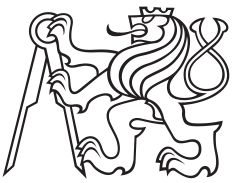
Master's Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Checking Experiment Design Methods

Michal Soucha
Open Informatics – Artificial Intelligence

May 2015
Supervisor: Ing. Radek Mařík, CSc.

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Computer Science and Engineering

DIPLOMA THESIS ASSIGNMENT

Student: **Bc. Michal Soucha**

Study programme: Open Informatics
Specialisation: Artificial Intelligence

Title of Diploma Thesis: **Checking experiment design methods**

Guidelines:

1) Make a summary of methods of checking sequence design.
2) Describe applicability of the methods on certain classes of finite-state machines.
3) Compare designed checking sequences in their length and their fault coverage.
4) Implement construction methods to the library of finite-state machines functions.
5) Propose and implement a construction algorithm based on the research.
6) Compare the proposed algorithm with previously published ones. Discuss its properties.

Bibliography/Sources:

[1] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. Introduction to Automata Theory, Languages, and Computation. Prentice Hall, 2006.

[2] Michael Kearns, and Umesh Virkumar Vazirani. An introduction to computational learning theory. The MIT Press, 1994.

[3] Arthur Gill. Introduction to the Theory of Finite-state Machines. McGraw-Hill Book Company, 1962.

[4] Tsun S. Chow. Testing software design modeled by finite-state machines. Software Engineering, IEEE Transactions on. 1978, (3), 178-187.

[5] David Lee, and Mihalis Yannakakis. Principles and methods of testing finite state machines-a survey. Proceedings of the IEEE. 1996, 84 (8), 1090-1123.

Diploma Thesis Supervisor: Ing. Radek Mařík, CSc.

Valid until the end of the summer semester of academic year 2015/2016

L.S.

doc. Ing. Filip Železný, Ph.D.
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, May 5, 2015

Acknowledgement / Declaration

I would like to thank my supervisor who introduced me to a very interesting field of automata theory. My thanks also to my family for their support in my studies and research.

I declare that I have developed the presented work independently and that I have listed all information sources used in accordance with the Methodical guidelines on maintaining ethical principles during the preparation of higher education theses.

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Prague, 10 May 2015

.....

Abstrakt / Abstract

Tento dokument obsahuje přehled testovacích metod konečně-stavových automatů. Testovací metoda kontroluje zda se nějaký automat shoduje s daným automatem. Nová metoda je představena a následně experimentálně prokázána, že je korektní testovací metodou. Dokument dále obsahuje klasifikaci konečně-stavových automatů, algoritmy konstruující separující sekvence a nový přístup kontrolování, zda je metoda korektní testovací metodou. Separující sekvence rozlišuje dva stavy automatu, se kterými souvisí.

Klíčová slova: Konečně-stavový automat, Testovací metoda, Soubor testů, Úplné pokrytí chyb, Kontrola pokrytí chyb, Separující sekvence, Stav charakterizující množina, Charakterizující množina

Překlad titulu: Konstrukce experimentu potvrzující ekvivalenci konečně-stavových automatů

This document is an overview of testing methods of finite-state machines. Testing methods verifies whether a machine coincides with given one. A new method is proposed and it is demonstrated to be correct testing method of finite-state machines experimentally. The document further comprises classification of finite-state machines, algorithms for creating separating sequences and a new approach to checking testing methods to be correct. A separating sequence relates to a pair of states. Such two states are distinguished by the separating sequence.

Keywords: Finite-state machine, Testing method, Checking sequence, Test suite, Complete fault coverage, Fault coverage checking, Separating sequence, State characterizing set, Characterizing set

Contents

Tables / Figures

Algorithms

Chapter 1
Introduction

Checking a product against its specification is common practice in many fields. We research *active learning* of *finite-state machines* [An87]. When we will be able to compare two machines, an automatic learning system can be created. Therefore, we focused on *testing methods* of finite-state machines that verify an implementation against the specification. Such a verification is called a *checking experiment* [Mo56].

There are many testing methods for different types of machines. Hence, we state types and classes of finite-state machines in Chapter 3 at first. Then testing methods we are aware of are described in Chapter 6. The following Chapter 7 proposes similarity relations between methods. Moreover, we capture the methods in terms of time of their proposal. Knowledge of the testing methods enables us to introduce a new method that is a generalization of two state-of-the-art methods. The new method called the *M-method* is proposed in Chapter 8.

Testing methods creates a set of input sequences based on given specification which is a finite-state machine. Such a set is called *test suite* and has to possess some properties to be able to verify an implementation against the specification. These properties and conditions are proposed in Chapter 4. When one cannot prove a method to be a correct testing method, a checker of produced test suite is employed. Checker finds out whether a given test suite reveals all possible faults in the implementation. We propose a new checker called the *Fault Coverage Checker* in Chapter 9. It is used to check correctness of the M-method experimentally due to the fact that the method has not been formally proven yet.

A comparison of performance of testing methods is captured in experiments in Chapter 11. Testing methods are compared in the length of created test suite. Performance of most methods depends on their implementation because methods usually contain an incompletely specified part of their design. We propose our implementation of methods in Chapter 10 so our results can be referenced for a comparison. Chapter 10 also comprises our algorithms for creating separating sequences and characterizing sets of sequences that are needed in testing methods. New parallel approaches for design shortest separating sequences are discussed as well.

Chapter 2 and Chapter 3 establish terms used through the entire thesis. A necessary terminology involves many fields of abstract mathematical domains. We try to unify denotation at the end of Chapter 3 because there is a wide variety of denotation in the literature. In addition, Chapter 3 states different types and classes of finite-state machines. A comparison of these classes is also included in Chapter 3. Chapter 5 then references work related to topics that this thesis deals with.

This Diploma thesis focuses on summarizing testing methods and proposing new approaches. Therefore, proofs of correctness of proposed algorithms are omitted and some notions are not illustrated in detail because they can be found in the referenced literature or they would require an extra space for a description so compactness of the thesis would be disturbed.

Chapter 2
Definition

Automata theory is a well-known part of computer science. However, its terminology is not unique, there are a lot of different definitions. So, at first, it is necessary to state definitions that will be used through the text. We propose definitions that can be found in almost all books dealing with this particular field. Note that some definitions are formulated in a shortened version because a purpose of this thesis is not to provide deeply precise definitions of all used terms.

2.1 Set theory

We assume that the reader is familiar with a *set* as a collection of elements. Particularly, we will use the following terms related to a finite set: cardinality, union, intersection, set difference, power set and partition.

Definition 2.1. A *cardinality* of a set A is the number of elements of the set A. It is denoted $|A|$.

Definition 2.2. Let A and B be sets. Then

$A \cup B$ is *union* of sets A and B; $A \cup B = \{x \mid x \in A \vee x \in B\}$,
$A \cap B$ is *intersection* of sets A and B; $A \cup B = \{x \mid x \in A \wedge x \in B\}$,
$A \setminus B$ is *set difference* of A and B; $A \setminus B = \{x \in A \mid x \notin B\}$.

Definition 2.3. A *power set* of a set A, denoted $\mathcal{P}(A)$, is the set whose members are all possible subsets of A, i.e. $\mathcal{P}(A) = \{B \mid B \subseteq A\}$.

Definition 2.4. A *partition* Π of a set A is a set of nonempty subsets of A such that every element $a \in A$ is in exactly one of these subsets, i.e., A is a disjoint union of subsets from Π.

We call a subset of a partition also a *group* or a *block*. When we deal with an *equivalence relation*, a subset of a partition is called an *equivalence class*.

2.2 Graph theory

A search state space is usually represented as a tree which is a special form of graph.

Definition 2.5. A *directed graph* G is a pair (V, E) where V is a set of nodes and E is a set of directed edges, i.e. a set of ordered pair of nodes.

For notation, let $(u, v) \in E$ be an edge of a directed graph $G = (V, E)$, then $u \in V$ is said to be a *predecessor* of v and $v \in V$ is said to be a *successor* of u.

Definition 2.6. Let $G = (V, E)$ be a directed graph. A node $v \in V$ is *reachable* from a node $u \in V$ if and only if $u = v$ or there exists a node $w \in V$ so that there is an edge from u to w, i.e. $(u, w) \in E$, and v is reachable from w.

Definition 2.7. Let $G = (V, E)$ be a directed graph. A *path* is a sequence of nodes so that each two successive nodes are connected by an edge $e \in E$ and all nodes in the path are distinct from one another.

Definition 2.8. Let $G = (V, E)$ be a directed graph. G is *strongly connected* if and only if each node $v \in V$ is reachable from any node $u \in V$.

2.2.1 Trees

There are many different types of trees but we use a successor tree only [De94].

Definition 2.9. A *rooted directed tree*, or a *successor tree*, is a directed graph $G = (V, E)$ with a special node $r \in V$, called *root*, with the following properties:

1) every node $u \in V$ is reachable from r,
2) the tree has exactly $n - 1$ edges, where n is number of nodes, i.e. $n = |V|$.

Nodes in a tree have a special notation. If (u, v) is an edge then u is said to be a *parent* of v and v is a *child*, or a successor, of u. If node u has no successor then u is said to be a *leaf*. Otherwise, u is said to be an *internal node*, i.e. $\exists v \in V : (u, v) \in E$.

Note that we assume only directed trees so that if we talk about a rooted tree it means a rooted directed tree. A *testing tree* is a successor tree with specialized nodes.

2.3 Alphabet and string

Definition 2.10. An *alphabet* X is a nonempty finite set of symbols $x_1, \ldots, x_p$, $p = |X|$. A *string*, or a *word*, over X is any finite sequence of symbols from X.

Then we can define the following:

ε	is the empty string,
X_ε	is the alphabet X extended with ε, i.e. $X_\varepsilon = X \cup \{\varepsilon\}$,
X^k	is the set of all strings over the alphabet X of length k; $X^0 = \{\varepsilon\}, X^1 = X$,
X^*	is the set of all strings over the alphabet X, i.e. $X^* = \bigcup_{k \in \mathbb{N}_0} X^k$.
$u \cdot v$	means concatenation of strings u and v. It can be also written as uv.
$\|u\|$	means the length of a string (word) u; $\|\varepsilon\| = 0$.

Note that set of all strings X^* always contains ε and $\forall u \in X^* : \varepsilon \cdot u = u = u \cdot \varepsilon$. Thus X^* is always nonempty and it is also countable because X is countable.

Definition 2.11. Let u, v, w be strings. v is a *prefix* of u if $u = v \cdot w$. Moreover, v is a *proper prefix* of u if w is nonempty, i.e. $w \neq \varepsilon$.

Definition 2.12. Let $\mathrm{pref}(A)$ denote the set of all prefixes of each string from the set A. A set of strings A is *prefix-closed* if for each sequence $u \in A$, it holds that A contains all prefixes of u, i.e. $A = \mathrm{pref}(A)$.

Definition 2.13. Let u and v be strings. u is *extended* by v and v is an *extension* of u when a word $u \cdot v$ is created. In similar way, let A and B be sets of strings. A is *extended* by B and B is an *extension* of A when a set of strings C contained each string from A extended by each string from B is formed. Formally, $C = A \otimes B = \{u \cdot v \mid u \in A \wedge v \in B\}$.

Chapter 3
Finite-state machine

An automaton with a finite number of states, or a finite-state machine (FSM), is a specific mathematical model of computation. The behavior of a system can be modeled as transitions between states. It is conceived as an abstract machine that can be in one of a finite number of states. It has many slightly different variants, such as Moore and Mealy machine, deterministic and nondeterministic finite automaton or nondeterministic finite-state machine [Ho06]. Therefore we propose their generalization and a discussion on specific types.

Definition 3.1. A finite-state machine is a quintuple (S, X, Y, s_0, h), where

- S is a finite nonempty set of states,
- X is an input alphabet (a finite nonempty set of symbols),
- Y is an output alphabet (a finite nonempty set of symbols),
- s_0 is an initial state, $s_0 \in S$,
- h is a behavior function: $h : S \times X_\varepsilon \to \mathcal{P}(S \times Y_\varepsilon)$,

Every FSM can be represented as a directed graph or a table. The graph form is called a *state diagram*. Nodes of the graph are states and edges represent transitions. So an edge is labeled with the input symbol which causes a transition between two states connected by the edge. The output symbol is attributed to the state and the edge according to the behavior function h. The table form lists transitions and corresponding outputs in a *behavior table*. A row corresponds to one state; the table has as many rows as the machine has states. Each column is linked to a symbol from the input alphabet or to the empty string. There is a set of pairs of next state and output in each cell. A cell could be empty since range of the behavior function is power set of state-output pairs.

Definition 3.1 is general enough to be able to describe all aforementioned types. On the other hand, it is not used due to its generalization. Before we will show that the well-known types are particular cases of the one proposed in Definition 3.1, common properties are stated.

Definition 3.2. A transition $(s, x) \in S \times X_\varepsilon$ is *defined* if and only if $|h(s, x)| \geq 1$. An input sequence $u = x_1 \cdot \ldots \cdot x_k \in X_\varepsilon^*$ is defined for state $s \in S$ if and only if there is a sequence of states $(s_i)_{i=1}^{k+1}$ and an output sequence $y_1 \cdot \ldots \cdot y_{k+1} \in Y_\varepsilon^*$ such that $s = s_1$ and for all $1 \leq i \leq k : (s_{i+1}, y_{i+1}) \in h(s_i, x_i)$. The set of all defined input sequences for state s is denoted $\Omega(s)$. The set of all defined input sequences for the initial state s_0 of FSM M is Ω_M, i.e. $\Omega_M = \Omega(s_0)$.

