally.
6. If a table field for some state  $s$  on some input  $x$  is empty, the resulting FSM will be partial.
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.

	A	B	C	D	E
1		<b>e1</b>	<b>e2</b>	<b>e3</b>	<b>e4</b>
2	<b>Door_Up</b>	Door_closing/a1			
3	<b>Door_Down</b>	Door_opening/a2			
4	<b>Door_stopped_going_down</b>	Door_closing/a1			
5	<b>Door_stopped_going_up</b>	Door_opening/a2			
6	<b>Door_closing</b>	Door_stopped_going_down/a3	Door_Down/a3		Door_opening/a4
7	<b>Door_opening</b>	Door_stopped_going_up/a3		Door_Up/a3	

Figure 3.2: Tabular format for modelling DFSMs.

An example of an admissible model CSV-format looks as follows, it corresponds to the DFSM transition table shown in Fig. 3.2.

```
;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;
```

The CSV creator is implemented as class `FsmFromCsvCreator` with header file

```
src/creators/FsmFromCsvCreator.hpp
```

Listing 3.2 shows how to create a FSM from a CSV file.

---

Listing 3.2: Create a FSM from a CSV file.

```
1 #include "Dfsm.hpp"
2 #include "FsmFromCsvCreator.hpp"
3
4 using namespace std;
5 using namespace libfsmtest;
6
7 void main(int argc, char* argv[]) {
8
9     FsmFromCsvCreator creator("garage-door-controller.csv");
10    unique_ptr<Fsm> fsm = creator.createFsm();
11
12 }
```

---

### 3.1.3 Reading FSMs from File With Automated Format Identification

The two FSM-from-file generators described above have been abstracted in class

`FsmFromFileCreator`

contained in files

`src/libfsmttest/creators/FsmFromFileCreator.hpp, .cpp`

An instance of `FsmFromFileCreator` first identifies the file type provided by the user and then performs the FSM instantiation using the FSM-from-file creators described above. This allows for simpler code in programs using the `libfsmttest`. In particular, programs reading FSMs from different file formats need not require users to specify the format explicitly or add file identification code in the programs.

---

Listing 3.3: Create an FSM from file with automated format identification.

```
1 #include "FsmFromFileCreator.hpp"
2 #include "..."
3
4 using namespace std;
5 using namespace libfsmttest;
6
7 void main(int argc, char* argv[]) {
8
9     // First parameter of FsmFromFileCreator constructor:
10    // basename of the file(s) to be opened
11    // Secod (optional) parameter: FSM name to be used.
12    FsmFromFileCreator creator("garage-door-controller",
13                               "GDC");
14    unique_ptr<Fsm> fsm = creator.createFsm();
15
16    // ... now continue working with fsm ...
17
18 }
```

---

Consider the example in Listing 3.3. An instance of `FsmFromFileCreator` is created by providing just the basename

"garage-door-controller" of the file. The instance will then proceed as follows in the call to method `createFsm`.

1. First, it is checked whether a file with the given basename and extension `.fsm` exists. If this is the case, the FSM will be instantiated with the help of class `FsmFromRawCreator`, using the raw format. If `.in`, `.out`, `.state` files with the same basename exist as well, they will be used to create the FSM's presentation layer. If not, a presentation layer using just non-negative numbers for inputs, outputs, and states will be created internally.
2. If (1.) does not apply, the instance of `FsmFromFileCreator` looks for a file with the given basename and extension `.csv`. If such a file exists, an instance of `FsmFromCsvCreator` will be internally created, and the FSM will be instantiated from the CSV file.
3. Else, if file names are provided including the extension, the raw format is expected to be contained in a file with extension `.fsm`, and the CSV format is expected to be contained in a file with extension `.csv`. If an unknown or no extension is provided, the `FsmFromFileCreator` instance tries to guess the file type from the file content.

Recall from the description of the three FSM-from-file creators, that FSM files with incomplete transition information will lead to the instantiation of partial FSMs. In Section 3.2.1, the option to perform auto-completion, that is, to create complete FSMs from partial specifications, is described. There, a third optional parameter of the `FsmFromFileCreator`-constructor is described which allows for automated application of an auto-completion transformation.

## 3.2 FSM Creation by Transformation

All FSM transformation classes reside in directory

```
src/libfsmtest/creators/transformers.
```

### 3.2.1 Auto-Completion Transformers

When specifying partial FSMs, inputs are enabled or disabled in a state-dependent way. If an input is enabled in a state we also say that it is *defined* in that state, otherwise we say that it is *undefined* in that state. There are different interpretations of inputs that are undefined in some state<sup>2</sup>.

1. The specification model is *incomplete*, and it is unknown what should happen for the undefined inputs in the respective states.
2. The undefined inputs *cannot occur* in the respective states (for example, the input corresponds to an input button of a graphical user interface which is not visible in the respective state).
3. The undefinedness of the input stands for *this input is ignored in this state*, so it's just a shorthand notation. This corresponds to a self loop transition labelled by this input and a null output.
4. The undefinedness of the input stands for *this input is not allowed in this state, and its occurrence will cause an error output and a transition into an error state which is never left again*. This is another shorthand notation. All inputs occurring when the FSM is in the error state cause a null-output.

For the interpretations 3 and 4, two so-called **auto-completion** transformers have been provided by classes

- `ToAutoCompleteWithSelfLoopTransformer` and

---

<sup>2</sup>A comprehensive discussion of these interpretations can be found in [15].



- ToAutoCompleteWithErrorStateTransformer,

both implemented in files

```
src/libfsmtest/creators/transformers/ToAutoCompletedTransformer.hpp
src/libfsmtest/creators/transformers/ToAutoCompletedTransformer.cpp
```

In Fig. 3.4, it is shown how the transformers are instantiated and invoked.

---

Listing 3.4: Use of the auto-completion transformers.

```
1 #include "creators/transformers/ToAutoCompletedTransformer.hpp"
2 #include "..."
3
4 using namespace std;
5 using namespace libfsmtest;
6
7 void main(int argc, char* argv[]) {
8     // ... read partial FSM from file,
9     // unique pointer fsm ...
10
11     // Transform fsm to completely specified FSM, using
12     // the self-loop auto-completion.
13     // As external presentation of the {\tt null} output,
14     // use string "_null". Type directive 'auto' means that
15     // the new machine is again of type unique_ptr<Fsm>.
16     ToAutoCompleteWithSelfLoopTransformer selfLooper(fsm.get(), "_null");
17     auto selfLoopCompletedFsm = selfLooper.createFsm();
18
19     // Transform fsm to completely specified FSM, using
20     // the auto-completion with transition to an error state.
21     // As external presentation of the {\tt null} output,
22     // use string "_null". For the error output, use "_error".
23     // As error state name, use "ERROR"
24     ToAutoCompleteWithErrorStateTransformer
25         errorStater(fsm.get(), "_error", "_null", "ERROR");
26     auto errorStateCompletedFsm = errorStater.createFsm();
27
28     // Continue working with completely specified FSMs selfLoopCompletedFsm
29     // and errorStateCompletedFsm
30     // . . .
31
32 }
```

---

Quite often, it is already clear when reading an FSM from a file that an auto-completion should be performed. Therefore, the FsmFromFileCreator

offers the option to provide such a transformer when instantiating an `FsmFromFileCreator` object. This is shown in Listing 3.5.

---

Listing 3.5: Use of the auto-completion transformers.

```
1 #include "..."  
2  
3 using namespace std;  
4 using namespace libfsmtest;  
5  
6 void main(int argc, char* argv[]) {  
7  
8     // We know that file garage-door-controller.csv contains a  
9     // partial FSM definition, and we want the FSM to be  
10    // auto-completed with self-loops and null-outputs.  
11    // To this end, an instance of ToAutoCompleteWithSelfLoopTransformer  
12    // is provided to creator as third parameter.  
13    // The instance of ToAutoCompleteWithSelfLoopTransformer  
14    // gets a nullptr for the FSM, since this FSM first  
15    // has to be read from file.  
16    FsmFromFileCreator  
17        creator("garage-door-controller.csv",  
18                "garage-door-controller",  
19                make_unique<ToAutoCompleteWithSelfLoopTransformer>  
20                    (nullptr, "null"));  
21    unique_ptr<Fsm> fsm = creator.createFsm();  
22  
23    // ... continue working with fsm ...  
24  
25 }
```

---

### 3.2.2 Transformation to Prime Machines

Recall that a **prime machine** is an FSM that is initially connected, observable, and minimised.

**Transformation to the prime machine.** Quite often, it is desirable to transform a given arbitrary FSM into an initially connected, observable, and minimised FSM in one step. This task is performed by class `ToPrimeTransformer` in files

```
src/libfsmtest/creators/transformers/ToPrimeTransformer.hpp  
src/libfsmtest/creators/transformers/ToPrimeTransformer.cpp
```

This transformer converts the given FSM according to the following steps.

1. If necessary, the FSM is transformed into an initially connected one.
2. If necessary, the resulting machine is transformed into an observable FSM.
3. The resulting machine is minimised.

These steps are performed by class `ToPrimeTransformer` through the individual transformers described in the paragraphs below.

Listing. 3.6 shows how to use this transformation.

---

Listing 3.6: Transformation to prime machine.

```
1 #include "creators/transformers/ToPrimeTransformer.hpp"
2 #include "... "
3
4 using namespace std;
5 using namespace libfsmtest;
6
7 void main(int argc, char* argv[]) {
8     // ... read partial FSM from file,
9     // unique pointer fsm ...
10
11     // Define the transformer to prime FSM,
12     // providing the original FSM to the constructor
13     ToPrimeTransformer prime(fsm.get());
14     // Perform the transformation and obtain unique pointer to
15     // the prime FSM
16     auto primeFsm = prime.createFsm();
17
18     // Continue working with the prime FSM
19     // . . .
20
21 }
```

---

**Transformation to initially connected machine.** Recall that an FSM is initially connected if all states can be reached from the initial states by repeated application of the transition relation. The transformer implemented by class `ToInitiallyConnectedTransformer` in files

```
src/libfsmtest/creators/transformers/ToInitiallyConnectedTransformer.hpp
src/libfsmtest/creators/transformers/ToInitiallyConnectedTransformer.cpp
```

performs this transformation task.