The input of behavior function h is a pair and thus there should be parentheses twice. We omit the second parentheses for the sake of simplicity. That is, we use $h(s, x)$ instead of correct $h((s, x))$. The same simplification will be employed in the rest of thesis and for other defined functions as well. Similarly, if a set is a singleton, the set's curly brackets are omitted as well. For example, we write $h(s, x) = (s', y)$ instead of correct $h(s, x) = \{(s', y)\}$. Notice that both input sequence and output sequence can contain the empty string ε according to Definition 3.2, e.g. $u = x_1 \cdot \varepsilon \cdot x_3 \in \Omega(s)$.

Definition 3.3. A FSM M is *completely specified*, or simply *complete*, if each transition is defined, i.e. $\forall s \in S \forall x \in X : |h(s,x)| \geq 1$. Otherwise, the FSM M is *partially specified*, or simply *partial*, i.e. when $\exists s \in S \exists x \in X : h(s,x) = \emptyset$.

For easier definition of next notions, we extend behavior function to sequences (strings) at first.

Definition 3.4. Extended behavior function $h^* : S \times X_\varepsilon^* \to \mathcal{P}(S \times Y_\varepsilon^*)$ is defined inductively by

(0) $h^*(s,x) = h(s,x) \quad \forall s \in S \forall x \in X_\varepsilon$

(1) $h^*(s,ux) = \bigcup_{(s',z) \in h^*(s,u)} \{(s'', zy) \mid (s'', y) \in h(s',x)\} \quad \forall s \in S \forall u \in \Omega(s) \forall x \in X_\varepsilon$

Note that if $ux \notin \Omega(s)$ then $h^*(s,ux) = \emptyset$, because $h(s',x)$ would be empty as well and thus (s'', y) and (s'', zy) would not be defined.

Each finite-state machine can be represented as a graph, therefore, the notion of connectedness is easily applied to FSM.

Definition 3.5. A FSM M is *initially connected* if and only if each state is reachable from the initial state, i.e. for each state s there exists a defined input sequence from the initial state s_0 to s; formally, $\forall s \in S \exists u \in \Omega_M \exists z \in Y_\varepsilon^* : (s,z) \in h^*(s_0,u)$.

Definition 3.6. A FSM M is *strongly connected* if and only if each state is reachable from each other. That is, $\forall s, s' \in S \exists u \in \Omega(s) \exists z \in Y_\varepsilon^* : (s',z) \in h^*(s,u)$.

Definition 3.7. Two states $s_i, s_j \in S$ are *distinguishable*, denoted $s_i \nsim s_j$, if there is an input sequence $u \in \Omega(s_i) \cap \Omega(s_j)$ such that the output sequences $z_i, z_j \in Y_\varepsilon^*$ produced as responses to applying u in both states are different, i.e.

$$\exists z_i, z_j \in Y_\varepsilon^* \exists s_i', s_j' \in S : (s_i', z_i) \in h^*(s_i,u) \wedge (s_j', z_j) \in h^*(s_j,u) \wedge z_i \neq z_j$$

The sequence u is called a *separating sequence* of states s_i and s_j.

Definition 3.8. A FSM M is *minimal*, or *reduced*, if and only if it is initially connected and each two states are distinguishable, i.e. $\forall s_i \neq s_j \in S : s_i \nsim s_j$.

Thus, a reduced FSM is a machine with least number of states in a group of equivalent machines. Machines are equivalent if they produce equal output on the same input and initial states of machines are equivalent. Minimization algorithms [Ho06] solve how to convert a FSM into its minimal form.

Definition 3.9. A FSM M is *resettable* if and only if M can be taken to its initial state s_0 from each state. Formally, there is an input $r \notin X$, called *reset*, that brings M into s_0 with the output $null \notin Y$, i.e. $\forall s \in S : h(s,r) = (s_0, null)$.

The output *null* cannot be in the output alphabet because we want to uniquely identify that reset was applied.

The stated notions can be defined for a specific type of FSM in easier way. Therefore, we propose now definitions of specific types and their relations to Definition 3.1.

3.1 Determinism

Definition 3.1 of FSM allows several next states for a defined transition. Nevertheless, it is more natural to have the uniquely defined next state for each defined transition, i.e. $\forall s \in S \forall x \in X : |h(s,x)| \leq 1$. Such a concept is called deterministic machine (DFSM).

It also includes that the machine cannot change its state when no input is applied, i.e. $\forall s \in S \, \exists y \in Y_\varepsilon : h(s, \varepsilon) = (s, y)$.

A machine that contains a transition leading into a couple of states is *nondeterministic*. Definition 3.1 thus defines nondeterministic finite-state machine (NFSM).

Definition 3.10. A *deterministic finite-state machine* is a septuple $(S, X, Y, s_0, D, \delta, \lambda)$, where S, X, Y, s_0 are a set of states, an input and an output alphabets and the initial state, respectively, and

D is a domain of defined transitions; $D \subseteq S \times X$,
δ is a state-transition function: $\delta : D \to S$,
λ is an output function: $\lambda : D \cup \{S \times \{\varepsilon\}\} \to Y_\varepsilon$.

A DFSM is a FSM $M = (S, X, Y, s_0, h)$ such that $D = \{(s, x) \mid |h(s, x)| = 1\}$ and

(i) $h(s, x) = (\delta(s, x), \lambda(s, x))$ $\quad \forall (s, x) \in D$
(ii) $h(s, x) = \emptyset$ $\quad \forall (s, x) \in (S \times X) \setminus D$
(iii) $h(s, \varepsilon) = (s, \lambda(s, \varepsilon))$ $\quad \forall s \in S$

A *state transition table* and an *output table* are used for deterministic finite-state machine instead of a behavior table. The content of cells is the only difference. State transition table has a state or nothing (not defined) in cell according to the transition function δ. The output function λ is listed in output table. Output table contains an output symbol, the empty string or nothing (not defined) in cell. A *state diagram* is other type of representation of DFSM, see Definition 3.1.

The deterministic FSM of Definition 3.10 is still too general for a wide application. There are two more used types of FSM. Both are specified DFSMs. They have assigned output symbols either to states only or to transition only.

3.2 Mealy and Moore model

Two types of DFSM are distinguished in automata theory: Mealy and Moore machines. A Mealy model can represent more complex systems than a Moore model with the same number of states. On the contrary, a representation of the Moore model needs less storing space, e.g. in computer. The Mealy model is used more in practice due to its generality. The Moore machine has tied outputs with states in contrast to the Mealy machine with outputs only on transitions.

Definition 3.11. A *Mealy machine* is a DFSM with a specified output function λ:

$\lambda(s, \varepsilon) = \varepsilon \quad \forall s \in S \qquad h(s, \varepsilon) = (s, \varepsilon)$
$\lambda(s, x) \neq \varepsilon \quad \forall s \in S \, \forall x \in X \qquad \exists y \in Y : h(s, x) = (\delta(s, x), y)$

The output function can be redefined as $\lambda : D \to Y$. On the right side of the specified output function λ in Definition 3.11 there is captured the correspondence with the behavior function h of FSM. Joined transition and output tables, or simply behavior table, of an example of Mealy machine is shown in Figure 3.1. One can see that the transitions on the empty string are not depicted because they produce the empty string and return back to their starting state; they are loops.

$S = \{A, B, C\}, X = \{a, b\}, Y = \{1, 2, 3\}, s_0 = A$

	a	b
A	B / 1	B / 2
B	A / 1	C / 3
C	B / 3	C / 1

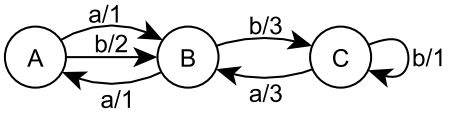


Figure 3.1. An example of the same Mealy machine in two representations

Definition 3.12. A *Moore machine* is a DFSM with a specified output function λ:

$$\lambda(s,\varepsilon) \neq \varepsilon \qquad \forall s \in S \qquad \exists y \in Y : h(s,\varepsilon) = (s,y)$$
$$\lambda(s,x) = \varepsilon \qquad \forall s \in S\, \forall x \in X \qquad h(s,x) = (\delta(s,x),\varepsilon)$$

The output function is needed only for labeling states therefore it can be stated as $\lambda : S \to Y$. Again, on the right side in Definition 3.12 there is the specification of the behavior function h describing the restriction of FSM to Moore machine.

If the first output symbol of a Moore machine, $\lambda(s_0)$, is disregarded then the machine can be readily converted to an output-equivalent Mealy machine. Each edge of Mealy model is labeled by the output symbol of the destination state of Moore model. Algorithms in this text are mainly customized for the Mealy model so the above transformation and some preliminary settings are needed. Usually we first compare the outputs of states and then handle with the machine as Mealy model.

In examples of Moore machines we will also use the behavior table instead of the separate transition and output tables. Such a table looks like a transition table but it is extended by one column which contains outputs of states. An example of a behavior table and also a state diagram of some Moore machine is shown in Figure 3.2. As for Mealy machine, unimportant outputs are omitted. Outputs are only shown in the last column of the behavior table and in nodes of the state diagram, in particular.

$S = \{A, B, C\}, X = \{a, b\}, Y = \{1, 2, 3\}, s_0 = A$

	a	b	ε
A	B	B	1
B	A	C	3
C	B	C	2

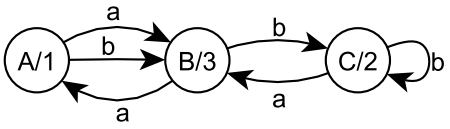


Figure 3.2. An example of the same Moore machine in two representations

3.2.1 Nondeterminism

Types Mealy and Moore can be generalized to be nondeterministic, that is a transition does not have to uniquely determine its next state. If we keep the main property of both models, i.e. labeling transitions or states with outputs, we can define nondeterministic variants of these specific FSM's types.

Definition 3.13. A *nondeterministic Mealy machine* is a FSM $M = (S, X, Y, s_0, h)$ with a specified behavior function $h : S \times X \to \mathcal{P}(S \times Y)$.

The nondeterministic finite-state machine is often defined like the nondeterministic Mealy machine (NMealy) [Wa10], [Lu95].

Definition 3.14. A *nondeterministic Moore machine* is a FSM $M = (S, X, Y, s_0, h)$ with a specified behavior function h such that

(i) $h(s,x) \in \mathcal{P}(S \times \{\varepsilon\}) \quad \forall s \in S\, \forall x \in X$

(ii) $h(s,\varepsilon) = (s,y) \quad \forall s \in S\, \exists y \in Y$

The Definition 3.14 of nondeterministic Moore machine (NMoore) can be simplified by employing a transition function $\delta : S \times X \to \mathcal{P}(S)$ and an output function $\lambda : S \to Y$ in place of the behavior function h. Then a NMoore is a sextuple $(S, X, Y, s_0, \delta, \lambda)$.

3.3 Automaton

A finite automaton are a special type of FSM that divides all input sequences into a set of accepted and the rest. A sequence $u \in X^*$ is *accepted* by the finite automaton if u ends in the set of states $F \subseteq S$, i.e. $\delta^*(s_0, u) \in F$. The extended transition function δ^* is stated in Definition 3.19 and it is similar to the extension of the behavior function h, see Definition 3.4. States of F are called *accepting*, or *final*. A set of accepted sequences (strings) is *regular language* [Ho06]. It is known that a language is regular if and only if there is a finite automaton accepting such a set of strings.

Now we propose definitions of three types of finite automaton and their relation to FSM. Each finite automaton accepts a regular language however they differ in expressive power. If we fix the number of states, a deterministic finite automaton (DFA) is less expressive than a nondeterministic finite automaton (NFA) and expressiveness of a NFA is smaller than a nondeterministic finite automaton with ε-moves (NFA-ε). Nevertheless, each of these three types can be translated into each other in polynomial time; polynomial in the number of states. In addition, there is minimization algorithm, i.e. finding reduced form of finite automaton, that also works in polynomial time [Ho06].

Definition 3.15. A *deterministic finite automaton* is a quintuple (S, X, s_0, δ, F), where S, X, s_0 are a set of states, an input alphabet and the initial state, respectively, and

- δ is a transition function: $\delta : S \times X \to S$,
- F is a set of *accepting*, or *final*, states; $F \subseteq S$,

A DFA is a complete Moore machine $M = (S, X, Y, s_0, D, \delta, \lambda)$ with binary output alphabet $Y = \{0, 1\}$, such that $D = S \times X$ since it is complete and $\lambda(s) = 1$ iff $s \in F$.

Definition 3.16. A *nondeterministic finite automaton* is a quintuple (S, X, s_0, δ, F), where S, X, s_0 are a set of states, an input alphabet and the initial state, respectively, and

- δ is a transition function: $\delta : S \times X \to \mathcal{P}(S)$,
- F is a set of *accepting*, or *final*, states; $F \subseteq S$,

A NFA is a nondeterministic Moore machine $M = (S, X, Y, s_0, \delta, \lambda)$ with binary output alphabet $Y = \{0, 1\}$ such that $\lambda(s) = 1$ if and only if $s \in F$.

Definition 3.17. A *nondeterministic finite automaton with ε-moves* is a NFA $M = (S, X, s_0, \delta, F)$ with an extended transition function $\delta : S \times X_\varepsilon \to \mathcal{P}(S)$.

A NFA-ε is a FSM $M = (S, X, Y, s_0, h)$ with binary output alphabet $Y = \{0, 1\}$ such that $F = \{s \mid (s, 1) \in h(s, \varepsilon)\}$ and

(i) $h(s, x) = \{(s', \varepsilon) \mid s' \in \delta(s, x)\} \quad \forall s \in S\, x \in X$

(ii) $h(s, \varepsilon) = \{(s', \varepsilon) \mid s' \in \delta(s, \varepsilon) \wedge s \neq s'\} \cup \{(s, 1)\} \quad \forall s \in F$

(iii) $h(s, \varepsilon) = \{(s', \varepsilon) \mid s' \in \delta(s, \varepsilon) \wedge s \neq s'\} \cup \{(s, 0)\} \quad \forall s \notin F$

3.4 Comparison of FSMs

There are other types of finite-state machines. One can restrict NFSM to be *observable*, that is, the next state is determined by the transition (s, x) and the output [Do05i]. However, we focus mainly on deterministic machines and thus the stated definitions are sufficient to get familiar with more general concepts of FSM.

This section is devoted to a comparison of special cases of FSMs that we defined in the previous text. At first, we count upper bound of the number of different machines

that have the same numbers of states, inputs and outputs for each defined type of FSM. Let n denote the number of states, i.e. $n = |S|$, p the number of input symbols, $p = |X|$, and q the number of output symbols, $q = |Y|$. Upper bound for particular class is the number of combinations how states and/or transitions can be labeled using the output alphabet Y and how states S are connected. Note that finite automata have binary output alphabet, i.e. $q = 2$.

Type of FSM	complete FSM	Total number
DFA	$2^n \cdot n^{np} = (2n^p)^n$	$(2n^p)^n$
NFA	$2^n \cdot (2^n - 1)^{np}$	$2^n \cdot (2^n)^{np} = 2^{n^2p+n}$
NFA-ε	$2^n \cdot (2^n - 1)^{n(p+1)}$	$2^n \cdot (2^n)^{n(p+1)} = 2^{n^2(p+1)+n}$
Moore	$q^n \cdot n^{np} = (qn^p)^n$	$q^n \cdot (n+1)^{np} = (q(n+1)^p)^n$
Mealy	$(qn)^{np}$	$(qn+1)^{np}$
DFSM	$(q+1)^n \cdot ((q+1)n)^{np}$	$(q+1)^n \cdot ((q+1)n+1)^{np}$
NMoore	$q^n \cdot (2^n - 1)^{np}$	$q^n \cdot (2^n)^{np} = (q2^{np})^n$
NMealy	$(2^{qn} - 1)^{np}$	$(2^{qn})^{np} = 2^{qn^2p}$
NFSM	$(2^{(q+1)n} - 1)^{n(p+1)}$	$(2^{(q+1)n})^{n(p+1)}$

Table 3.1. The Maximal Number of Different Machines in a Specific Class of FSMs

Table 3.1 shows derived upper bounds for each class of defined FSMs. We have counted sizes for the following types: deterministic finite automaton (Definition 3.15), nondeterministic finite automaton (Definition 3.16), nondeterministic finite automaton with ε-moves (Definition 3.17), Moore machine (Definition 3.12), Mealy machine (Definition 3.11), deterministic finite-state machine (Definition 3.10), nondeterministic Moore machine (Definition 3.14), nondeterministic Mealy machine (Definition 3.13) and nondeterministic finite-state machine (Definition 3.1). For each type of FSM there are two upper bounds. The first one is the maximal number of different machines that are completely specified. The second bound is the total number of machines in particular class; it includes partial FSMs. Note that the numbers are really upper bounds because some combinations represent equivalent machines.

Proposed definitions of different types of finite-state machine is connected in a way of specific restriction of the behavior function h. For example, DFSM allows a transition to enter at most one state and Moore model allows output symbols only by states. We can define a relation between classes of FSMs. Relations are oriented so we represent them as a directed graph with classes as nodes and relations as edges. There is a directed edge from class A to class B if there is a possibility to extend the definition of class A to obtain the definition of class B, or equivalently, one can restrict class B more so class A is defined. Transitivity holds in the relation graph. Therefore, we captured only relations of closest classes in Figure 3.3 for clarity.

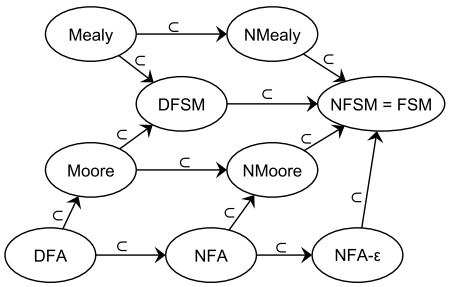


Figure 3.3. Relations of FSMs' Classes

As soon as we have upper bounds for each class of FSMs and relations of classes, we can compare how approximately smaller a specific class of FSMs is than another class that includes the former one. If we deal with a class of complete machines, 'C' precedes the name of the specific class of FSMs. Ratios of classes connected in Figure 3.3 follow. Note that the numbers of states, inputs and outputs are still fixed.

$$\frac{\text{DFA}}{\text{NFA}} = \left(\frac{2n^p}{2^{np+1}}\right)^n = \left(\frac{n}{2^n}\right)^{np} \qquad \frac{\text{NFA}}{\text{NFA-}\varepsilon} = \left(\frac{2^{np+1}}{2^{np+n+1}}\right)^n = \left(\frac{1}{2^n}\right)^n$$

$$\frac{\text{NFA-}\varepsilon}{\text{NFSM}} = \left(\frac{2^{np+n+1}}{2^{(q+1)n(p+1)}}\right)^n = \left(\frac{2}{2^{qn(p+1)}}\right)^n \qquad \frac{\text{CDFSM}}{\text{NFSM}} = \left(\frac{(q+1)^{n+p}\cdot n^p}{2^{(q+1)n(p+1)}}\right)^n$$

$$\frac{\text{DFA}}{\text{CMoore}} = \left(\frac{2n^p}{qn^p}\right)^n = \left(\frac{2}{q}\right)^n \qquad \frac{\text{NFA}}{\text{NMoore}} = \left(\frac{2^{np+1}}{q2^{np}}\right)^n = \left(\frac{2}{q}\right)^n$$

$$\frac{\text{CMealy}}{\text{NMealy}} = \left(\frac{(qn)^p}{2^{qnp}}\right)^n = \left(\frac{qn}{2^{qn}}\right)^{np} \qquad \frac{\text{CMoore}}{\text{NMoore}} = \left(\frac{qn^p}{q2^{np}}\right)^n = \left(\frac{n}{2^n}\right)^{np}$$

$$\frac{\text{CMealy}}{\text{CDFSM}} = \left(\frac{(qn)^p}{(q+1)^{n+p}\cdot n^p}\right)^n = \left(\frac{q^p}{(q+1)^{n+p}}\right)^n \qquad \frac{\text{CMoore}}{\text{CDFSM}} = \left(\frac{qn^p}{(q+1)^{n+p}\cdot n^p}\right)^n = \left(\frac{q}{(q+1)^{n+p}}\right)^n$$

$$\frac{\text{NMealy}}{\text{NFSM}} = \left(\frac{2^{qnp}}{2^{(q+1)n(p+1)}}\right)^n = \left(\frac{1}{2^{n(q+p+1)}}\right)^n \qquad \frac{\text{NMoore}}{\text{NFSM}} = \left(\frac{q2^{np}}{2^{(q+1)n(p+1)}}\right)^n = \left(\frac{q}{2^{qn(p+1)+n}}\right)^n$$

In the next section we will state that we deal mainly with complete Moore and Mealy machines in this thesis. Hence the ratio of these two classes is interesting for us although their relation was not set. In Section 3.2 we mentioned that a Mealy model can represent more complex systems than a Moore model with the same number of states. Consequently, the class of Mealy machines is larger than the Moore one. How much is captured in the following ratio.

$$\frac{\text{CMoore}}{\text{CMealy}} = \left(\frac{qn^p}{(qn)^p}\right)^n = \left(\frac{1}{q^{p-1}}\right)^n$$

Our last comparison of FSMs' classes is depicted in Figure 3.4. There is again captured relations of type classes as in Figure 3.3. Nevertheless, this time the relations are shown more illustratively. Figure 3.4 is splitted into two parts. Both rectangles represent the entire class of finite-state machines with the fixed numbers of states, inputs and outputs. The left part shows relations of classes: deterministic finite-state machine, nondeterministic and deterministic Moore and Mealy machines. Notice that there is a Moore machine that cannot be modeled by a Mealy machine with the same number of states. The right part of Figure 3.4 depicts besides relations of finite automata even relations of some property classes of FSMs. In particular, there are captured classes of minimal, completely specified, strongly connected and initially connected machines.

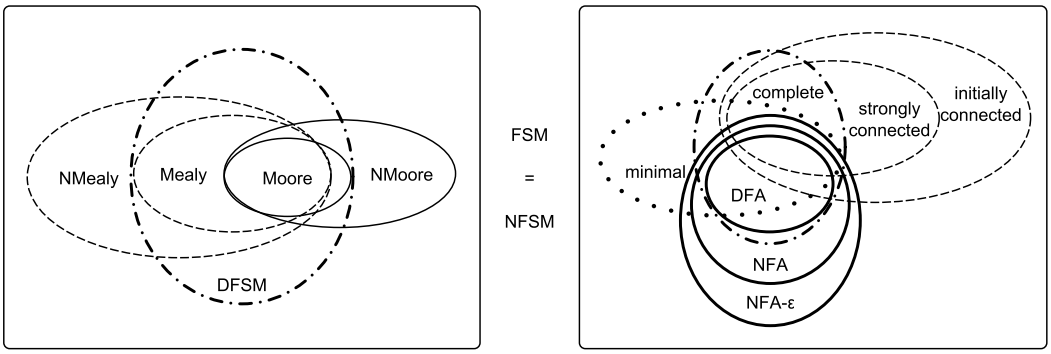


Figure 3.4. Relations of Type and Property Classes of Finite-State Machines

3.5 Use in the Thesis

This thesis focuses on testing Moore and Mealy machines. For the sake of simplicity, when we will refer *finite-state machine*, *FSM*, *automaton* or simply *machine*, we mean DFSM $M = (S, X, Y, s_0, D, \delta, \lambda)$, see Definition 3.10. In addition, M is restricted to Moore and Mealy machines, i.e. the output function λ is $\lambda : S \to Y$ for Moore model and $\lambda : D \to Y$ for Mealy model. The transition function $\delta : D \to S$ is the same for both types.

Another restrictions on machine M will be stated for particular need. We usually deal with a *minimal completely specified* FSM which is *initially* or *strongly connected*.

Some notions stated before in this chapter will be now simplify using the transition function δ and the output function λ instead of the behavior function h.

Definition 3.18. A transition $(s, x) \in S \times X$ is *defined* if and only if $(s, x) \in D$. An input sequence $u = x_1 \cdot \ldots \cdot x_k \in X^*$ is defined for state $s \in S$ if and only if there is a sequence of states $(s_i)_{i=1}^{k+1}$ such that $s = s_1$ and for all $1 \leq i \leq k : (s_i, x_i) \in D \wedge s_{i+1} = \delta(s_i, x_i)$. The set of all defined input sequences for state s is denoted $\Omega(s)$. The set of all defined input sequences for the initial state s_0 of FSM M is Ω_M, i.e. $\Omega_M = \Omega(s_0)$.

Definition 3.19. Extended transition function $\delta^* : D^* \to S$, $D^* \subseteq S \times X^*$, is defined inductively by

$$(0) \quad \delta^*(s, \varepsilon) = s \quad \forall s \in S$$
$$(1) \quad \delta^*(s, xu) = \delta^*(\delta(s, x), u) \quad \forall (s, x) \in D \, \forall u \in \Omega(\delta(s, x))$$

The extended transition function describes the last state after processing of the entire defined input sequence. The transition function δ and its extended version δ^* will be also used on a set of states or a set of sequences. Result is a set of states in both cases.

$$\delta^*(Q, u) = \{\delta^*(s, u) \mid s \in Q\} \quad Q \subseteq S, u \in X^*$$
$$\delta^*(s, U) = \{\delta^*(s, u) \mid u \in U\} \quad s \in S, U \subseteq X^*$$

When we need to describe the output sequence obtained as a response to a defined input sequence, we use extended output function.

Definition 3.20. Extended output function $\lambda^* : D^* \to Y^*$, $D^* \subseteq S \times X^*$, is defined inductively by

$$(0) \quad \lambda^*(s, \varepsilon) = \lambda(s, \varepsilon) \quad \forall s \in S$$
$$(1) \quad \lambda^*(s, xu) = \lambda(s, x) \cdot \lambda^*(\delta(s, x), u) \quad \forall (s, x) \in D \, \forall u \in \Omega(\delta(s, x))$$

Definition 3.20 is proposed for DFSM (Definition 3.10) in general. We remind that $\lambda(s, \varepsilon) = \varepsilon$ for Mealy machines (Definition 3.11) and $\lambda(s, x)$ could be adjusted for Moore machines (Definition 3.12) in Definition 3.20 (1) as $\lambda(s, \varepsilon) \cdot \lambda(s, x)$ which produces the desired output.

Definition 3.21. A FSM M is *completely specified*, or simply *complete*, if each transition is defined, i.e. $D = S \times X$. Otherwise, the FSM M is *partially specified*, or simply *partial*, i.e. when $D \subset S \times X$.

Definition 3.22. A FSM M is *initially connected* if and only if each state is reachable from the initial state, i.e. for each state s there exists a defined input sequence from the initial state s_0 to s; formally, $\forall s \in S \, \exists u \in \Omega_M : s = \delta^*(s_0, u)$.

Definition 3.23. A FSM M is *strongly connected* if and only if each state is reachable from each other. That is, $\forall s, s' \in S \, \exists u \in \Omega(s) : s' = \delta^*(s, u)$.

Definition 3.24. Two states $s_i, s_j \in S$ are *distinguishable*, denoted $s_i \nsim s_j$, if there is an input sequence $u \in \Omega(s_i) \cap \Omega(s_j)$ such that the output sequences produced as response to applying u in both states are different, i.e. $\lambda^*(s_i, u) \neq \lambda^*(s_j, u)$. The sequence u is called a *separating sequence* of states s_i and s_j.