Listing. 3.7 shows how to use this transformation.

---

Listing 3.7: Transformation to an initially connected FSM.

```
1 #include "creators/transformers/ToPrimeTransformer.hpp"
2 #include "... "
3
4 using namespace std;
5 using namespace libfsmtest;
6
7 void main(int argc, char* argv[]) {
8     // ... read partial FSM from file,
9     // unique pointer fsm ...
10
11     // Define the transformer to initially connected FSM,
12     // providing the original FSM to the constructor
13     ToInitiallyConnectedTransformer ict(fsm.get());
14     // Perform the transformation and obtain unique pointer to
15     // the initially connected FSM
16     auto primeFsm = prime.createFsm();
17
18     // Continue working with the initially connected FSM
19     // . . .
20
21 }
```

---

**Transformation to observableFSMs.** Any FSM can be transformed into a language-equivalent observable one [14].<sup>3</sup> To this end, class `ToObservableTransformer` in files

```
src/libfsmtest/creators/transformers/ToObservableTransformer.hpp
src/libfsmtest/creators/transformers/ToObservableTransformer.cpp
```

implements the standard algorithm for the transformation of arbitrary (partial or completely specified) FSM into observable ones with the same language. The transformer will throw an exception if the FSM is not initially connected. If the FSM is already observable, a copy will be returned by operation `createFsm()` (see Listing 3.8

Listing 3.8 shows how to use this transformation.

---

<sup>3</sup>The algorithm is applicable to both completely specified and partial machines. It is interesting to note, however, that in case of partial FSMs, the transformed FSM is not necessarily quasi-equivalent to the original FSM [7], and not necessarily a strong reduction of the original FSM [15].

---

Listing 3.8: Transformation to an observable FSM.

```
1 #include "creators/transformers/ToPrimeTransformer.hpp"
2 #include "... "
3
4 using namespace std;
5 using namespace libfsmtest;
6
7 void main(int argc, char* argv[]) {
8     // ... read partial FSM from file,
9     // unique pointer fsm ...
10
11     // Define the transformer to observable FSM,
12     // providing the original FSM to the constructor
13     ToObservableTransformer obs(fsm.get());
14     // Perform the necessary transformations and obtain
15     // the unique pointer to the observable FSM
16     auto fsmObs = obs.createFsm();
17
18     // Continue working with the observable FSM
19     // . . .
20
21 }
```

---

**Minimising observable, initially connected machines.** For minimising observable, initially connected machines, the transformer of class `ToMinimisedTransformer` is used. It is implemented in files

`src/libfsmtest/creators/transformers/ToMinimisedTransformer.hpp`, `.cpp`

This transformer throws exceptions if the FSM to be transformed is not initially connected or not observable.

Listing 3.9 shows how to use this transformation.

---

Listing 3.9: Transformation to minimised FSM.

```
1 #include "creators/transformers/ToPrimeTransformer.hpp"
2 #include "... "
3
4 using namespace std;
5 using namespace libfsmtest;
6
7 void main(int argc, char* argv[]) {
8     // ... read partial FSM from file,
9     // unique pointer fsm ...
10
11     // Define the transformer to minimised FSM,
12     // providing the original FSM to the constructor
13     ToMinimisedTransformer mintrans(fsm.get());
14     // Perform the necessary transformations and obtain
15     // the unique pointer to the minimised FSM
16     auto fsmMin = mintrans.createFsm();
17
18     // Continue working with the minimised FSM
19     // . . .
20
21 }
```

---

### 3.3 Random Creation and Mutation of FSMs

There are many use cases for randomly generated FSMs. All classes for random FSM creation and mutation of existing FSMs reside in directory

```
src/libfsmtest/creators/randoms.
```

Every class for random creation of FSMs inherits from the class `RandomFsmCreator`, which takes care of the initialisation of a presentation layer; see header file

```
src/libfsmtest/creators/randoms/RandomFsmCreator.hpp.
```

To implement other methods for random creation, this file should be inherited in a new concrete class. Classes for random mutation of FSMs should inherit the already discussed `Transformer` class: a basic concept of the `libfsmtest` class library is to never change an existing FSM, but to transform it into another one. Therefore, mutations are created by transforming the existing FSM into a different one.

#### 3.3.1 Creation of Random Completely Specified FSMs.

Recall that a FSM is **completely specified** if there exists an outgoing transition for every possible input in every state. FSMs, OFSMs, and DFSMs can be completely specified.

**Random Completely Specified FSM** The class to create random completely specified FSMs can be found in files

```
src/libfsmtest/creators/randoms/RandomCompletelySpecifiedFsm.hpp, .cpp
```

For a given numbers of states, inputs and outputs the creator will then generate a completely specified FSM. During the generation it is also asserted that the FSM generated is initially connected. For each state, the FSM will have at least one, at most three outgoing transitions for each input. Listing 3.10 shows how to use this creator.



---

Listing 3.10: Create a Random Completely Specified FSM.

```
1 #include "RandomCompletelySpecifiedFsmCreator.hpp"
2 #include "... "
3
4 using namespace std;
5 using namespace libfsmtest;
6
7 void main(int argc, char* argv[]) {
8
9     // First parameter of RandomCompletelySpecifiedFsmCreator constructor:
10    // Number of states the FSM will have.
11    // Second parameter: Number of inputs the alphabet will have.
12    // Third parameter: Number of outputs the alphabet will have.
13    // Fourth (optional) parameter: FSM name to be used.
14    RandomCompletelySpecifiedFsmCreator creator(5, 7, 4, "random");
15    unique_ptr<Fsm> fsm = creator.createFsm();
16
17    // ... now continue working with fsm ...
18
19 }
```

---

If it is required to produce an *observable*, but potentially nondeterministic FSM at random, the machine is created as shown in Listing 3.10, but then transformed into an observable one, using the transformer to observable FSMs described in Section 3.2. Using FSM method `isObservable()`, it should be checked whether such a transformation is needed, since the FSM generated at random may already be observable. Note, however, that the transformed OFSM will generally not have the same number of states as requested in the random creation.

**Random Completely Specified DFSM.** The class to create random completely specified DFSMs can be found in files

`src/libfsmtest/creators/randoms/RandomCompletelySpecifiedDfsm.hpp, .cpp`

For a given numbers of states, inputs and outputs the creator will then generate a completely specified DFSM. Each state will have exactly one outgoing transition for each input. Listing 3.11 shows how to use this creator. Recall that DFSMs are automatically observable.

---

Listing 3.11: Create a Random Completely Specified FSM.

```
1 #include "RandomCompletelySpecifiedDfsmCreator.hpp"
2 #include "... "
3
4 using namespace std;
5 using namespace libfsmtest;
6
7 void main(int argc, char* argv[]) {
8
9     // First parameter of RandomCompletelySpecifiedDfsmCreator constructor:
10    // Number of states the DFMS will have.
11    // Second parameter: Number of inputs the alphabet will have.
12    // Third parameter: Number of outputs the alphabet will have.
13    // Fourth (optional) parameter: DFMS name to be used.
14    RandomCompletelySpecifiedDfsmCreator creator(5, 7, 4, "random");
15    unique_ptr<Fsm> fsm = creator.createFsm();
16
17    // ... now continue working with dfsm ...
18
19 }
```

---

### 3.3.2 Random Removal of Transitions

The class to randomly remove transitions from FSMs can be found in files

src/libfsmtest/creators/randoms/RandomTransitionRemoval.hpp, .cpp

Given a number of transitions to remove, this transformer will randomly choose a state that has at least one transition, then randomly choose one of those transitions and remove it. The transformer will stop and return the altered FSM once the number of transitions to remove has been reached, or there are no transitions left to remove. Listing 3.11 shows how to use this transformer.

---

Listing 3.12: Transformation to remove random transitions of an FSM.

```
1 #include "creators/randoms/RandomTransitionRemoval.hpp"
2 #include "...
3
4 using namespace std;
5 using namespace libfsmtest;
6
7 void main(int argc, char* argv[]) {
8     // ... create or retrieve FSM
9     // unique pointer fsm ...
10
11     // Define the transformer to remove random transitions,
12     // providing the original FSM to the constructor
13     // as well as the number of transition to remove
14     // and an (optional) suffix to append to the name.
15     RandomTransitionRemoval remover(fsm.get(), 5);
16     // Perform the necessary transformations and obtain
17     // the unique pointer to the altered FSM
18     auto fsmRm = remover.createFsm();
19
20     // Continue working with the altered FSM
21     // . . .
22
23 }
```

---

# Chapter 4

## Test Suite Generation

### 4.1 Test Generation Frame, Test Generation Method, and Test Suite

To generate test suites using different methods (like W-Method, H-Method, etc.), three re-usable concepts have been implemented in `libfsmtest`.

- A **test generation frame** is instantiated to handle the test suite generation task in a way that is independent on the specific method.
- **Test generation methods** are always implemented as FSM-visitors. The test generation frame receives a pointer to the test method to be used as a parameter of its constructor.
- The test generation frame provides an operation to activate the test generation process. This produces a **test suite** which can be processed further in the program or written to disk.

Consider Listing 4.1 to inspect how these concepts are applied in practise.