The following two definitions remain same like for general FSM, see Definition 3.8 for *minimal* FSM and Definition 3.9 defining *reset* property.

Definition 3.25. A FSM M is *minimal*, or *reduced*, if and only if it is initially connected and each two states are distinguishable, i.e. $\forall s_i \neq s_j \in S : s_i \nsim s_j$.

Definition 3.26. A FSM M is *resettable* if and only if M can be taken to its initial state s_0 from each state. Formally, there is an input $r \notin X$, called *reset*, that brings M into s_0 with the output $null \notin Y$, i.e. $\forall s \in S : \delta(s, r) = s_0 \wedge \lambda(s, r) = null$.

3.6 Input sequences

An input sequence can have special property with respect to a machine. For example, response to a sequence is unique for a state or even for each state of the machine. We have studied sequences with identification and verification purpose in our previous work [So14]. The basic identification sequences are discussed there in detail; besides definitions there are construction algorithms and examples. The basic identification sequence are preset and adaptive distinguishing sequence, state verifying sequence, state characterizing set, homing sequence and synchronizing sequence. Here we propose only definitions of those we will use. The definitions usually assume complete FSM $M = (S, X, Y, s_0, D, \delta, \lambda)$, therefore the input sequence is not stated as 'defined' explicitly.

Definition 3.27. A *preset distinguishing sequence* (PDS) for a machine is an input sequence u such that the output sequence produced by the machine in response to u is different for each initial state, i.e., $\lambda^*(s_i, u) \neq \lambda^*(s_j, u)$ for every pair of states $s_i \neq s_j$.
(Definition 2.1 in [Le94])

Definition 3.28. An *adaptive distinguishing sequence* (ADS) is a rooted tree R with exactly n leaves; the internal nodes are labeled with input symbols, the edges are labeled with output symbols, and the leaves are labeled with states of the FSM such that: 1) edges emanating from a given node have distinct output symbols, and 2) for every leaf of R, if u, z are the input and output strings, respectively, formed by the node and edge labels on the path from the root to the leaf, and if the leaf is labeled by state s_i of the FSM then $z = \lambda^*(s_i, u)$. The length of the sequence is the depth of the tree.
(Definition 3.1 in [Le94])

Instead of representation ADS as a tree, one can list the sequences forming paths from the root to leaves. A set of these sequences is called *distinguishing set* in [Bo74]. Then we denote d_i the particular distinguishing sequence related to state s_i.

Definition 3.29. A *state verifying sequence* of a state $s \in S$ is an input sequence $u \in X^*$, such that the output sequence produced by the machine in response to u from any state other than s is different than that from s, i.e., $\lambda^*(s_i, u) \neq \lambda^*(s, u)$ for any $s_i \neq s$.
(Definition 4.1 in [Le94])

State verifying sequence (SVS) is usually called *Unique Input Output Sequence* (UIOS) in the literature. However, if one deals with deterministic machines that uniquely determine the output sequence when an input sequence is applied in a state, the notion 'state verifying sequence' better expresses the purpose of the sequence.

Definition 3.30. A *state characterizing set* W_i of a state $s_i \in S$ is a set of input sequences $u_k \in X^*$, such that the set of output sequences produced by the machine in response to all u_k from any state other than s_i is different than that from s_i, i.e., $\{\lambda^*(s_j, u_k) \mid u_k \in W_i\} \neq \{\lambda^*(s_i, u_k) \mid u_k \in W_i\}$ for any $s_j \neq s_i$.

In other words, for each state different from s_i there exists an input sequence u_k in the state characterizing set of state s_i, such that the output sequences produced by the machine in response to u_k are different, i.e., $\forall s_j \neq s_i \; \exists u_k \in W_i : \lambda^*(s_j, u_k) \neq \lambda^*(s_i, u_k)$.

State characterizing set (SCSet) is called *state identifier* in [Pe91]. We will need a special relation of state characterizing sets that ensures a separating sequence of s_i and s_j is contained in related sets.

Definition 3.31. For a family, or a set, H of *harmonized state identifiers* $H_1, ..., H_n$ the following holds: $\forall s_i \neq s_j \in S$, $\exists u \in \text{pref}(H_i) \cap \text{pref}(H_j) : \lambda^*(s_i, u) \neq \lambda^*(s_j, u)$.

In case when one wants to distinguish all states from each other, a characterizing set (CSet), or *characterization set* [Ch78], can be used.

Definition 3.32. A *characterizing set* W is a set of input sequences $u_k \in X^*$, such that for each pair of states $s_i \neq s_j$, there is sequence $u_k \in W$ that distinguishes these two states, i.e., $\lambda^*(s_i, u_k) \neq \lambda^*(s_j, u_k)$.

Definition 3.33. An input sequence u is said to be a *homing sequence* (HS) if the final state of the machine can be determined uniquely from the machine's response to u, regardless of the initial state. These final states of the machine are determined by observing the output sequence produced by applying a homing sequence to the machine. (Definition in [De94])

It is worth mentioning that not each FSM has each basic sequence. We proposed relations of basic sequences and class of finite-state machines in [So14]. In short, each FSM having preset distinguishing sequence has adaptive distinguishing sequence, each FSM having ADS has state verifying sequence for each state and each FSM having SVS for each state has state characterizing set for each state. The contrary does not hold; machine with ADS does not have to have PDS, for example. Furthermore, every reduced FSM has SCSet for each state, family of harmonized state identifiers, characterizing set and homing sequence. However, none of basic sequences can be unique in given machine, e.g. a FSM can have several minimal length HS or PDS.

Notions of state and transition cover sets are now defined as usual in the automata testing field [Fu91]. They are used to declare that each state or each transition is visited during the testing process.

Definition 3.34. An input sequence $u \in \Omega_M$ is called *transfer* for state s if $\delta(s_0, u) = s$. A *state cover*, denoted SC, of FSM M is a set of defined input sequences such that there exists a transfer sequence for each state, i.e. $\delta^*(s_0, SC) = S$. The state cover is *minimal* if it contains n sequences, i.e. $|SC| = |S|$.

Definition 3.35. Given FSM M, a transition (s, x) is covered in a defined sequence $u \in \Omega_M$, or u covers the transition (s, x), if and only if there are input sequences $v, w \in X^*$ such that $u = vxw$ and $\delta(s_0, v) = s$. A *transition cover*, denoted TC, of FSM M is a set of defined input sequences such that each transition is covered in a sequence of TC, i.e. $\forall (s, x) \in D \; \exists u = vxw \in TC : \delta^*(s_0, v) = s$.

The last definition is a special concatenation of sequence sets. It will simplify the description if one wants to extend each sequence of given set by different set of sequences. Note that extension of a sequence is proposed in Definition 2.13.

Definition 3.36. Let V_i be a set of defined input sequences of state s_i, i.e. $V_i \subseteq \Omega(s_i)$. Let V be a set of V_i for FSM M such that each state s_i has the related V_i in V. Let U be a set of defined input sequences of FSM M; $U \subseteq \Omega_M$. U is *extended by related sets* V_i, denoted $U \circ V_i$, if each sequence $u \in U$ is extended by set V_i that is related to the final state s_i when u applied in the initial state s_0. Formally,

$$U \circ V_i = \bigcup_{u \in U} \{u \cdot v \mid v \in V_i \wedge V_i \text{ relates to } s_i = \delta^*(s_0, u)\}$$

3.7 Denotation unification

The literature dealing FSMs and their testing is very extensive and it contains wide range of different notations. In this section, we propose a unification of denotations and reasons why we have chosen such symbols to represent particular notion.

We use uppercase for sets, lowercase for elements and the Greek alphabet for function mainly. Particularly, as in the previous sections, we denote FSM by $M = (S, X, Y, s_0, D, \delta, \lambda)$, where the set of states is $S = \{s_0, s_1, \ldots, s_{n-1}\}$, the input alphabet is $X = \{x_1, \ldots, x_p\}$, the output alphabet is $Y = \{y_1, \ldots, y_q\}$, the initial state is $s_0 \in S$, the domain $D \subseteq S \times X$ and δ, λ are the transition and output functions. The sizes of sets are as follows: $n = |S|$, $p = |X|$ and $q = |Y|$. The same denotation of sizes is used in [Gi62] and [Le96p] but the former uses Z for the output alphabet and the latter I and O for the input and output alphabets. These notations are also very accurate however we prefer X and Y due to the fact that O coincides with 0 (zero) and we use I rather as a set of indexes. When we need to refer a second FSM, we use machine $N = (Q, X', Y', q_0, D', \Delta, \Lambda), m = |Q|$. Notice the correspondence between M and N; the input and output alphabets and the domains only differ in the prime symbol, the transition and output functions use the same Greek letters but lower or upper case. The only difference is denotation of states; M has s_i and N has q_j.

Input sequences, or strings over the input alphabet X, are denoted by u, v, w, t. The sequence $t \in X^*$ usually refers to a test case and T, $t \in T$, is a test suite, see Definition 4.1. Some other lowercase letters can be used for an input sequence in special cases, d_i is a distinguishing sequence of state s_i, for example. An output sequence, or a string over the output alphabet Y, is mainly represented by z.

The characters from the beginning of the alphabet are employed for labeling states and inputs in examples. We use $A, B, C, \ldots$ for denotation of states and $a, b, c, \ldots$ for input symbols, in particular. Then, output symbols are represented by numbers $0, 1, 2, 3, \ldots$. The letters i, j, k, l are reserved for indexing. A set of indexes is usually denoted by I and often contains numbers beginning from 0, e.g. $i \in I = \{0, 1, 2\}$. The reset input is always denoted by r. Some other letters are reserved for particular use, e.g. W (W_i) denotes a (state) characterizing set, H is a family of state identifiers (H_i), F is a fault domain (Definition 4.4) and so on.

The last note is about use of the Greek alphabet. We have already defined a function of symbols δ, Δ, λ and Λ. We refresh that the empty string is ε and the set of defined input sequences for state s is $\Omega(s)$. Some authors use upper Greek letters for sets of symbols, e.g. Σ for the input alphabet [Ho06], however, we consider X, Y as apter for the input and output alphabets. The letters X and Y are simpler and correspond to well-used mathematical formula for function $f : X \to Y$. Other authors apply $\alpha, \beta, \ldots$ as input sequences [Si10]. But we prefer u, v, w, t because of the mentioned division of alphabets; uppercase of the Latin alphabet for sets, lowercase of the Latin alphabet for elements and the Greek alphabet for functions.

Chapter 4
Test properties

The previous chapter stated different types of FSM and their properties. That is, we dealt with one machine so far. Now we will focus on how to check whether two (or more) machines do the same. In other words, we will compare machines if they have equal behavior. This process is called *conformance testing*, or *fault detection* [Le96p]. It takes a FSM M, called a *specification*, and generates *tests* based on the specification. Tests are input sequences if M is deterministic, and input/output sequences if M is not deterministic. Tests are generated according to chosen method, see Chapter 6. Testing consists of generating tests and applying each test to the *implementation under test* (IUT), or simply the *implementation*. The implementation can contain any fault in general. It would be impossible to confirm machine equivalence unless potential faults were restricted by a *fault model*.

4.1 Fault model

A fault model describes possible faults that may occur in the implementation under test. The most common fault model is that the implementation is modeled by a FSM N with up to m states. The maximal number of states m is known apriori.

Before we will state property of tests, we need to formally define a test and relations between machines.

Definition 4.1. A defined input sequence of FSM M is called a *test case*, or simply *test*, of M. A *test suite* T of M is a finite prefix-closed set of tests of M. A test $t \in T$ is *maximal* (with respect to T), if it is not a proper prefix of another test in T.

Only maximal test sequences are applied in testing because they cover all responses to the entire test suite. If a test suite contains only one maximal sequence, the sequence is called *checking sequence* (CS). On the other hand, if a test suite contains several tests we need to consider resettable machine and the reset is used before a test case is applied. Therefore, the *length of test suite* T is sum of tests' lengths increased by 1, i.e. $\text{len}(T) = \sum_{t \in T}(|t| + 1)$.

The following definition of relations between states of FSM is adapted from [Pe05].

Definition 4.2. Given a FSM $M = (S, X, Y, s_0, D, \delta, \lambda)$ and states $s_i, s_j \in S$,

(i) s_i and s_j are *compatible*, written $s_i \sim s_j$, if $\Omega(s_i) \cap \Omega(s_j) = \emptyset$ or for all $u \in \Omega(s_i) \cap \Omega(s_j) : \lambda^*(s_i, u) = \lambda^*(s_j, u)$;

(ii) s_i is *quasi-equivalent* to s_j, written $s_i \sqsupseteq s_j$, if $\Omega(s_i) \supseteq \Omega(s_j)$ and for all $u \in \Omega(s_j) : \lambda^*(s_i, u) = \lambda^*(s_j, u)$;

(iii) s_i and s_j are *equivalent*, written $s_i \cong s_j$, if $\Omega(s_i) = \Omega(s_j)$ and for all $u \in \Omega(s_j) : \lambda^*(s_i, u) = \lambda^*(s_j, u)$;

(iv) s_i and s_j are *distinguishable* (by u), written $s_i \nsim s_j$ ($s_i \nsim_u s_j$), if there exists an input sequence $u \in \Omega(s_i) \cap \Omega(s_j)$, called a *separating* sequence, such that $\lambda^*(s_i, u) \neq \lambda^*(s_j, u)$; given a set of input sequences U, we also write $s_i \nsim_U s_j$ if $s_i \nsim_u s_j$ and $u \in U$.

Notice that distinguishable states, Definition 4.2 (iv), were defined previously even for general FSM in Definition 3.7.

The relations quasi-equivalence, equivalence and distinguishability can be applied to states from different machines and so extended to relation between machines.

Definition 4.3. Let $M = (S, X, Y, s_0, D_M, \delta, \lambda)$ and $N = (Q, X, Y, q_0, D_N, \Delta, \Lambda)$ be FSMs such that $S \cap Q = \emptyset$. Then,

(i) M is *quasi-equivalent* to N, written $M \sqsupseteq N$, if $s_0 \sqsupseteq q_0$, i.e. $\Omega_M \supseteq \Omega_N$ and for all $u \in \Omega_N : \lambda^*(s_0, u) = \Lambda^*(q_0, u)$;

(ii) M and N are *equivalent*, written $M \cong N$, if $s_0 \cong q_0$, i.e. $\Omega_M = \Omega_N$ and for all $u \in \Omega_M : \lambda^*(s_0, u) = \Lambda^*(q_0, u)$;

(iii) M and N are *distinguishable* (by u), written $M \nsim N$ ($M \nsim_u N$), if there exists an input sequence $u \in \Omega_M \cap \Omega_N$ such that $\lambda^*(s_0, u) \neq \Lambda^*(q_0, u)$; given a set of input sequences U, we also write $M \nsim_U N$ if $M \nsim_u N$ and $u \in U$.

According to Definition 4.3, both machines operates over the same alphabet X. If it was not so, the machines would become harder to compare.

The quasi-equivalence relation is also called *weak conformance* [Le96p] Then *strong conformance* refers to the equivalence relation.

Definition 4.4. Given FSM $M = (S, X, Y, s_0, D, \delta, \lambda)$, a *fault domain* $F_m(M)$ of the machine M is a set of finite-state machines over the alphabet X with up to m states. A FSM $N \in F_m(M)$ is called

- *conforming* implementation of the M if $N \sqsupseteq M$, or
- *nonconforming* implementation of the M if $N \nsim M$.

A fault domain $F_m(M)$ can be specified by a fault function Φ, as it is stated in [Pe92].

Definition 4.5. A fault function Φ for the given machine $M = (S, X, Y, s_0, D_M, \delta, \lambda)$ describes possible faults in the implementation $N = (Q, X, Y', q_0, D_N, \Delta, \Lambda)$. Formally, $\Phi : Q \times X \to \mathcal{P}(Q \times Y')$, where $Q \supseteq S$, $Y' \supseteq Y$ and $\mathcal{P}(Q \times Y')$ is power set of $Q \times Y'$, and $\forall (s, x) \in D : (\delta(s, x), \lambda(s, x)) \in \Phi(s, x)$. Then, a transition (s, x) is called *valid* if $\Phi(s, x) = \{(\delta(s, x), \lambda(s, x))\}$, otherwise the transition is called *suspicious*.

A fault function Φ is only stated for Mealy machine M because the restricted fault domain $F_m(M)$ contains nondeterministic Mealy machines. Definition 4.5 can be adapted for Moore machine by extension of the alphabets with the empty string ε.

There are several types of faults in the implementation N, $m = |Q|$:

1. Same type of fault and $m = n$, i.e. the implementation has no extra states:
   - transfer faults: $\Phi(s, x) = \{(s', \lambda(s, x)) \mid s' \in S\}$ for all $(s, x) \in S \times X$
   - output faults: $\Phi(s, x) = \{(\delta(s, x), y) \mid y \in Y\}$ for all $(s, x) \in S \times X$
   - input faults: $\Phi(s, x_i) = \Phi(s, x_j) = \{(\delta(s, x_i), \lambda(s, x_i)), (\delta(s, x_j), \lambda(s, x_j))\}$ $\forall s \in S$
2. Arbitrary type of fault and $m > n$: $\Phi(s, x) = Q \times Y'$ for all $(s, x) \in Q \times X$

4.2 Completeness

Definition 4.6. A test suite T is *F_m-complete* if and only if T distinguishes each nonconforming implementation $N \in F_m(M)$ from the specification M, i.e. $N \nsim_T M$.

The number of states m in the IUT are usually assumed to be greater or equal to the number of states n in the reduced specification M, i.e. $m \geq n$. In other words, the

implementation can have extra states. If the fault domain F_m is not further restricted, i.e. it contains all machines with up to m states, we say m-complete test suite instead of F_m-complete. It is clear that if T is m-complete then it is k-complete, $k < m$, as well. We deal with minimal FSMs, therefore $F_k(M)$ for $k < n$ cannot contain a conforming implementation of M. Nonetheless, there is also a method confirming p-completeness, $p \leq n$, see P-method in Chapter 6. The reason for a special method is reduction of the size of test suite. Note that p in the notion 'p-completeness' does not correspond to the size of the input alphabet X. As a reminder, we use p for $|X|$ throughout the text however with connection to p-complete test suite it only means $p \leq n = |S|$.

4.3 Sufficient conditions

Let's consider a resettable initially connected machine first. Sufficient conditions for m-completeness in general or for n-completeness if $m = n$ are stated in [Pe96] or [Do05i]. Their reformulation into a condensate form is proposed in the next theorem.

Theorem 4.1. A test suite T is m-complete with respect to the reduced machine M with n states if T contains the set P of sequences $SC \otimes X^{m-n+1} \cap \Omega_M$ and the following conditions hold:

(1) for each $u \in SC$ and each $v \in P$ such that $\delta^*(s_0, u) \neq \delta^*(s_0, v)$, there should be two sequences $uw, vw \in T$ such that $\lambda^*(\delta^*(s_0, u), w) \neq \lambda^*(\delta^*(s_0, v), w)$; and

(2) if $m > n$ then for each $u \in P \setminus SC$ and each $v \in \text{pref}(u) \setminus SC$ such that $\delta^*(s_0, u) \neq \delta^*(s_0, v)$, there should be two sequences $uw, vw \in T$ such that $\lambda^*(\delta^*(s_0, u), w) \neq \lambda^*(\delta^*(s_0, v), w)$.

Theorem 4.1 (1) is divided to two cases in the literature. All states are distinguished from each other by extension of SC at first. Then remaining transitions are verified. Theorem 4.1 (1) verifies the next state against states reached by SC. On the other hand, Theorem 4.1 (2) verifies transitions that may be connected with an extra state. Theorem 4.1 (2) has to be checked only if an implementation is assumed to have more states than the specification.

A sufficient condition based on use of distinguishing sequence and notions of d- and *t-recognized* states in the checking sequence is stated in [Ur97]. These notions are basis for a lot of testing methods creating checking sequence, especially D-method in Section 6.2.2, however they were further generalized with the notion of a *confirmed set*.

A weaker sufficient condition that needs neither reset nor DS is proposed in [Si08] (restricted to checking sequence) and in [Si10] (for general use). It is based on notion a confirmed set and it is proposed for n-completeness of test suite. Let $F_T(M)$ be a set of all machines from the fault domain $F(M)$ that have the same response to the test suite T as the specification M. The next definition and theorem consider only n-completeness, hence the corresponding fault domain is $F_n(M)$.

Definition 4.7. Let T be a test suite of a FSM $M = (S, X, Y, s_0, D, \delta, \lambda)$ and $K \subseteq T$. The set K is *confirmed* if $\delta^*(s_0, K) = S$ and, for each $N \in F_T(M)$, it holds that for all $u, v \in K : \Delta^*(q_0, u) = \Delta^*(q_0, v)$ if and only if $\delta^*(s_0, u) = \delta^*(s_0, v)$. An input sequence is *confirmed* if there exists a confirmed set that contains it.

We refer [Si09c] for the next notions. When we need to indicate the final state reached by $u \in K$ then we say that u is *confirmed as state* $s = \delta^*(s_0, u)$ (in M), or that it is *s-confirmed*. Sequence v is said to be *verified in state* s (or *s-verified*) if u is s-confirmed and uv is s'-confirmed, $s' = \delta^*(s, v)$. We say that the transition (s, x) is

verified if x is s-verified. We say that a sequence v is *output-confirmed in state s* (or *s-output-confirmed*), if for any $N \in F_T(M)$, it holds that $\Lambda^*(q_s, v) = \lambda^*(s, v)$, where $q_s \in Q$ is the state reached in N by an s-confirmed sequence.

Theorem 4.2. Let T be a test suite of an initially connected reduced FSM $M = (S, X, Y, s_0, D, \delta, \lambda)$ with n states. T is n-complete for M, if there exists a confirmed set $K \subseteq T$ such that empty sequence is confirmed and each defined transition is verified.

The Theorem 4.2 is a composition of theorems from [Si08, Si09c] and [Si10]. The theorem holds even if the test suite T has only one maximal sequence, that is, checking sequence is under consideration as in [Si09c], or the machine is partially specified [Si08].

Convergence and divergence of tests w.r.t. a set of FSMs is another notion that can condition completeness of a test suite. These notions are proposed in [Si09c, Si12] for example.

Definition 4.8. Given a set of FSMs F, two tests are *F-convergent*, if they converge (i.e. transfer from the initial state to the same state) in each FSM of the set F; and two tests are *F-divergent*, if they diverge (i.e. transfer from the initial state to different states) in each FSM of F.

We use F to denote the set of FSMs on purpose. F denoted a fault domain up to now and it will continue. A fault domain is defined as a set of FSMs and the Definition 4.8 will be employed just with connection to fault domain. A consequence of the Definition 4.8 follows. Two sequences u and v are $F_T(M)$-convergent if for each $N \in F_T(M)$, it holds that $\Delta^*(q_0, u) = \Delta^*(q_0, v)$. We say that u and v $F_T(M)$-*converges* [Si09c]. As a reminder, $F_T(M)$ is a set of all machines from the fault domain $F(M)$ that have the same response to the test suite T like the specification M. For simplicity, we use F_T without the corresponding specification M when it is clear from the context. Similarly, the set F in which tests are convergent or divergent will be omitted, and M-convergent or M-divergent tests of a machine M is used instead of $\{M\}$-convergent or $\{M\}$-divergent.

Convergence relation is reflexive, symmetric and transitive, i.e. it is an equivalence relation over the set of tests [Si12]. Tests $u_i \in T$ can be thus partitioned into corresponding equivalence classes $[u_i] = \{u_j \in T \mid u_i \text{ and } u_j \text{ converge}\}$.

Definition 4.9. Given a test suite T and a fault domain $F(M)$ containing the specification FSM M, a set of tests in T is *F_T-convergence-preserving* if all its M-convergent tests are F_T-convergent; a set of tests in T is *F_T-divergence-preserving* if all its M-divergent tests are F_T-divergent.

The last sufficient condition stated in Theorem 4.3 and [Si12] is applicable to m-complete test suite and it is based on convergence of tests. It employs a notion of initialized set. A set A is *initialized* if it contains the empty sequence, i.e. $\varepsilon \in A$.

Theorem 4.3. Let T be a test suite for a FSM M and $F(M)$ be a related fault domain. If T contains a F_T-convergence-preserving initialized transition cover for M, then T is a F-complete test suite for M.

Proofs of the proposed theorems can be found in the referenced literature.

Chapter 5
Related Work

Checking experiment has been studied by many researchers for the last six decades. In 1956, Moore [Mo56] stated an upper bound of the length of checking sequence. Gill [Gi62] proposed some additional distinguishing experiments and their upper bounds in 1962. Two years later, Hennie [He64] described the construction of a checking sequence based on preset distinguishing sequence, aka the *D-method*. He also proposed a modification of the method to be applicable to machines without PDS. We call it the *DW-method*. His work became a basis for others.

Hsieh proposed a quite different testing method in 1971 [Hs71] which we call the *HrADS-method* and is able to create even m-complete checking sequence. This method employs a sufficient condition that was later formulated by Vasilevskii [Va73] and Chow [Ch78] to design the *W-method*. The W-method creates a test suite with several sequences so it is applicable only to resettable machines.

More than a few new methods have been proposed since then. The *Wp-method* [Fu91], the *HSI-method* [Pe91], the *FF-method* [Pe92], the *H-method* [Do05i], the *SC-method* [Pe05], the *P-method* [Si09f] and the *SPY-method* [Si12] generate several test cases as the W-method and so they assume reliable reset in each state of the given machine. Section 6.1 describes these methods in more detail. On the contrary, testing methods producing a checking sequence are discussed in Section 6.2. These methods include the D-method which was extensively improved for example in [Go70, Ur97, Hi06, Ch05, Ur06, Du09], the *AD-method* [Bo74, Hi09], the DW-method reformulated in [Re95] and improved to the *DWp-method* [Po13], the UIO-methods, the *CSP-method* [Vu90], the *C-method* [Si08, Si09c] and the *K-method* [Ka10].

The UIO-methods consist of approaches that use state verifying sequences, or Unique Input Output sequences, for state verification. The *UIO-method* is simply derived from the D-method [Sa85, Sa88]. Nevertheless, the *UIOv-method* [Ch89] had to be proposed due to the fact the UIO-method has not a complete fault coverage as it was shown in [Si88]. A state can possess several SVSs, this fact is employed by the *MUIO-method* [Sh92] to reduce the length of checking experiment. Then a number of improvements of the UIO-methods including optimization of tests' composition or overlapping of test sequences was proposed [Ah91, Ch95i, Ch06, Du07, Wa10].

The aforementioned methods are deterministic but there are attempts to conform the implementation-specification equivalence by a probabilistic or random approach. For instance, a guided random walk is employed in [Le96c] and a test is selected at random with the estimation of probability of undetected error in [Ya95]. The latter approach relates to PAC-learning, described in [Ke94], for example. This thesis contains other general notions besides PAC-learning. One can find combinatorial problems like Traveling Salesman Problem (TSP), Maximum Flow and Minimum Cost Flow (MCF) or Rural Chinese Postman Problem (RCPP) in [La76] and the notion of a NP-complete problem and time complexity of an algorithm in [Ho06], for example.