---

Listing 4.1: Application of test generation frame, method, and suite.

```
1 #include "..."  
2  
3 using namespace std;  
4 using namespace libfsmtest;  
5  
6 int main(int argc, char* argv[]) {  
7  
8     // Create FSM from file  
9     FsmFromFileCreator creator("f.fsm","f");  
10    auto fsm = creator.createFsm();  
11  
12    // Create test generation frame:  
13    // "SUITE-W-0" is the name of the test suite.  
14    // The FSM is moved into the generation frame,  
15    // since it serves there as the reference model  
16    // to create test cases from.  
17    // As generation method, an instance of the WMethod  
18    // is provided. This instance needs the maximal number  
19    // of additional states the implementation might contain.  
20    int numAddStates = 0;  
21    TestGenerationFrame genFrame("SUITE-W-0",  
22                                move(fsm),  
23                                make_unique<WMethod>(numAddStates));  
24  
25    // Generate the test suite and write it to file  
26    genFrame.generateTestSuite();  
27    genFrame.writeToFile();  
28 }
```

---

When method `generateTestSuite()` is invoked (line 26), the test generation frame `genFrame` created in this example will activate the test generation internally according to the visitor pattern: the FSM-visitor instance of class `WMethod` is provided as parameter in an `accept(...visitor instance...)` call on the reference FSM. This is all hidden from users of `libfsmtest`, but must be studied before creating and adding your own test generation methods to `libfsmtest`.

The effect of the `genFrame.writeToFile()` call is as follows.

1. The test suite consisting of a number of input traces is written into a text file named as the test suite with extension `.txt`. The input events are represented in external form, as specified in the FSM's presentation layer. For the example in Listing 4.1, the demonstration

program usage-demo (see Chapter 2) produces the following test suite in file SUITE-W-0.txt. Each line represents one test case specified by the associated inputs to be exercised on the system under test.

```
1 e1, e1, e1
2 e1, e1, e2
3 e1, e2, e1, e1
4 e1, e2, e1, e2
5 e1, e2, e2, e1
6 e1, e2, e2, e2
7 e1, e2, e3, e1
8 e1, e2, e3, e2
9 e1, e2, e4, e1
10 e1, e2, e4, e2
11 e1, e3, e1
12 e1, e3, e2
13 e1, e4, e1, e1
14 e1, e4, e1, e2
15 e1, e4, e2, e1
16 e1, e4, e2, e2
17 e1, e4, e3, e1
18 e1, e4, e3, e2
19 e1, e4, e4, e1
20 e1, e4, e4, e2
21 e2, e1
22 e2, e2
23 e3, e1
24 e3, e2
25 e4, e1
26 e4, e2
```

2. The reference FSM is written in raw format (see Section 3.1.1) to files <FSM-name>.fsm, <FSM-name>.in, <FSM-name>.out, and <FSM-name>.state. These FSM files, together with the test cases file, need to be provided to the test harness described in Chapter 5, when running the generated suite against a software under test. The harness will use the reference FSM as a test oracle.

## 4.2 Available Test Generation Methods

In the current version, the following generation methods listed in Table 4.1 have been provided, more are yet to come in the near future. All methods are implemented in files located in

src/libfsmtest/visitors

Each method may pose requirements for admissible reference FSMs to fulfill. These requirements are listed in Table 4.2. If the method is called with a reference FSM that violates one of the method’s application requirements, the implementing visitor will throw an exception.

Table 4.1: Test generation methods currently available in libfsmtest.

Method	Class Name	Files	References
W-Method	WMethod	WMethod.hpp, .cpp	[1], [18], [14, Section 4.6]
WP-Method	WPMethod	WPMethod.hpp, .cpp	[11], [14, Section 4.8.1]
H-Method	HMethod	HMethod.hpp, .cpp	[3], [14, Section 4.7]
SPYH-Method	SPYHMethod	SPYHMethod.hpp, .cpp	[16]
T-Method	TMethod	TMethod.hpp, .cpp	[13], [14, Section 4.3]
Safety-complete H-Method	SHMethod	SHMethod.hpp, .cpp	[9], [8]
Classical (non-adaptive) state counting method	ClassicalStateCountingMethod	ClassicalStateCountingMethod.hpp, .cpp	[6]
Strong State Counting Method	StrongStateCountingMethod	StrongStateCountingMethod.hpp, .cpp	[15]

Table 4.2: Requirements for the test generation methods currently available in `libfsmtest`.

Method	Requirements
W-Method	Requires the existence of a characterisation set. This is guaranteed for completely specified reference FSMs.
WP-Method	Requires the existence of a characterisation set and state identification sets. This is guaranteed for completely specified reference FSMs.
H-Method	Requires harmonized traces. This is guaranteed for deterministic or completely specified reference FSMs.
SPYH-Method	Requires a completely specified and deterministic reference FSM.
T-Method	Requires a completely specified reference FSM.
Safety-complete H-Method	Requires harmonized traces. This is guaranteed for deterministic or completely specified reference FSMs.
Classical (non-adaptive) state counting method	Requires a completely specified reference FSM.
Strong State Counting Method	Reference FSMs are required to be observable, as the general approach to make any FSM observable does not preserve strong reduction.

When using a specific method, the test generation frame (see lines 21—23 in Listing 4.1) gets this method's class name as type parameter in the

```
1 make_unique< _type_ >(numAddStates)
```

instantiation command. For example, when the H-Method should be used, the generation frame in lines 21—23 of Listing 4.1 is created with statement

```
1 TestGenerationFrame
2     genFrame("SUITE-H-0", // Any test suite name which makes
3                 // clear that the H-Method has been used
4                 move(fsm),
5                 make_unique<HMethod>(numAddStates));
```



## 4.3 Test Generation Methods Using Abstraction

In the context of **property-oriented testing** [12], we are no longer focused on verifying a conformance relation between reference model and implementation. Instead, it has to be tested whether the SUT fulfils certain properties that are also fulfilled by the reference model. Properties are conditions about inputs, outputs, and their causal ordering. In practical applications, properties are often equivalent to, or derived from requirements to be fulfilled by the implementation. The most general way to specify properties is by means of a temporal logic such as LTL [17]. This, however, is currently not yet supported by `libfsmtest`.

A slightly less general, but still quite powerful way is to specify properties by means of **FSM abstractions**. The theory behind this has been investigated in [8, 9]. Note that it is applicable to deterministic, completely specified FSMs only. We introduce the – quite intuitive – concept here by means of an example.

**Example 1.** Consider the completely specified DFSM  $A$  shown in Fig. 4.1 with input alphabet  $\Sigma_I = \{c_1, \dots, c_6\}$  and output alphabet  $\Sigma_O = \{d_0, \dots, d_4\}$ . Suppose we wish to test whether the implementation satisfies the following property which is obviously fulfilled by  $A$ .

**Property 1.** *If the inputs are always in range  $\{c_1, c_2, c_3\}$  then the outputs will always be in range  $\{d_0, d_1\}$ .* (\*)

Expressed in LTL, this property is specified by

$$G(c_1 \vee c_2 \vee c_3) \Rightarrow G(d_0 \vee d_1),$$

but we will not need this for the FSM abstraction approach. Instead, we specify an abstracted FSM  $\alpha(A)$  as follows:

1. The input alphabet of  $\alpha(A)$  equals that of  $A$ , that is,  $\{c_1, \dots, c_6\}$ ,
2. the output alphabet of  $\alpha(A)$  is  $\{e_0, e_1\}$ , where  $e_0$  stands for “ $A$ -output is in  $\{d_0, d_1\}$ ” and  $e_1$  stands for “ $A$ -output is not in  $\{d_0, d_1\}$ ”,

3. the states of  $\alpha(A)$  and the initial state are the same as in  $A$ , and
4. the transition relation  $\alpha(R)$  of  $\alpha(A)$  is obtained from the transition relation  $R$  of  $A$  as

$$\begin{aligned}\alpha(R) = & \{(s, x, e_0, s') \mid \exists y \in \{d_0, d_1\} \cdot (s, x, y, s') \in R\} \cup \\ & \{(s, x, e_1, s') \mid \exists y \in \{d_2, d_3, d_4\} \cdot (s, x, y, s') \in R\}.\end{aligned}$$

Intuitively speaking,  $\alpha(A)$  has the same transition graph topology as  $A$ , and the transitions are labelled by the same inputs as in  $A$ . The outputs, however, are abstracted to the new values  $e_0, e_1$ , depending on whether the corresponding  $A$ -output is in  $\{d_0, d_1\}$  or not. This abstraction machine  $\alpha(A)$  is shown in Fig. 4.2.

Since the abstracted FSM has fewer outputs, it distinguishes fewer states than  $A$ : indeed, the minimised machine of  $\alpha(A)$  only has two states, as shown in Fig. 4.3. Obviously,  $\alpha(A)$  fulfils the **abstracted property**

**Property 1a.** *If the inputs are always in range  $\{c_1, c_2, c_3\}$  then the output will always be  $e_0$ .* (\*\*)

Now the theory developed in [8, 9] states that we can apply the **Safety-complete H-Method (SH-Method)** to derive an exhaustive test suite which is guaranteed to fail on an implementation violating property (\*), because the abstraction FSM consistently abstracts this property to the one specified in (\*\*). The SH-Method differs from the H-Method in the fact that distinguishing traces  $\gamma$  are appended to certain traces  $\alpha, \beta$  already contained in the test suite only if the states reached by  $\alpha$  and  $\beta$ , respectively, are also distinguishable in the abstracted FSM. The “normal” H method appends  $\gamma$  to  $\alpha$  and  $\beta$  already if these reach states that are distinguishable in  $A$ <sup>1</sup>.

As a consequence, the SH-Method may result in significantly fewer test cases than the H-Method. For the FSM example  $A$  discussed here, the Safety-H-Method and the conventional H-Method produce the following

---

<sup>1</sup>States  $q, q'$  that are distinguishable in  $\alpha(A)$  are by construction also distinguishable in  $A$ , but not every pair of states distinguishable in  $A$  is distinguishable in  $\alpha(A)$ .

numbers of test cases, depending on the maximal value  $a$  of additional states assumed for the implementation.

	$a = 0$	$a = 1$	$a = 2$	$a = 3$
SH-Method test suite size	21	126	756	4536
H-Method test suite size	28	158	982	5888
Ratio	0.75	0.79	0.77	0.77

Further examples are presented in [8, 9].