Development of sufficient conditions for complete fault coverage of generated test suite (Section 4.3) goes hand in hand with improving of testing methods. In 1996,

Petrenko and Bochmann stated the condition of Theorem 4.1 for m-complete test suite proving almost all methods that produce several tests. Ural et al. [Ur97] proposed one year later sufficient conditions using preset distinguishing sequence and so the D-method was proved to generate a checking sequence, or an n-complete test suite. A sufficient condition for use of adaptive distinguishing sequence instead of PDS was posed by Hierons et al. in 2009 [Hi09]. With the notion of a confirmed set in 2008, Simão and Petrenko [Si08] stated a sufficient condition for n-completeness captured in Theorem 4.2. The authors then focused on creating of a confirmed set needed for Theorem 4.2. A sufficient condition for the existence of a confirmed set stated in Theorem 8.1 was described in 2010 [Si10]. To prove the SPY-method, Theorem 4.3 for m-completeness was proposed by Simão, Petrenko and Yevtushenko in 2012 [Si12].

Completeness of fault coverage of test suite was checked empirically before sufficient conditions were discovered. In 1988, Sidhu and Leung [Si88] tested produced test suites on randomly generated machines. This was sufficient to show that the UIO-method does not produce a complete test suite in general. Vuong and Ko proposed the CSP-method in 1990 [Vu90]. The method generates all machines that respond to the test suite as the specification. Constraint Satisfaction Problem is used for the generation of machines. Ghedamsi and Bochmann stated another approach that is able to locate fault and adds additional diagnostic test cases, considering one fault in [Gh92] and multiple faults in [Gh93]. In 1993, Zhu and Chanson [Zh93] formalized the CSP-method and added a constructive algorithm. A method based on minimization of FSMs was described by Yao et al. in 1994 [Ya94]. This method detects a complete test suite and is also applicable on partial FSMs. However, time complexity of the method as well as the CSP-method is exponential in the length of test suite. Moreover, reduction of partial FSMs is NP-complete [Go07]. A great survey of fault coverage of tests was done by Petrenko and Bochmann in 1996 [Pe96]. It comprises sufficient conditions and methods for complete fault coverage detection, even for different types of FSMs. A new algorithm for checking n-completeness of test suite was proposed by Simão and Petrenko in 2010 [Si10]. It is based on finding n-clique in a n-partite distinguishability graph created from tests. A discussion how to solve this NP-complete problem is included. The method is usable for partial and reduced FSMs with or without reset. State-Recognition Patterns introduced by Kapus-Kolar in 2012 [Ka12n] could be used for further improving of methods for checking fault coverage completeness because they propose new ways how to distinguish states in test cases.

Testing methods are compared with each other from time to time. Here is a few examples of methods' comparison: [Si89, Ur92, Do05e, En13]. Lower bounds of a checking sequence generated by different versions of the D-method are discussed in [Jo10]. Lee and Yannakakis showed that design of PDS and SVS are PSPACE-complete in 1994 [Le94]. The authors summarized principles of testing FSMs two years later [Le96p].

A design procedure for charactering set as well as for related separating sequences, state characterizing sets and a family of harmonized state identifiers is needed for some testing methods. Such a procedure was described by Gill in 1962 [Gi62]. The method is based on minimization algorithm and its partition tables. In 1971, Hopcroft [Ho71] showed that a FSM can be reduce in $O(n \log n)$ however an inverse transition function δ^{-1} is required. In appendix of [Lu95] there is a design algorithm generating CSet and HSIs but it is not constructive and it relies on partitions again. Another design method of CSet with theorems proving correctness is proposed in [Mi10] however it contains small mistake in Theorem 3 which can influence the proposed algorithm. Compared to our algorithms in Section 10.2 and [So14] their method requires more space.

Chapter 6
Testing methods

Conformance testing proves that a system complies with its specification. A fault domain must be set to be possible to make a conclusion after testing. We focus on completely-specified deterministic finite-state machines that represent the specification. Our fault domain contains all deterministic machines with up to m states, i.e. m-complete test suite design methods are considered. Notice that a faulty machine can be partially-specified.

The specification is also assumed to be reduced. Each FSM has minimal form and it can be reduced to its minimal form. Note that partial FSMs may have several distinguishable reduced forms. Methods for constructing a reduced form of a partial machine are more involved than that for complete machines [Ko10]. Minimization of complete FSM has polynomial time complexity, however it can run even in $O(n \log n)$ [Ho71].

Methods creating a test suite T according to the specification can be divided by the number of maximal tests in T. A test suite with several maximal test cases needs the specification with reset feature but it comes with price of assuming the initially-connected specification. On the other hand, testing methods producing a checking sequence assume the strongly-connected specification, i.e. a more restricted class of FSMs than the one of initially-connected machines.

This chapter describes testing methods that we are aware of. We propose basic idea behind a given particular method and a detail design if the method takes a part in our experiments. Notice that some methods even deal with partial or nondeterministic machines. Methods assume that each test is completely performed, i.e. there is no state in which an input as a part of the test cannot be processed in the implementation under test. This is a simplification because if a test sequence gets stuck, we find an error in the implementation.

6.1 Resettable machines

A test suite of a FSM with reset feature usually consists of several input sequences. Each sequence is applied to the initial state that is fixed in the machine. The reset input r always brings the machine into the initial state. Note that there are two types of reset in the literature. We consider the *reliable* reset however if the reset was unreliable, i.e. its application does not have to bring the machine to the initial state, the reset functionality would have to be tested.

Testing methods confirm correctness of each transition; the machine really goes from state s_i to state s_j under input a if the specification holds such that. Hence transition cover set TC is employed in each proposed testing method because it contains shortest sequences connecting the initial state with all states, i.e. state cover set SC, extended by each symbol of the input alphabet X. That is, each transition is covered by at least one sequence of TC. The empty string ε is in TC as well. The methods differ in the way how they manage confirmation of the next state after processing a tested transition.

The TC is extended multiple times by the entire input alphabet if the maximum number of states m in the implementation is greater than the number of states n in the specification. The proposed methods consider the sets of sequences P and R as follows.

$$P = \bigcup_{i=0}^{m-n} TC \otimes X^i = \bigcup_{i=0}^{m-n+1} SC \otimes X^i \quad \text{and} \quad R = P \setminus \bigcup_{i=0}^{m-n} SC \otimes X^i$$

Notice that $P = TC$ and $R = TC \setminus SC$ if there is no extra state in the implementation. In addition, if the specification M is not completely specified, defined sequences are chosen only, i.e. P and R are redefined as $P \cap \Omega_M$ and $R \cap \Omega_M$.

Theorem 4.1 enables us to use arbitrary identification sequences for state verification. Therefore, we propose three testing methods that we have not found in the literature. The methods are the PDS-method, ADS-method and SVS-method. They employ input sequences (name of method) with stronger verification power than characterizing set. These sequences are preferably used for checking sequence design, see Section 6.2.

6.1.1 PDS-method

The PDS-method employs *preset distinguishing sequence* (Definition 3.27) for confirmation of the next state. Every state produces a different output sequence on distinguishing sequence and thus PDS distinguishes all states. The advantage of the PDS-method is that only one sequence is applied after each sequence of P, the preset distinguishing sequence in particular. However, there are finite-state machines without PDS which restrict the applicability of this method, see Section 3.6. A test suite T is formally described as follows:

$$T = P \otimes DS$$

6.1.2 ADS-method

The ADS-method is similar to the PDS-method but it uses *adaptive distinguishing sequence* (Definition 3.28) for verification of the next state. ADS is a set of input sequences d_i that distinguishes every state from each other. The ADS-method produces shorter test suites than the PDS-method and it is applicable to more machines because there are more more machines having ADS than PDS, see Section 3.6. However, not all machines have an ADS. A test suite T is formed as follows:

$$T = P \circ d_i$$

6.1.3 SVS-method

The SVS-method confirms transitions using *state verifying sequences* (Definition 3.29), aka *unique input output sequences*. SVS does not have such verifying power as distinguishing sequences so $SVSet = \bigcup_{s_i \in S} SVS_i$ has to be introduced. Each pair of different states has to be extended by the same separating sequence according to Theorem 4.1. Thus, SVSet is appended to each sequence of $P \setminus R$ and SVS_i of state s_i verifies the last transition, i.e. it appends to related sequence of R. In case of absence of extra states, the set $P \setminus R$ coincides with SC and R with the rest transitions. The SVS-method corresponds to the UIOv-method (Section 6.2.5) but it does not concatenate tests to a checking sequence so a test suite is created as follows:

$$T = (P \setminus R) \otimes SVSet \cup R \circ SVS_i$$

SVS does not have to exist for each state. In such case, we use a state characterizing set W_i in place of non-existent SVS_i.

6.1.4 W-method

The W-method was described by Vasilevskii in 1973 [Va73] and by Chow in 1978 [Ch78]. It simply takes characterizing set W and applies each sequence after all tested transitions. That is, a test suite T is concatenation of sets P and W:

$$T = P \otimes W$$

6.1.5 Wp-method

The partial W-method, aka the Wp-method introduced by Fujiwara et al. in 1991 [Fu91], reduces the total number of input symbols used for testing. It consists of two phases. The former phase distinguishes each state reached by a sequence of $P \setminus R$ from others using characterizing set W like in the SVS-method. That is, state cover set SC is concatenated with characterizing set W if there is no extra state. The latter phase confirms all remaining transitions; last transitions of sequences of R. Use of state characterizing set W_i of the final state s_i after applying a sequence from R is enough for confirmation. Nevertheless, each state characterizing set W_i must be subset of characterizing set W used in the first phase, i.e. $\forall s_i \in S : W_i \subseteq W$. Then a test suite T is formed as follows:

$$T = (P \setminus R) \otimes W \cup R \circ W_i$$

The Wp-method is proposed for nondeterministic FSMs in [Lu94] and it is basis for some testing methods constructing checking sequence, for instance [Ch03] or [Xi06]; the latter reformulates concatenation of test cases as Asymmetric TSP.

6.1.6 HSI-method

The harmonized state identifiers-method, aka the HSI-method proposed by Petrenko in 1991 [Pe91], is comparable to the Wp-method in the length of test suite [Do05e]. It is based on special property required on state characterizing sets, or state identifiers. A harmonized state identifier H_i, see Definition 3.31, is applied after a sequence of P whose final state is s_i. Thus, a test suite T is concatenation of P with related harmonized state characterizing set H_i:

$$T = P \circ H_i$$

The HSI-method is also stated for partial nondeterministic FSMs in [Lu95] where a constructive algorithm for obtaining a family of HSIs is included as well. A probabilistic approach is described in [Ya95]. It applies tests created by the HSI-method at random and estimates a rate of undetected errors in the implementation with a probability of the estimation. The estimation is similar to general concept of PAC-learning [Ke94].

6.1.7 FF-method

The Fault Function-based method produces a test suite in accordance to the chosen fault function, see Definition 4.5. It builds tests to remove uncertainty after suspicious transition. Suspicious transitions are those causing errors. Thus, each suspicious transition is extended by separating sequences distinguishing the correct next state from the others defined by the fault function Φ. If a separating sequence contains suspicious transition, its different output must be checked from the other states. More precise conditions are stated by method's authors Petrenko and Yevtushenko in [Pe92].

The FF-method uses harmonized state identifier in the subset of states for removal of uncertainty after suspicious transition. Therefore, the method is equal to the HSI-method if all faults may occur, i.e. the implementation N is from the fault domain $F_m(M)$ of the specification M.

6.1.8 H-method

The H-method proposed by Dorofeeva et al. in 2005 [Do05i] is similar to the HSI-method. It is based on the sufficient condition for complete test suite, see Theorem 4.1. Choosing separating sequences on the fly is the only difference from the HSI-method. The H-method extends P by sequences distinguishing each two different states reached by P. Formally, the construction of a test suite T is described in four steps:

1. $T = P$,
2. For each $u, v \in SC$ let $s_u = \delta^*(s_0, u)$ and $s_v = \delta^*(s_0, v)$, and if $s_u \neq s_v$ and T does not contain sequences distinguishing states s_u and s_v, i.e. $\neg\exists u \cdot w, v \cdot w \in T : \lambda^*(s_u, w) \neq \lambda^*(s_v, w)$, then add such $u \cdot w$ and $v \cdot w$ to T,
3. For each $u \in SC, v \in P \setminus SC$ let $s_u = \delta^*(s_0, u)$ and $s_v = \delta^*(s_0, v)$, and if $s_u \neq s_v$ and T does not contain sequences distinguishing states s_u and s_v, i.e. $\neg\exists u \cdot w, v \cdot w \in T : \lambda^*(s_u, w) \neq \lambda^*(s_v, w)$, then add such $u \cdot w$ and $v \cdot w$ to T,
4. $(m > n)$ For each $u, v \in P \setminus SC$ let $s_u = \delta^*(s_0, u)$ and $s_v = \delta^*(s_0, v)$, and if $s_u \neq s_v$, $u \in \text{pref}(v)$ and T does not contain sequences distinguishing states s_u and s_v, i.e. $\neg\exists u \cdot w, v \cdot w \in T : \lambda^*(s_u, w) \neq \lambda^*(s_v, w)$, then add such $u \cdot w$ and $v \cdot w$ to T.

The algorithm is also generalized for nondeterministic FSMs in [Do05i].

6.1.9 SC-method

The State-Counting method proposed by Petrenko and Yevtushenko in 2005 [Pe05] designs m-complete test suite for unreduced partial deterministic FSM. Partial machines can have several minimal forms because there can be indistinguishable states. The method is based on the notion of *distinguishing machine*, i.e. composition of two machines, the specification and the implementation in particular. Then a test suite T containing all defined sequences of length up to mn is m-complete. Such a test suite T can be reduced by counting particular states traversed by a test. Distinguishable and quasi-equivalent states (Definition 4.2) help recognize state's occurrence in a test case.

6.1.10 P-method

The P-method generates a p-complete test suite, $p \leq n$. An algorithm with a proof of correctness was proposed by Simão and Petrenko in 2009 [Si09f]. It uses Theorem 4.3 as follows. A test suite T is p-complete if: $p < n$ and T contains a F_T-divergent set with $p + 1$ tests, or $p = n$ and T contains a F_T-convergence-preserving initialized transition cover for M.

The method gradually extends initial test suite until the condition. It keeps F_T-convergent and F_T-divergent pairs of tests and constructs a divergence graph captured F_T-divergent relations of tests. An F_T-divergent set corresponds to a clique in a divergence graph. To fulfill the condition a maximal clique is searched. Finding a maximal clique in a graph is NP-complete however suboptimal solution can be used as it is discussed in [Si09f]. If $p = n$, an additional procedure is needed to create a F_T-convergence-preserving initialized transition cover. Such a procedure is similar to aforementioned methods but instead of state cover SC the P-method handles tests in found maximal clique of size n. Each transition has to be verified so its final state is distinguished against each different state reached by a test of the maximal clique. The choice of separating sequences is not restricted. They can be chosen in advance as in the HSI-method or on-the-fly as in the H-method.

An extensive comparison with other methods is in [En13], for example. There is shown that the P-method generates the shortest test suite among the W-, H-, HSI-, SPY- and P- methods. However, the implementations are not included so we cannot use the results. Moreover, the P-method is not generalized to create m-complete test suites therefore it is not consider in the experiments.

6.1.11 SPY-method

The SPY-method employs convergence relation (Definition 4.8) to reduce test branching and thus the length of test suite T. It was stated by Simão, Petrenko and Yevtushenko in 2012 [Si12]. Test branching means that a nonempty sequence is extended by several sequences so several maximal tests are created, e.g. the characterizing set W has to be applied in a state s so at least $|W|$ maximal tests will contain the transfer sequence u, $s = \delta^*(s_0, u)$. It was proved that transitions verified so far extend sets of convergent sequences. A verified transition (s, x) implies that the transfer sequence $v \in SC$, $\delta(s, x) = \delta^*(s_0, v)$, converges with ux, where $s = \delta^*(s_0, u)$. Then a sequence u of convergent ones is chosen for verification of an unverified transition (s, x), $s = \delta^*(s_0, u)$, so test branching can be minimized.

The method uses harmonized state identifiers H_i for distinguishing states like the HSI-method. It starts with initialized state cover SC extended by H_i. Such a set is F_T-divergence-preserving. The method verifies iteratively each transition so the test suite T becomes F_T-convergence-preserving set that contains initialized transition cover. This ensures completeness of the test suite, see Theorem 4.3.

The partition induced by the F_T-convergent tests is denoted by Π and the block $[t]_\Pi$ of the partition Π contains tests $t' \in T$ convergent with $t \in T$. This notation is extended to a set of tests $K \subseteq T$ and thus $[K]_\Pi$ denotes the union of the blocks $[t]_\Pi$ over all tests $t \in K$. Two convergent sequences extended by the same sequence form convergent sequence, i.e. $[u] = [v]$ implies $[uw] = [vw]$. Simultaneously with adding a new test to T, the partition Π increases its size by new classes that correspond to the added test cases. Some of new classes are convergent so they are merged. Let closure(Π) denote the partition obtained after merging each convergent sequence classes.

We need to choose which convergent sequences will be extended by particular distinguishing sequence w. The choice influences test branching so here is the rule. Given class of convergent sequences $[u]_\Pi$, let $t_i = u_i w \notin T$, $u_i \in [u]_\Pi$, be the tests so that one of them should be added to T. Select a t_i such that there is a maximal test $t' \in T$ that is prefix of the t_i, then $\text{len}(T \cup \{t_i\}) = \text{len}(T) + |t_i| - |t'|$ is minimal. If there are more t_i with a maximal test as prefix, choose the one that increases the length of test suite least. On the contrary, if such a maximal test t' does not exist for each t_i, select the $u_i \in SC$ for extension. The length of T then grows by $|t_i| + 1$ which is minimal because the $u_i \in SC$ is shortest transfer sequence to particular state. The algorithm follows:

1. $T = SC \circ H_i$, $\Pi = \{\{u\} \mid u \in T\}$,
2. For each transition (s, x) not covered $[SC]_\Pi$ and each $e \in X^{m-n}$ do:
   (i) Let $u, v \in SC$ be such that $\delta^*(s_0, u) = s$ and $\delta^*(s_0, v) = \delta(s, x)$.
   (ii) For each $w \in H_i$, $s_i = \delta^*(s_0, ve)$
      1) Select $u' \in [u]_\Pi$ such that $\text{len}(T \cup \{u'xew\})$ is minimal and add $u'xew$ to T.
      2) Select $v' \in [v]_\Pi$ such that $\text{len}(T \cup \{v'ew\})$ is minimal and add $v'ew$ to T.
      3) Update Π by closure($\Pi \cup \{\{t\} \mid t \in \{u'x, v'\} \otimes \text{pref}(ew)\}$)
   (iii) Update Π by closure($\Pi \cup \{[ux]_\Pi \cup [v]_\Pi\}$)

6.2 Machines without reset

Testing methods creating checking sequence do not consider reset feature because the machine under test does not possess it or the application of reset is very costly. Another assumption thus usually restricts the tested system. The specification needs to be strongly connected to be possible to verify each transition.

The methods we considered here assume that the implementation is in the given initial state before a checking sequence is applied. However, a homing sequence can precede the checking experiment to ensure the initial state. Most methods create test cases at first and then compose these sequences to produce a checking sequence. Fault domain contains all deterministic machines with up to n states so the methods focus on n-complete test suite. Thus, the methods assume a deterministic specification and usually a complete-specified one.

6.2.1 HrADS-method

The method proposed in [Hs71] generates m-complete test suite contained one maximal test. It was originally called the State Counting method but we call it the HrADS-method because of its design. Test cases are created as in the ADS-method (Section 6.1.2) and each test is prepended by a homing sequence that ends in the given initial state of the machine. A checking sequence is then formed to contain all tests.

The paper by Hsieh [Hs71] from 1971 uses different notions however in fact it follows Theorem 4.1 with use of homing sequence. Therefore, the HrADS-method deals with extra states in the implementation. The name of method is derived from use of *H*oming sequence as *r*eset and *A*daptive *D*istinguishing *S*equence for state verification. It is remarkable that the paper contains a lot of new notions however neither the notions nor the method were further developed.

6.2.2 D-method

The D-method is the most researched testing method. It uses a preset distinguishing sequence for state verification so it is applicable only on machines that possess PDS d. Extra states in the implementation are not considered hence the method produces an n-complete checking sequence.

Basis was proposed by Hennie [He64] in 1964. The method consists of two parts: checking a response to d in each state and checking each transition. States are sorted in advance and in this order d is applied to each state in the first phase. Transfer sequences are employed. A transfer sequence t_{ij} concatenates $\delta^*(s_i, d)$ and s_j if s_j follows s_i in the given order. The second phase checks each transition. A transition (s_j, x) is verified by a CS's subsequence $d \cdot t_{ij} \cdot x \cdot d$ applied in state s_i. This approach sets upper bound on the length of the produced checking sequence.

In 1970, Gönenc [Go70] introduced notions of *d-*, *q-* and *t-recognized* states in the checking sequence. These notions are used to prove correctness of the method in [Ur97], i.e. the method really generates a checking sequence. Gönenc named the sequences generated by both phases. The first phase produces an α-sequence and the second phase a β-sequence. There are approaches how to construct such sequences. In short, d is gradually appended until there is a state which is not recognized by $d \cdot d$. Transfer sequences are used only when the constructed sequence c contains $d \cdot d$ applied in the state reached by c, i.e. $c = uddv$ and $\delta^*(s_0, u) = \delta^*(s_0, c)$. The β-sequence is a concatenation of so-called cells and transfer sequences when they are needed. A cell of the transition (s, x) is a sequence $x \cdot d$ applied in the state s. Not all cells need to be included in the β-sequence because some transitions are verified in the α-sequence.

Generalization of both aforementioned approaches and proof of correctness were proposed by Ural et al. in 1997 [Ur97]. The α-sequence and the β-sequence are divided into sets of sequences α-set and C-set respectively. The elements of sets are then captured as edges in an auxiliary graph G' that contains each state of the given machine twice. A checking sequence has to include all elements of α-set and C-set so Rural Chinese Postman Problem (RCPP) covering the sets is solved on G'. A RCPP finds a path over a subset of edges with minimal cost [Ah91]. A RCPP is known to be NP-complete in general. Fortunately, the given task can be cast as Minimum-Cost Flow (MCF) which is polynomially solvable [Ur97]. The contribution is thus interleaving of both phases.

Hierons and Ural further reduced the length of produced checking sequence in 2002 [Hi02]. They specified the design of α-set so an α'-set is created and they also proved that some edges in G' are not needed. In 2006, they described how to obtain constructively the α'-set [Hi06]. A transition can be verified by appending a sequence of α'-set so the related test segment, i.e. a cell, of C-set does not have to be included in the CS. This was proposed formally by Chen et al. in 2005 [Ch05] and by Ural and Zhang in 2006 [Ur06]. The latter paper also introduced overlapping of sequences of α'-set and C-set so the auxiliary graph G' contains edges with negative cost representing the length of the overlapped subsequence. The last improvement of the D-method we are aware of is based on a notion of invertible sequences [Hi96, Hi97]. In 2009, Duan and Chen [Du09] showed an enhancement with alternative β-sequences using acyclic property related to invertible sequences. However, Kapus-Kolar [Ka12e] showed three years later that the method is unsafe due to the cyclic dependencies. She proposed a modification of the method in 2014 [Ka14].

6.2.3 AD-method

Boute showed in 1974 that an Adaptive Distinguishing Sequence can be used instead of PDS in the D-method [Bo74]. We call the D-method using ADS the AD-method. In 2009, Hierons et al. [Hi09] proved that even recent versions of the D-method can employ ADS in place of PDS without loss of test suite completeness. The AD-method is more applicable than the D-method because more FSMs have got an ADS and moreover there is a polynomial (suboptimal) design algorithm for ADS.

6.2.4 DW-methods

There are machines without a distinguishing sequence. Therefore, Hennie [He64] proposed with the D-method an extension to wider class of machines, i.e. all deterministic FSMs. It employs characterizing set W in place of preset distinguishing sequence d. We call this method the DW-method. Each sequence w_i of W needs to be applied in the same state that is checked currently. It was shown that it is sufficient to repeat a sequence $(n+1)$-times to ensure that the final and the start states are equal. A locating sequence l is introduced. For $W = \{w_1, w_2\}$ it is created as $l = (w_1 \cdot t_{ij})^{n+1} \cdot w_2$, where a transfer sequence t_{ij} brings the machine from the state $s_i = \delta^*(s_j, w_1)$ to the state s_j when l is applied in s_j. Notice that the length of checking sequence created in this way is exponential in the number of states.

The DW-method was described in detail with the recursive definition of a locating sequence and casting to Rural Chinese Postman Problem by Rezaki and Ural in 1995 [Re95]. Use of state characterizing sets was formulated by Porto et al. in 2013 [Po13]. It is similar as development the Wp-method from the W-method so we called this improved approach the DWp-method.

6.2.5 UIO-methods

Use state verifying sequences (Definition 3.29) for state identification has got a lot of focus of researchers because more machines have SVSs than distinguishing sequences.

The UIO-method was proposed by Sabnani and Dahbura in 1985 [Sa85, Sa88]. It uses reset feature so this method could be in Section 6.1 but the original paper forms a checking sequence CS from smaller tests that each verifies a transition and reset r is also verified as a standard input. The UIO-method is formally described as follows:

$$T = TC \circ SVS_i \cup SC \otimes \{r \cdot SVS_0\} \longrightarrow CS = \bigodot_{t_j \in T} r \cdot t_j = r \cdot t_1 \cdot r \cdot t_2 \cdot \ldots \cdot r \cdot t_{|T|}$$

Moreover, a signature was introduced when a state does not posses a SVS. A signature is a concatenation of separating sequences interleaving with transfer sequences.

Unfortunately, in 1988, Sidhu and Leung [Si88] showed that the UIO-method has not a complete fault coverage. Therefore, the UIOv-method was proposed one year later by Chan et al. [Ch89]. The UIOv-method coincides with the SVS-method (Section 6.1.3) however it generates only n-complete test suite. The method was improved using an exclusive tree for generation of SVSs in 1995 [Ch95i] but a complete fault coverage was not proven.

Aho et al. casted the UIO-method to Rural Chinese Postman Problem (RCPP) in 1988 [Ah91]. The reset r is treated as a standard input, i.e. each state has a transition leading to the initial state s_0 on r. A test segment of a transition (s, x) is $x \cdot SVS_{\delta(s,x)}$ applied in s. The state diagram is augmented by all test segments and RCPP over every test segment is solved on this augmented graph.

The MUIO-method was proposed by Shen et al. in 1992 [Sh92]. If a state possesses more SVSs, the method chooses the one minimizes the length of experiment. Nevertheless, a completeness of checking experiment was again not proven. The MUIO-method was casted as Traveling Salesman Problem that was solved by Hopfield Neural Networks [Ku94], and was a basis for a distributed testing method using synchronizable test sequences [Ch95s, Ch06], for example.

Overlapping and invertible sequences can be used to further reduce the length of experiment as the paper by Duan and Chen [Du07] shows. However, even there is not a proof that the method creates a checking sequence.