□

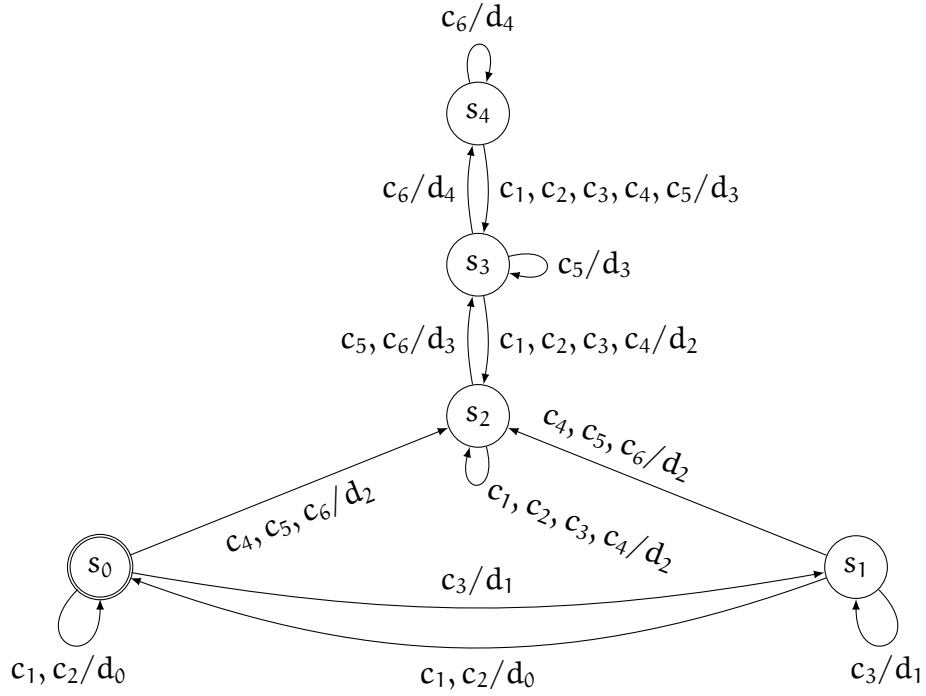


Figure 4.1: FSM A with different regions: once state  $s_2$  has been reached, the FSM will only visit states in  $\{s_2, s_3, s_4\}$ ; it will never return to  $s_0$  or  $s_1$ .

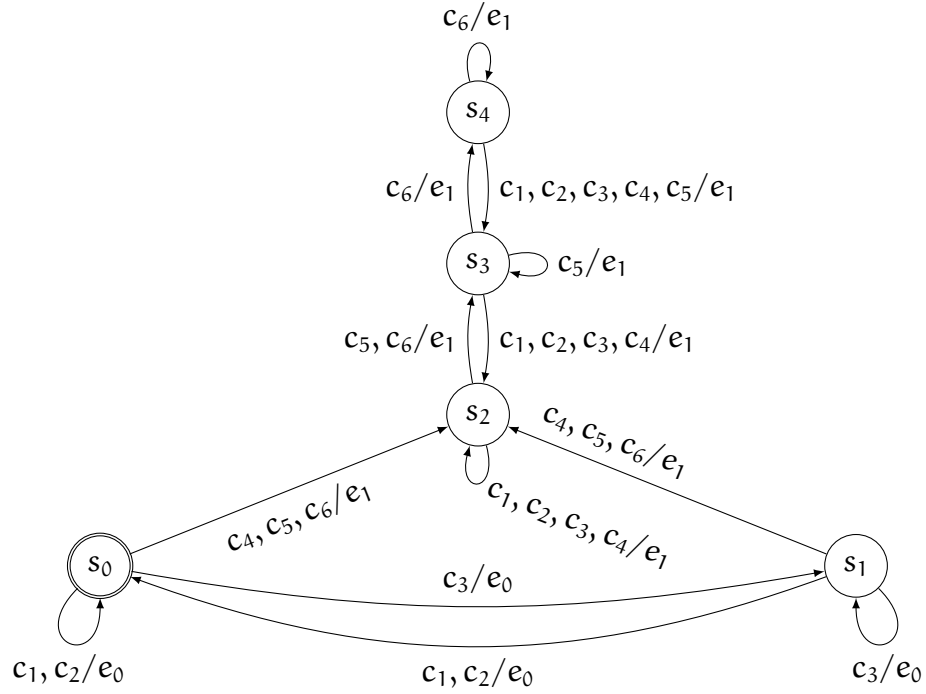


Figure 4.2: FSM abstraction  $\alpha(A)$  of the original FSM  $A$  shown in Fig. 4.1.

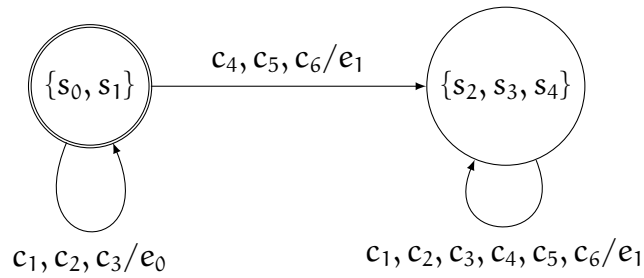


Figure 4.3: Minimised FSM associated with  $\alpha(A)$  from Fig. 4.2.

The abstraction concept described above is implemented by the SH-Method (class SHMethod). When using abstraction machines, the test generation frame is created with an additional parameter:

```

1 // ... read reference FSM and abstraction FSM ...
2 // referenceFsm is a unique pointer to the reference FSM.
3 // abstractionFsm is a unique pointer to the abstraction FSM.
4 TestGenerationFrame
5     genFrame("SAFETY-H-METHOD-FSBRTSX",
6             move(referenceFsm),
7             make_unique<SHMethod>(numAdditionalStates),
8             move(abstractionFsm));
9
10 // Generate the test suite and write it to file
11 genFrame.generateTestSuite();
12 genFrame.writeToFile();

```

Observe that the SH-Method is exhaustive, but not sound. This means that an implementation can fail a test suite even though it correctly implements the property for which the abstraction FSM has been created. In this case, the test suite has uncovered a violation of language equivalence, which we consider as a good thing, because in principle, the SUT should really be equivalent to the reference model, though we are currently only interested in a certain property. Test suites generated by the SH-Method will never fail for implementations that are language equivalent to the reference model. In [8] it has been shown for a specific type of properties that it is possible to create complete (i.e. exhaustive and sound) test suites that only fail if the specified property is violated. This insight, however, is of theoretical value only, because these test suites may become *larger* than suites establishing language equivalence.

The fact that *two* FSMs are required for the SH-Method deserves an explanation. In principle, it would be possible to use the abstracted model itself as reference machine. However, the difference  $\alpha$  between the number of states in the minimised reference machine and the potential number of states in the minimised DFMS representing the implementation behaviour would be larger than for the original reference machine. The test suite size, however, grows exponentially in  $\alpha$ . Therefore, it is better to use the original machine (*A* in the example above) with a smaller value of  $\alpha$ .

Furthermore, note that it is not always the case that utilisation of an abstraction FSM will reduce the test suite size in comparison to testing for language equivalence. The following heuristics is applicable to decide this.

- The Safety-H-Method never produces more test cases than the H-Method.

- If the prime machine of the FSM abstraction still has the same size as the prime machine of the reference model, then no reduction is to be expected.
- If all states of the reference model's prime machine can be distinguished by very few very short traces, then the test case reduction to be achieved by the Safety-H-Method can be expected to be quite small, even if the prime machine of the FSM abstraction has fewer states than that of the reference model.
- If the reference FSM contains a region that is of no relevance for the property to be checked, and if this region can never be left once entered, the test suite size reduction achieved by the SH-Method grows with the size of this region.
- The ratio "*number of test cases generated by SH-Method / number of test cases generated by H-Method*" does not change significantly with the number  $a$  of potential additional states in the implementation.

In any case, the test suites can be calculated beforehand, and if their size is nearly identical, it is more advisable to test for language equivalence, since this guarantees that *all* properties fulfilled by the reference model are also fulfilled by the implementation.

Finally, note that the FSM abstraction and the resulting test suite created by the SH-Method are not only applicable to a single property, but to *all* properties captured by the same abstraction FSM. This fact is well-known from the field of model checking. If a Kripke structure has a labelling function  $L$  mapping concrete states  $s$  to sets  $L(s) \subseteq AP$  of atomic propositions that are fulfilled in this state, then the resulting Kripke structure can be used for property checking of *all* temporal formulas (LTL, CTL, CTL\*) over atomic propositions from  $AP$  [2].

**Example 2.** Consider the following property of  $A$  from Example 1 which is captured by the same FSM abstraction  $\alpha(A)$ .

**Property 2.** *After an output in  $\{d_2, d_3, d_4\}$  has been produced, there will never be another output from  $\{d_0, d_1\}$ .*

Using LTL, this property would be expressed as

$$\mathbf{G}((d_2 \vee d_3 \vee d_4) \Rightarrow \mathbf{G}(\neg d_0 \wedge \neg d_1)).$$

This property is encoded in  $\alpha(A)$  as well, since it can be expressed by

**Property 2a.** *After output  $e_1$  has been produced, there will never be another output  $e_0$ .*

The test suite created by the SH-Method for Property 1 from Example 1 is also exhaustive for Property 2.  $\square$

# Chapter 5

## Using Generator, Checker, and Test Harness – the Workflow

### 5.1 A Sample Test Campaign

Throughout this chapter, we work with a simple model-based testing campaign, using an example originally introduced by Paul C. Jorgensen in [10].

#### 5.1.1 Reference Model Description

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. 5.1. 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, respectively. As inputs, the GDC receives a command “button pressed” (e1) from a remote control device, and two events “door reaches position down” (e2) and “door reaches position up” (e3) from two door position sensors. 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.



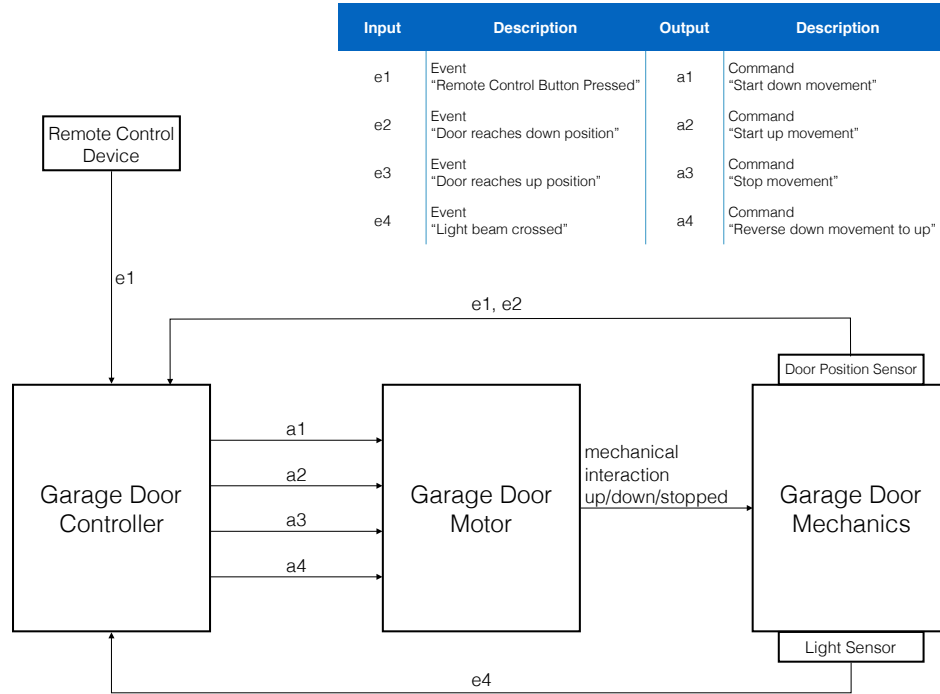


Figure 5.1: Garage door controller and its operational environment.

The expected behaviour of the GDC is modelled by the FSM in Fig. 5.2. In the initial state `Door_Up`, the door is expected to be in the UP position, and the “button pressed” event `e1` from the remote control triggers a “Start down movement” command `a1` to the motor. The GDC transits to state `Door_closing`. In this 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 `Door_opening`. During down movement in state `Door_closing`, another occurrence of the `e1`-event leads to a “Stop movement” command `a3` to the motor, and the controller transits to state `Door_stopped_going_down`. From there, the downward movement is resumed (output `a1`), as soon as another `e1`-command is given.

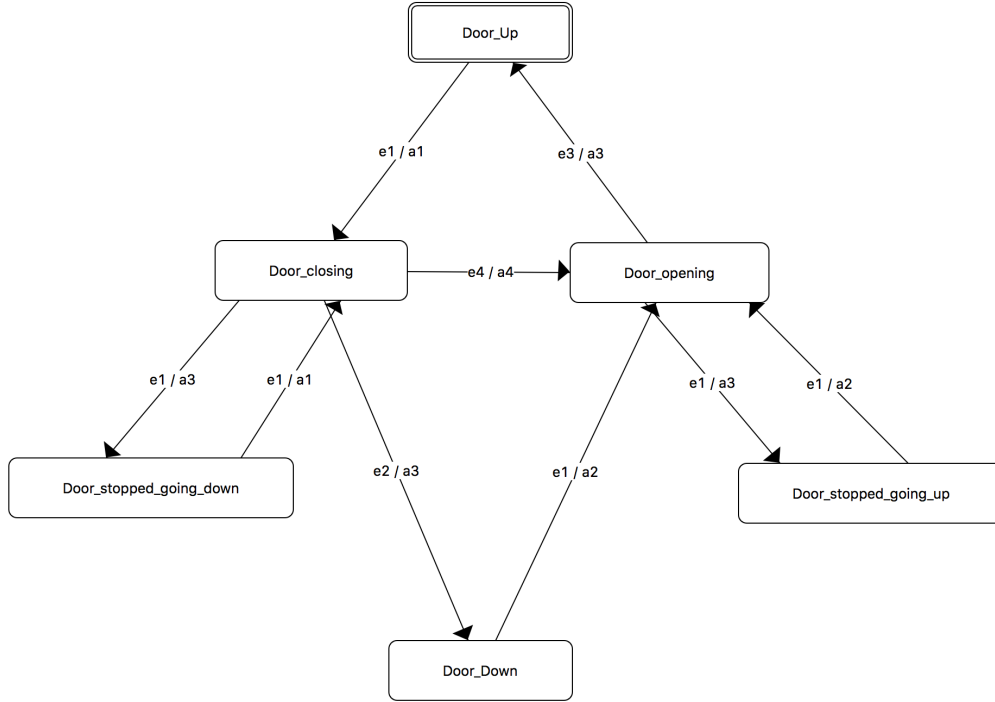


Figure 5.2: Behaviour of the garage door controller, modelled by a DFSM.

When the door sensor signals that the door has reached the down position (e2), the motor is stopped with command a3, and the controller transits into state Door\_down. From this state, 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. 5.3, the same DFSM is modelled by means of a transition table.

The missing transitions in each state have the “self-loop-with null outputs” interpretation, as explained in Section 3.2.1. Therefore, the DFSM is meant to be completely specified. When reading the DFSM from input files where the self-loop transitions are missing, the auto completion transformer `ToAutoCompleteWithSelfLoopTransformer` introduced in Section 3.2.1 needs to be applied.

	A	B	C	D	E
1		<b>e1</b>	<b>e2</b>	<b>e3</b>	<b>e4</b>
2	<b>Door_Up</b>	Door_closing/a1			
3	<b>Door_Down</b>	Door_opening/a2			
4	<b>Door_stopped_going_down</b>	Door_closing/a1			
5	<b>Door_stopped_going_up</b>	Door_opening/a2			
6	<b>Door_closing</b>	Door_stopped_going_down/a3	Door_Down/a3		Door_opening/a4
7	<b>Door_opening</b>	Door_stopped_going_up/a3		Door_Up/a3	

Figure 5.3: Tabular format for modelling DFSMs.

Note that the DFSM in Fig. 5.2 is not minimal; it has been represented in this form to optimise its readability. The equivalent minimised machine is shown in Fig. 5.4. This has been constructed using the prime machine transformer described in Section 3.2.2. The output graph shown in Fig. 5.4 has been created by using the ToDotFileVisitor described in Chapter 6. This produces files in the so-called .dot-format, from which the GraphViz<sup>1</sup> tool creates graph representations.

In main program file `usage_demo.cpp`, these transformations have been programmed in procedure `demo_transformToPrimeMachineGdc()`.

---

<sup>1</sup><http://www.graphviz.org>

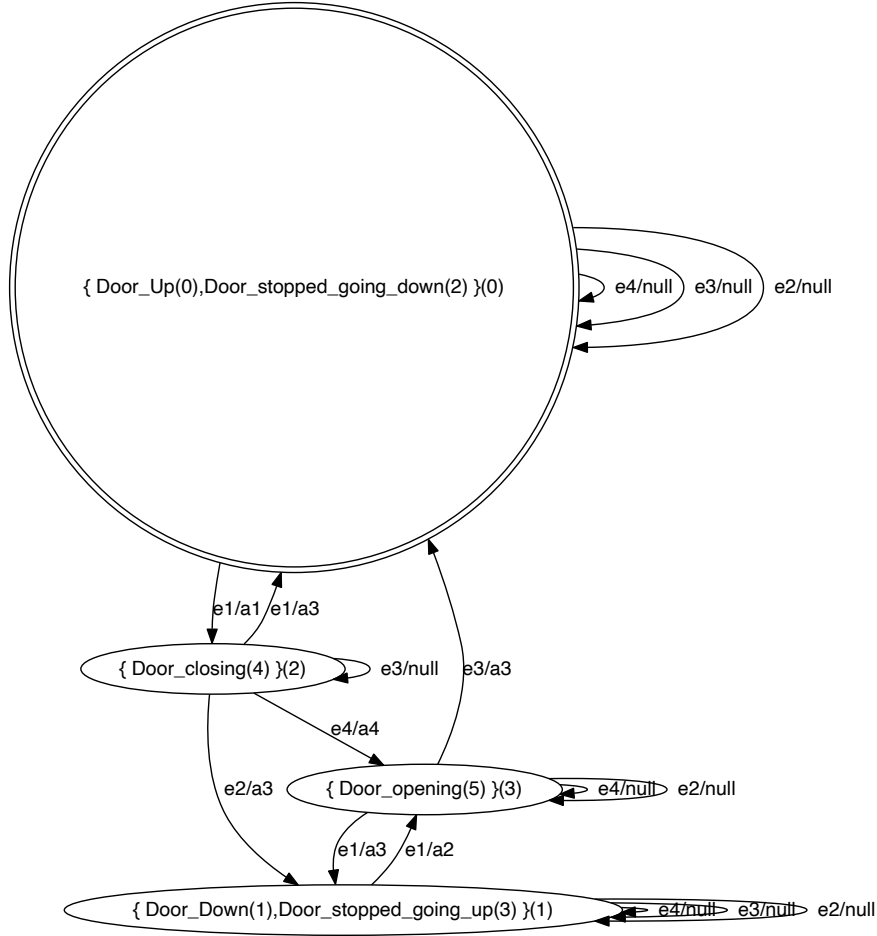


Figure 5.4: Minimised, auto-completed DFSM, equivalent to the GDC model from Fig. 5.2.

### 5.1.2 GDC System Under Test

A sample implementation in C++ is given in the FSM Library, directory `src/harness/example`, in file `gdclib.cpp`; the public operation interfaces are specified in `gdclib.hpp` 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_t;
15
16 extern void gdc_reset();
17 extern gdc_outputs_t gdc(gdc_inputs_t x);

```

The GDC expects its inputs in enumeration format `gdc_inputs_t` and returns actions to the motor in format `gdc_outputs_t`. The implementation in `gdclib.cpp` follows the state machine programming paradigm and is straightforward, so that no further comments are needed.

In Section 5.4 it is shown how test suites generated from the GDC model discussed in Section 5.1.1 can be executed against this C<sup>++</sup> application, using the test harness provided by `libfsmttest`.

## 5.2 Using the Test Generator

Users only interested in generating test suites with the existing methods available in the `libfsmttest` class library do not need to write their own programs with instances of test generation frame and test generation visitors, as described in Chapter 4. Instead, they can invoke the program generator which is created from the library and source file

`src/generator/generator.cpp`

when building the `libfsmttest` class library<sup>2</sup>.

The generator program is invoked from its directory as follows

```
1 ./generator [<options>] \  
2             <test suite name> \  
3             <Path to reference model> \  
4             [<Path to abstraction model>]
```

Parameter `<test suite name>` specifies the name of the test suite to be generated. On termination of the generator, the test cases file containing the input sequences to be exercised on the system under test is then named

```
1     <test suite name>.txt
```

and stored in the directory from where generator has been invoked.

Parameter `<Path to reference model>` gives the path and basename of the reference model. The generator uses the `FsmFromFileCreator` described in Section 3.1.3 internally, so file extensions and file types are determined automatically.

Optional parameter `<Path to abstraction model>` gives the path and basename of the abstraction model. This is only used when applying the safety-complete H method (see Section 4.3).

The generator produces a test suite according to the following options.

1. If no options are provided, the W-Method is applied with 0 additional states assumed for the implementation (see variable `numAddStates` in Listing 4.1).

---

<sup>2</sup>Recall from README.md that the executables reside in sub-directories of the build directories `build.Debug`, `build.Release`, or `xcodebuild.Debug`. For example, the generator program has path `build.Release/generator/generator`.

2. `-a<number>` sets the value of additional states to be assumed for the implementation to `<number>`. This option is used in combination with the options selecting the test generation method described below.
3. `-cself` specifies that an auto completion transformation should be applied on the FSM input files with self loops and null outputs, as described in Section 3.2.1. This option is used in combination with the options selecting the test generation method described below.
4. `-cerror` specifies that an auto completion transformation should be applied on the FSM input files with transition to an error state and error output, as described in Section 3.2.1. This option is used in combination with the options selecting the test generation method described below.
5. `-null <null string>` specifies the name of the null output, as used in an auto-completed FSM's presentation layer. This option is only useful if one of the options `-cself` or `-cerror` have been provided as well. If this option is not provided, the default value `_null` will be chosen for the null output.
6. `-errorout <error output string>` specifies the name of the error output, as used in an auto-completed FSM's presentation layer. This option is only useful if the option `-cerror` has been provided as well. If this option is not provided, the default value `_error` will be chosen for the error output.
7. `-errorstate <error state string>` specifies the name of the error state, as used in an auto-completed FSM's presentation layer. This option is only useful if the option `-cerror` has been provided as well. If this option is not provided, the default value `ERROR` will be chosen for the error output.

The following options select the test generation method to be used; only one of them may be chosen when invoking the generator.

1. `-w` – selection of the W-method

2. `-wp` – selection of the Wp-method
3. `-h` – selection of the H-method
4. `-spyh` – selection of the SPYH method
5. `-csc` – selection of the classical state counting method
6. `-ssc` – selection of the strong state counting method
7. `-sh` – selection of the safety-complete H-method.  
This is the only option requiring the additional parameter  
<Path to abstraction model>.

**Example 3.** For generating a test suite for the garage door controller (GDC) described in Section 5.1, we first note that raw file specifications of the GDC exist in the `resources/` directory, files

`garage.fsm`, `garage.in`, `garage.out`, `garage.state`

The `fsm`-file contains all transitions for a complete DFMSM, so no auto-completion directives are required when calling the generator. Assume that we wish to use the H-Method for test suite generation under the hypothesis that the software under test has at most 2 extra states.<sup>3</sup> The name of the test suite should be ‘SUITE-GDC-H-2’. Activating the generator from its build directory, say,

`libfsmtest/build.Release/generator/`

requires command

```
1 ./generator -h -a2 "SUITE-GDC-H-2" ../../resources/garage
```

On termination, the output

---

<sup>3</sup>More precisely, we assume that the unknown minimised DFMSM representing the true behaviour of the software under test has at most 2 additional states in comparison to the minimised GDC reference model.



```
Test generation completed.
Number of test cases: 277
Total length          : 1493
Test case file        : SUITE-GDC-H-2.txt
```

is written to the console. The test case file contains 277 lines, each line representing the inputs for one test case. The "Total length" is number of input events summed up over all test cases in the suite.

The generator creates the following files in the directory from where it has been activated:

1. File SUITE-GDC-H-2.txt containing the test cases. Its first lines look like this (recall that e1, e2, ... are inputs to the GDC, as described Section 5.1):

```
e1, e1, e1, e1, e1
e1, e1, e1, e2, e1
. . .
e1, e2, e1, e1, e1, e1
e1, e2, e1, e1, e1, e2
e1, e2, e1, e1, e2, e1
e1, e2, e1, e1, e3, e1
. . .
```

2. Files

```
garage.fsm, garage.in, garage.out, garage.state
```

because these represent the reference FSM which is part of the test suite, where it serves as the test oracle.

When using the generated test suite in the checker or test harness, described in Section 5.3 and Section 5.4, respectively, these files have to be supplied to the respective tool. □

## 5.3 Using the Checker

Users who intend to apply a test suite to a given FSM do not need to write their own programs to do so. Instead, they can invoke the program checker which is implemented in the source file

```
src/checker/checker.cpp
```

and generated when building the `libfsmtest`. Currently this checker supports test suite application with respect to testing for language equivalence.

The checker program is invoked from its directory as follows:

```
./checker [options]
    <Path to the test suite>
    <Path to the reference model>
    <Path to the FSM to test>
```

The parameter `<Path to the test suite>` specifies the file where the test suite to be applied is read from. For reference on test suite generation see [Section 5.2](#).

Parameter `<Path to the reference model>` gives the path and base-name of the reference model. The checker uses the `FsmFromFileCreator` described in [Section 3.1.3](#) internally, so file extensions and file types are determined automatically.

The path and basename of the model to apply the test suite to has to be given as the parameter `<Path to the FSM to test>`. As with `<Path to the reference model>`, the checker uses the `FsmFromFileCreator`, so the file extensions and file types are determined automatically.

With these three mandatory parameters, the checker tries to read all given necessary files and throws an exception if any of those are not available<sup>4</sup>.

If all files could be read successfully, a final check ensures that both the reference FSM and the FSM to test have the same input and output

---

<sup>4</sup>Note that the current implementation will warn about an unsupported file format if any of the given FSMs could not be read, even if that is due to missing files.

alphabets<sup>5</sup>.

Possible options for the checker are the options `-cself`, `-cerror`, `-null`, `-errorout` and `-errorstate` as described in Section 5.2. These options control whether, which and how an auto completion transformer shall be applied to the SUT model before checking it.

When all parameters have been parsed successfully, the input sequences in the test suite will be evaluated on both the reference FSM and the FSM to test. For both FSMs and for each input sequence in the test suite the set of produced output traces is determined, and differences in these sets are examined. For each output sequence produced by one of the FSMs that is not produced by the other FSM in response to the same input sequence, the checker determines the output sequence with the longest common prefix and prints both sequences.

```
FAILURE: SUT implements unspecified output trace 0,0,2,2
         for input trace 0,0,1,1
         Closest match diverges at step 4: 1
FAILURE: SUT does not implement output trace 0,2,1,1
         for input trace 0,1,1,1
         Closest match diverges at step 3: 2,2
         (Expected: 1,1)
FAILURE: SUT implements unspecified output trace 0,2,2,2
         for input trace 0,1,1,1
         Closest match diverges at step 3: 1,1
FAILURE: SUT does not implement output trace 1,0,2,0,2
         for input trace 1,0,1,0,1
```

However, if the sets of output sequences produced by both machines agree for all input sequences in the test suite, the checker notes this as PASS and exits.

**Example 4.** For checking an implementation of the garage door controller (GDC) described in Section 5.1, we note again that raw file specifications of the GDC exist in the `resources/` directory, files

---

<sup>5</sup>The current implementation even requires both the input and output alphabet pairs to define the symbols in the same order.

```
garage.fsm, garage.in, garage.out, garage.state
```

Furthermore, we will use the test suite generated in Section 5.2, ‘‘SUITE-GDC-H-2’’ to check a mutated implementation. To generate this mutation we copy the GDC FSM files, rename the copies to

```
garage-mutant.fsm, garage-mutant.in,  
garage-mutant.out, garage-mutant.state
```

and modify the `garage-mutant.fsm` file. In this example, we remove the transition from the state `Door_stopped_going_down` to the state `Door_closing` and change the output on the self-loop of `Door_Up` triggered by input `e3` from `null` to `a3`.

Assuming the current working directory contains the checker executable and that the `garage`, `garage-mutant` and test suite files are in the directory `/tmp`, we invoke the checker as follows

```
1 ./checker /tmp/SUITE-GDC-H-2 /tmp/garage /tmp/garage-mutant
```

On termination, the checker prints numerous lines beginning with `FAILURE:`, as the SUT clearly is not equal to the GDC but in the fault domain. The first lines read as follows:

```
1 FAILURE: SUT does not implement output trace "a1,a3,a1,a3,a1"  
   for input trace "e1,e1,e1,e1,e1"  
2       Closest match diverges at step 3: ""  
3       (Expected: "a1,a3,a1")  
4 FAILURE: SUT implements unspecified output trace "a1,a3" for  
   input trace "e1,e1,e1,e1,e1"  
5       Closest match diverges at step 3: "a1,a3,a1"
```

The first failure here shows the missing transition: When executing the input sequence `e1,e1,e1`, we expect the output sequence `a1,a3,a1`. However, due to the missing transition in the mutant, the mutants execution stops after the second input. The *closest match* mentioned by the checker is the sequence with the sequence produced by the SUT with the longest prefix common with the expected output sequence. Beginning at step 3, i.e. after the second `e1` input, the SUT produces an empty trace, which is indicated by the second line. The third line shows the output sequence that was expected at that position.

The second failure shows the same fault but from the other perspective: The SUT produces an unexpected output sequence `a1,a3` and the closest match is the output sequence in the set of expected output sequences that shares the longest prefix with the produced sequence.

Further down the list of failures, we see the following lines:

```
1 FAILURE: SUT does not implement output trace "a1,a4,a3,null,a1,
    a3" for input trace "e1,e4,e3,e3,e1,e1"
2         Closest match diverges at step 4: "a3,a1,a3"
3         (Expected: "null,a1,a3")
4 FAILURE: SUT implements unspecified output trace "a1,a4,a3,a3,a1
    ,a3" for input trace "e1,e4,e3,e3,e1,e1"
5         Closest match diverges at step 4: "null,a1,a3"
```

These reflect the mutated output: The expected output at that position in the execution sequence would have been the output `null`, whereas the SUT produces the output `a3`. □

## 5.4 Using the Test Harness

A **test harness** is a program which exercises a given test suite on a software under test (SUT). The `libfsmttest` class library comes with a test harness which allows to execute test suites generated by means of one of the FSM-based methods described above against a C<sup>++</sup> library consisting of one or more operations to be tested. The re-usable test harness requires *input refinement* (each input alphabet value of the test case needs to be mapped to a concrete SUT operation call with input parameter values and presets of attributes) and *output abstraction* (the effect of each operation call on return value, reference parameters and attributes needs to be abstracted to the corresponding value of the reference FSM's output alphabet).

To support this, the test harness operates with a *SUT wrapper*. This is a C<sup>++</sup>-source frame to be completed for each test campaign, offering a function with fixed signature `std::string sut(const std::string& x)` to the test harness for calling the SUT. For each input to be exercised on the SUT in a test step, the test harness calls `sut(x)`, where `x` is the input alphabet value as string. The wrapper maps `x` to concrete input data (parameters and attributes) of the SUT and calls the associated SUT operation. The SUT response is abstracted by the wrapper to an output alphabet value which is returned as string from call `sut(x)` to the harness. The harness checks SUT reactions by simulating the test suite's reference FSM in back-to-back fashion and comparing outputs. In Fig. 5.5, the interplay between harness, wrapper and SUT is depicted.

The harness is contained in `libfsmttest` as file

```
libfsmttest/src/harness/harness.cpp
```

It needs to be compiled and linked with the SUT, but there should be no need in general to make any adaptations in this file.

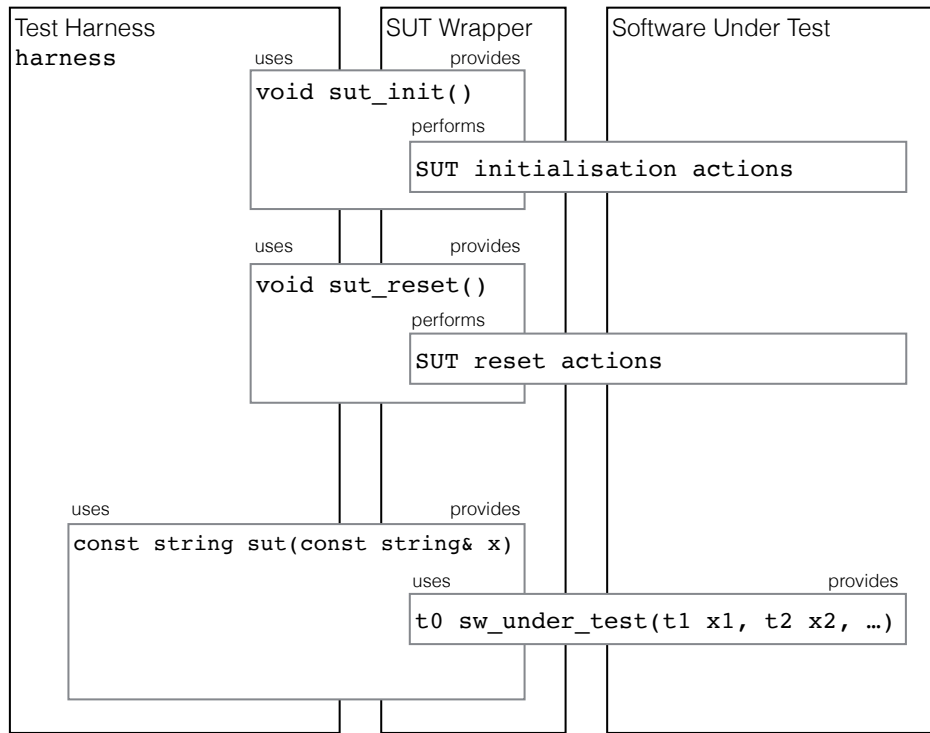


Figure 5.5: Test harness, SUT wrapper, and software under test.

The wrapper source frame is provided by file

`libfsmtest/src/harness/sut_wrapper.cpp`

This file needs to be edited; the source frame is shown in Listing 5.1. As can be seen in the source frame, the following code needs to be included.

1. Include-directives to SUT-specific header files, so that the software under test can be invoked from the wrapper, and the attributes to be preset can be accessed (line 4 in Listing 5.1).
2. Data structures (typically maps) of functions for input refinement and output abstraction need to be inserted (line 10 in Listing 5.1).

3. The SUT has to be initialised in the body of function `void sut_init()`. If the SUT software does not consist of a static class, it is usually required to instantiate SUT objects here and make pointers to these objects available in the global wrapper data.
4. The body of function `void sut_reset()` needs to be filled in. The code to be inserted here should re-initialise the SUT, so that a new test case can be applied to the SUT residing again in its initial state.
5. The body of the function `const string sut(const string& input)` needs to be provided. Here, the input data refinement is performed and the associated SUT operation is called. The returned data and changed attributes are abstracted to the return string `fsmOutputEvent` and returned to the calling harness.



---

Listing 5.1: SUT wrapper source frame.

```
1 #include <string>
2
3 // Include header files of library to be tested
4 // #include "..."
5
6 /** Helper data structures and functions for
7  * SUT test wrapper */
8 // ...
9 using namespace std;
10 void sut_init() {
11     // initialise wrapper data structures
12     // for mapping FSM inputs to SUT inputs
13     // and vice versa
14     // ...
15     // Initialise SUT, if required, by calling
16     // initialisation functions or initialising
17     // global SUT variables
18     // ...
19 }
20
21 void sut_reset() {
22     // Insert code suitable for resetting SUT into
23     // its initial state
24     // ...
25 }
26
27 const string sut(const string& input) {
28     string fsmOutputEvent;
29     // Transform FSM input event passed as string
30     // 'input' to SUT input variable settings and
31     // global variables settings
32     // ...
33     // Call the SUT function addressed by the FSM input event
34     // with the input parameter values defined before
35     // ...
36     // Convert the return value, the (in-)out-parameter values,
37     // and the global SUT variables to the FSM output event
38     // represented as string fsmOutputEvent
39     // ...
40     // return output event in string representation
41     return fsmOutputEvent;
42 }
```

**Example 5.** In directory

`libfsmtest/src/harness/example`

an example test suite has been provided for testing the garage door controller (GDC) code which is contained there in files `gdclib.cpp`, `.hpp`. The reference DFSM is the result of a generator call (see Section 5.2) and resides in files `GDC.fsm`, `.in`, `.out`, `.state`. The test cases are contained in `SUITE-GDC.txt`.

The SUT wrapper for this test application is shown in Listing 5.2. In lines 10—15, this wrapper applies a particularly simple, but frequently applicable variant of input refinement and output abstraction. To analyse this, recall from the GDC software interface specified in Section 5.1.2 that inputs to the GDC are specified as enumeration values `e1`, `...`, `e4` and outputs by enumeration values `nop`, `a1`, `...`, `a4`. The test harness uses inputs encoded by the presentation layer of the reference DFSM (see file `GDC.in`). This encoding uses strings `"e1"`, `...`, `"e4"`. As a consequence, input refinement simply requires a mapping of input alphabet strings to associated enumeration values. This is realised in lines 10—12 as `map fsmIn2gdcIn`. Conversely, the GDC software return values `nop`, `a1`, `...`, `a4` need to be mapped to output alphabet values of the reference DFSM (file `GDC.out`) which has the string values `"null"`, `"a1"`, `...`, `"a4"`. This is realised by `map gdcOut2fsmOut` in lines 13—15.

With these mappings at hand, the implementation of the `sut()` wrapper function shown in lines 23—34 is straightforward: the input alphabet value `input` is transformed by means of `map fsmIn2gdcIn` into an enumeration value which is used as input argument of the SUT function `gdc()`. The return value `y` of this SUT call is transformed by means of `map gdcOut2fsmOut` into a string of the DFSM output alphabet and used as return value of the wrapper function `sut()`.

---

Listing 5.2: SUT wrapper for the garage door controller example.

```
1 #include <string>
2 #include <map>
3 using namespace std;
4 // Include header files of library to be tested
5 #include "gdclib.hpp"
6 /**
7  *   Helper data structures and functions for
8  *   SUT test wrapper
9  */
10 map<string, gdc_inputs_t> fsmIn2gdcIn = {
11     {"e1", e1}, {"e2", e2}, {"e3", e3}, {"e4", e4}
12 };
13 map<gdc_outputs_t, string> gdcOut2fsmOut = {
14     {nop, "null"}, {a1, "a1"}, {a2, "a2"}, {a3, "a3"}, {a4, "a4"}
15 };
16 using namespace std;
17 void sut_init() {
18     gdc_reset();
19 }
20 void sut_reset() {
21     gdc_reset();
22 }
23 const string sut(const string& input) {
24     string fsmOutputEvent;
25     map<string, gdc_inputs_t>::iterator
26         inputIte = fsmIn2gdcIn.find(input);
27     if ( inputIte == fsmIn2gdcIn.end() ) return fsmOutputEvent;
28     gdc_outputs_t y = gdc( inputIte->second );
29     map<gdc_outputs_t, string>::iterator
30         outputIte = gdcOut2fsmOut.find(y);
31     if ( outputIte == gdcOut2fsmOut.end() ) return
32         fsmOutputEvent;
33     fsmOutputEvent = outputIte->second;
34     return fsmOutputEvent;
35 }
```

---

Using the harness/example sub-directory as working directory, the harness is now compiled and linked together with wrapper and SUT using command

```

1 c++ -std=c++17 -o harness \
2   -I./ -I../libfsmtest \
3   *.cpp ../harness.cpp
4   ../libfsmtest/libfsmtest.a -lc++

```

Command `c++` is a link to the  $C^{++}$  compiler. The `-std` option indicates that  $C^{++}$  2017 syntax is admissible (some compilers use an older  $C^{++}$  standard if this option is missing; this might lead to compile errors). Options `-I...` indicate where to look for  $C^{++}$  header files. In line 3 all `cpp`-files in the local directory (wrapper and SUT code) are referenced, as well as the harness source code residing in the directory above. In line 4, the `libfsmtest` library and the  $C^{++}$  standard library are made available to the other object files. As a result of the compilation and linking process, the executable `harness` is created which can be invoked with different test suites and reference DFSMs to be executed against the GDC software under test. For the `SUITE-GDC.txt` test cases, the execution command is

```

1 ./harness SUITE-GDC GDC

```

which leads to result

```

1 PASS: e1/a1, e1/a3, e1/a1
2 PASS: e1/a1, e2/a3, e1/a2, e1/a3
3 PASS: e1/a1, e2/a3, e1/a2, e2/null
4 PASS: e1/a1, e2/a3, e2/null, e1/a2
5 PASS: e1/a1, e2/a3, e3/null, e1/a2
6 PASS: e1/a1, e2/a3, e4/null, e1/a2
7 PASS: e1/a1, e3/null, e1/a3
8 PASS: e1/a1, e3/null, e2/a3
9 PASS: e1/a1, e4/a4, e1/a3, e1/a2
10 PASS: e1/a1, e4/a4, e2/null, e1/a3
11 PASS: e1/a1, e4/a4, e2/null, e2/null
12 PASS: e1/a1, e4/a4, e3/a3, e1/a1
13 PASS: e1/a1, e4/a4, e4/null, e1/a3
14 PASS: e1/a1, e4/a4, e4/null, e2/null
15 PASS: e2/null, e1/a1
16 PASS: e3/null, e1/a1
17 PASS: e4/null, e1/a1

```

written to the console.

Readers are invited to experiment with different SUT implementations, fault injections into the `gdclib.cpp` 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 behaviour of the software under test. If this is suspected, the parameter `-a <additional states>` has to be used for test generation with a suitable estimate (see Section 5.2). Otherwise it is not guaranteed that the test suite will uncover every violation of language equivalence between implementation and reference model.  $\square$

Note that the current version of the test harness only checks for language equivalence with deterministic reference FSMs and deterministic implementations. Nondeterminism and reduction testing, as well as support of other conformance relations (quasi reduction and strong reduction) will be included in future versions of the harness.

## Chapter 6

# Visitors for Saving FSMs to Disk

The `libfsmtest` class library offers three ways to store FSM instances to disk:

- Raw file format described in Section 3.1.1.
- CSV format described in Section 3.1.2.
- GraphViz format (also called “dot format”) to be visualised by GraphViz<sup>1</sup>.

To store an FSM instance in raw format, class `ToFsmFileVisitor` with files

```
src/libfsmtest/visitors/ToFsmFileVisitor.hpp, .cpp
```

is used. To store an FSM instance in CSV format, class `ToCsvFileVisitor` with files

```
src/libfsmtest/visitors/ToCsvFileVisitor.hpp, .cpp
```

is used. To store an FSM instance in GraphViz format, class `ToDotFileVisitor` with files

---

<sup>1</sup><https://graphviz.org>

src/libfsmtest/visitors/ToDotFileVisitor.hpp, .cpp

is used. All visitors get the filename as input parameter of their constructor. Listing 6.1 shows how to apply these visitors.

---

Listing 6.1: Write an FSM to disk in three different formats format.

```
1 #include "..."  
2  
3 using namespace std;  
4 using namespace libfsmtest;  
5  
6 int main(int argc, char* argv[]) {  
7  
8     // read and transform FSMs  
9     // ...  
10  
11     // Write FSM pointed to by unique pointer 'fsm'  
12     // in raw format to disk.  
13     // We use the FSM name as base name. The visitor creates  
14     // 4 files (extensions .fsm, .in, .out, .state) with this  
15     // basename.  
16     ToFsmFileVisitor visitor(fsm->getName());  
17     fsm->accept(visitor);  
18     visitor.writeToFile();  
19  
20     // Write FSM pointed to by unique pointer 'fsm' in  
21     // CSV format to disk. Again, we use the FSM name  
22     // as file basename  
23     ToCsvFileVisitor csv(fsm->getName());  
24     fsm->accept(csv);  
25     csv.writeToFile();  
26  
27     // Write FSM pointed to by unique pointer 'fsm' in  
28     // GraphViz format to disk  
29     ToDotFileVisitor dot(fsm->getName()+".dot");  
30     fsm->accept(dot);  
31     dot.writeToFile();  
32  
33 }
```

---

# Chapter 7

## Bibliography

- [1] Tsun S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–186, March 1978.
- [2] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999. ISBN 0262032708.
- [3] 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. ISBN 3-540-29189-X. doi: 10.1007/11562436\16. URL <https://doi.org/10.1007/11562436>.
- [4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., USA, 1995. ISBN 0201633612.
- [5] Arthur Gill. *Introduction to the theory of finite-state machines*. McGraw-Hill, New York, 1962.
- [6] Robert M. Hierons. Testing from a nondeterministic finite state machine using adaptive state counting. *IEEE Trans. Computers*, 53(10):1330–1342, 2004. doi: 10.1109/TC.2004.85. URL <http://doi.ieeecomputersociety.org/10.1109/TC.2004.85>.
- [7] Robert M. Hierons. FSM quasi-equivalence testing via reduction and observing absences. *Sci. Comput. Program.*, 177:1–18, 2019. doi: 10.1016/j.scico.2019.03.004.
- [8] Wen-ling Huang and Jan Peleska. Complete requirements-based testing with finite state machines. *CoRR*, abs/2105.11786, 2021. URL <https://arxiv.org/abs/2105.11786>.



- [9] Wen-ling Huang, Sadik Özoguz, and Jan Peleska. Safety-complete test suites. *Software Quality Journal*, 27(2):589–613, 2019. doi: 10.1007/s11219-018-9421-y. URL <https://doi.org/10.1007/s11219-018-9421-y>.
- [10] Paul C. Jorgensen. *The Craft of Model-Based Testing*. CRC Press, Boca Raton, 2017.
- [11] Gang Luo, Gregor von Bochmann, and Alexandre Petrenko. Test selection based on communicating nondeterministic finite-state machines using a generalized wp-method. *IEEE Trans. Software Eng.*, 20(2):149–162, 1994. doi: 10.1109/32.265636.
- [12] Patricia D. L. Machado, Daniel A. Silva, and Alexandre C. Mota. Towards Property Oriented Testing. *Electronic Notes in Theoretical Computer Science*, 184(Supplement C):3–19, July 2007. ISSN 1571-0661. doi: 10.1016/j.entcs.2007.06.001. URL <http://www.sciencedirect.com/science/article/pii/S157106610700432X>.
- [13] S. Naito and M. Tsunoyama. Fault detection for sequential machines by transition tours. In *Proc. IEEE Fault Tolerant Comput. Conf.*, pages 162–178, 1981.
- [14] Jan Peleska and Wen-ling Huang. *Test Automation - Foundations and Applications of Model-based Testing*. University of Bremen, January 2021. URL <http://www.informatik.uni-bremen.de/agbs/jp/papers/test-automation-huang-peleska.pdf>. Lecture notes.
- [15] Robert Sachtleben and Jan Peleska. Effective grey-box testing with partial FSM models. *CoRR*, abs/2106.14284, 2021. URL <https://arxiv.org/abs/2106.14284>.
- [16] Michal Soucha and Kirill Bogdanov. Spyh-method: An improvement in testing of finite-state machines. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops, Västerås, Sweden, April 9-13, 2018*, pages 194–203. IEEE Computer Society, 2018. doi: 10.1109/ICSTW.2018.00050.
- [17] Jaco van de Pol and Jeroen Meijer. Synchronous or Alternating? In Tiziana Margaria, Susanne Graf, and Kim G. Larsen, editors, *Models, Mindsets, Meta: The What, the How, and the Why Not? Essays Dedicated to Bernhard Steffen on the Occasion of His 60th Birthday*, Lecture Notes in Computer Science, pages 417–430. Springer International Publishing, Cham, 2019. ISBN 978-3-030-22348-9. doi: 10.1007/978-3-030-22348-9\_24. URL [https://doi.org/10.1007/978-3-030-22348-9\\_24](https://doi.org/10.1007/978-3-030-22348-9_24).
- [18] M. P. Vasilevskii. Failure diagnosis of automata. *Kibernetika (Transl.)*, 4:98–108, July-August 1973.