6.2.6 CSP-method

The CSP-method is rather an approach how a checking sequence, or a test suite, could be created than a constructive algorithm. It was proposed by Vuong and Ko in 1990 [Vu90] and it is motivated by Constraint Satisfaction Problem. If one has got a checking sequence CS, the CSP-method produces all machines with up to n states having the same response to CS as the specification. Then additional test should be created to distinguish the faulty machines from the specification. The main idea is to tie the final state of a test with a domain of all possible states that can be reached by the test case in a machine accepting the CS. Domains are reduced if tests are distinguished in the given CS, e.g. there are tests $t_i, t_j, t_i \cdot w, t_j \cdot w \in CS$ and $\lambda(\delta(s_0, t_i), w) \neq \lambda(\delta(s_0, t_j), w)$ then the final states of t_i and t_j are different.

A fault coverage analysis method based on the CSP-method is reformulated in [Zh93] and our improved algorithm is in Chapter 9. Improving of fault coverage of the UIO-method is proposed in [Wa10]. It employs the same idea as the CSP-method, i.e. it reduces domains of states reached by tests and adds new tests to make domains singletons.

6.2.7 C-method

The C-method gradually extends a confirmed set C by appending ADS until each transition is verified so the sufficient condition in Theorem 4.2 holds. The method was proposed by Simão and Petrenko first in 2008 [Si08] and with improvement by output-confirmed sequences in 2009 [Si09c]. Partial deterministic machines are considered and a reduction using reliable reset (if available) is discussed as well in [Si08]. The method was not named by the authors so we call it the C-method because it is based on a notion of confirmed set of sequences.

Let t be a (checking) sequence. A confirmed set $C(t)$ over t contains all prefixes of t that can be added to C by repeatedly applying the following three rules. Note that notions of output-confirmed and convergence of tests are defined as well as confirmed set in Section 4.3. As a reminder, d_s is fixed adaptive distinguishing sequence of state s.

1. If u is extended (in t) by d_s, where $s = \delta^*(s_0, u)$, then add u to $C(t)$.
2. If there exists $u \in C(t)$ and v is verified in $\delta^*(s_0, u)$, then add $u \cdot v$ to $C(t)$.
3. Given a transfer sequence u to state s, if for each $q \in S \setminus \{s\}$ there exists an q-output-confirmed sequence v distinguishing s and q, such that v extends u or a sequence which is F_t-convergent with u, then add u to $C(t)$.

The method then can be described in the following steps. The test sequence t_i is empty at the beginning, i.e. $t_0 \leftarrow \varepsilon$, and the confirmed set $C(t_i)$ is updated as soon as t_i is extended.

1. If $t_i \notin C(t_i)$ then
   (i) Let u be the shortest prefix of t_i such that $u \notin C(t_i)$, $t_i = u \cdot v$, $d_{\delta^*(s_0,u)} = v \cdot w$
   (ii) Update $t_{i+1} \leftarrow t_i \cdot w$
2. Otherwise, i.e. $t_i \in C(t_i)$
   (i) Determine a shortest verified transfer sequence u from state $\delta^*(s_0, t_i)$ to some state s, such that there exists $x \in X$ and (s, x) is unverified. Let $s' = \delta(s, x)$.
   (ii) If x is a prefix of d_s, then let v be such that $d_s = x \cdot v$. Determine the shortest prefix w of $d_{s'}$, such that, for each $q \in S \setminus \{s'\}$ which is not distinguishable from s' by any q-output-confirmed prefix of v, there exists a common prefix of w and d_q which distinguishes s' and q. Otherwise, i.e., if x is not a prefix of d_s, let $w = d_{s'}$.
   (iii) Update $t_{i+1} \leftarrow t_i \cdot u \cdot x \cdot w$
3. If there is an unverified transition, update $i \leftarrow i + 1$ and go to step 1.

Although the C-method uses local optimization by overlapping distinguishing sequences, it leads to shorter checking sequences than the D-method produces. The D-method optimizes concatenation of preset test sequences globally.

6.2.8 K-method

The most promising method for creating shortest checking sequences was proposed by Kapus-Kolar in 2010 [Ka10]. We call it the K-method by the author's name. The method is based on so-called State-Recognition Patterns (SRP) [Ka12n] that employ detail logical reasoning to reduce domains of reached states. It is similar to the CSP-method however the reasoning is much deeper. Moreover, the K-method advices how to extend test segments to gradually verify all transitions. Therefore, the length of created checking sequence is shorter than by the D-method which chooses test segments in advance and then composes them.

Chapter 7
Summary of Methods

Testing methods were developed a lot in the last few decades. There are plenty of approaches for verification that the implementation coincides with the specification. We tried to describe the methods that we are aware of in the previous chapter. This chapter groups these methods and shows their relations.

The methods are sorted in chronological order and are shown in Figure 7.1. As in the Chapter 6, we group the methods by the cardinality of a test suite they produce. That is, the W-, Wp-, HSI-, FF-, H-, SC-, P- and SPY- methods form one group because they create a test suite with several sequences so a resettable machine is needed. The second group contains the D-, DW-, HrADS-, UIO-, UIOv-, CSP-, MUIO-, AD-, C-, DWp- and K- methods that create a checking sequence, i.e. a test suite of size one.

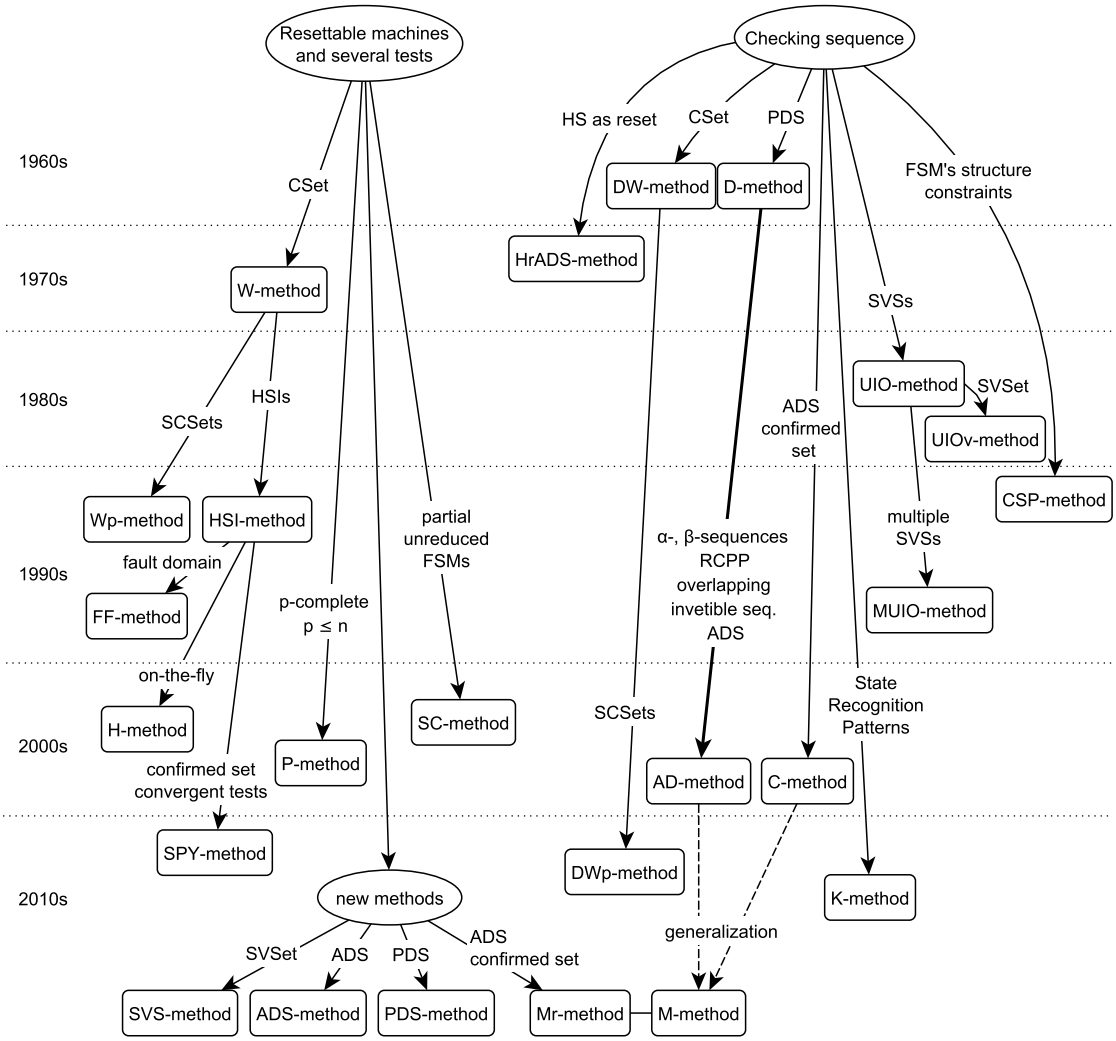


Figure 7.1. History of Testing Methods

An edge to a method in Figure 7.1 is labeled by a characteristic factor of the method, e.g. type of sequence used for verification. Some methods are based on others. In such case, an edge connect two related methods; it goes from the basic method to the derived method. For example, take the D-method and its derived AD-method. The D-method was very researched so a lot of improvements was invented. Adaptive distinguishing sequence then replaced preset distinguishing sequence and the AD-method was introduced. Notice that the corresponding edge is emphasized due to the amount of work spent on the D-method.

In this thesis we propose several new methods. They are shown with the classical ones in Figure 7.1. The SVS-, ADS- and PDS- methods are proposed in the previous chapter and the M-method with its modification using reset, the Mr-method, are introduced in the following chapter.

Chapter 8
M-method

The *M-method* is introduced as a generalization of the C-method (Section 6.2.7) and the AD-method (Section 6.2.3) in this chapter. All the testing methods so far were based on a simple testing scheme. The implementation is transfered to an identified state, an unverified transition is processed and a verification sequence (or a set of sequences) is appended. The shift to an identified state is essential for the testing methods.

For a machine with reset ability, it is easy to transfer the machine reliably to the same state multiple times. Transitions are then verified in the order of reaching them from the initial state. If one wanted to strictly follow the testing scheme, the set of tests, i.e. a transition cover extended by verification sequences, would be ordered so that transitions from the initial state would be verified first, then transitions from a state reached by a verified transition would be processed and so on. However, the testing methods for resettable machines do not require such a restriction of tests' order. One can start with verification of the furthermost transition, e.g. the longest test is applied first, even though a preceding transition has not been verified. We know that the implementation is correctly implemented, i.e. all transitions are verified, after the entire test suite was executed. There are some assumptions that must hold, see Chapter 4, but we will not mention them because the focus is now on motivation for a new approach.

Testing methods generating a checking sequence have to do more work to transfer the machine reliably to the same state because reset is not usually available. There are two main approaches. One is based on verification of distinguishing sequence and the second on the notion of confirmed set. The D-method and its derivations including the AD-method require that each state is uniquely identified in the experiment and the distinguishing sequence (DS) used for identification is also a distinguishing sequence of the implementation. Then the use of DS enables to identify a state. On the other hand, the C-method employs a confirmed set of sequences to identify a state.

The former approach can be optimized globally because test subsequences are created in advance. These preset sequences are limitations of the approach. It can perform better only if some of these sequences are eliminated. Only local optimization is currently possible in the latter approach due to the design of the C-method. As we show in Section 8.1.3, the limitation of this method is just the greedy choice of unverified transition that are to be checked. We asked ourselves whether is it possible to somehow remove limitations of both approaches.

8.1 Checking sequence

Standard technique to handle a difficult problem is relaxation of a constraint. We decided to remove the requirement of identified state before checking an unverified transition. We think that if each transition is verified by means of appending it by a distinguishing sequence, all states will be identified retrospectively. It is similar to the case of resettable machines, if there is a fault in the implementation and a test containing the fault is processed successfully, another test displays it. In other words, assume a

transition error in the implementation. After passing this transition, an unverified transition outgoing from the reached state is to be checked. The implementation responses to the distinguishing sequence correctly hence the fault is not detected. However, notice that the state is uniquely identified due to application of the distinguishing sequence. We know there is a subsequence of the given checking sequence that should started with the error transition followed by the distinguishing sequence. This is the point where the fault shall appear.

The described notion enables us to relax the basic testing scheme and to formulate the M-method. The M-method simply says a distinguishing sequence is a prefix of checking sequence so the initial state is identified, and that there is a verifying subsequence for each transition in checking sequence.

The basic version of the M-method employs entire adaptive distinguishing sequence for transition verification. Checking sequence then has to contain each sequence of a set U as its subsequence. The set U is defined as follows:

$$U = \{(s_0, d_0)\} \cup \bigcup_{s_i \in S} \{(s_i, x \cdot d_j) \mid x \in X, s_j = \delta(s_i, x)\}$$

where d_i is a part of adaptive distinguishing sequence that separates state s_i.

The optimization problem of minimizing length of a sequence that has to include a set of sequences can be cast as Asymmetric Traveling Salesman Problem (ATSP). Traveling salesman problem (TSP) finds minimal-cost tour over all nodes in a weighted directed graph. Asymmetric TSP (ATSP) signifies that edges between two nodes have different costs. TSP as well as ATSP are known to be NP-complete problems.

Nodes are elements of U in case of creating checking sequence. An edge $(u_i, u_j) = ((s_i, t_i), (s_j, t_j))$ indicates that the sequence t_i of u_i appears before the sequence t_j of u_j. There are two possible connections of t_j after t_i. The 'after' is not strict in this case so the sequences can overlap. Formally, $(s_i, t_i) \in U$ *overlaps* $(s_j, t_j) \in U$ by a non-empty sequence v if there are a prefix v_i of t_i and a suffix v_j of t_j such that $t_i = v_i \cdot v$, $t_j = v \cdot v_j$ and $s_j = \delta^*(s_i, v_i)$. The first connection is that u_i overlaps u_j and the second one is that a shortest sequence w_{kj} connecting the final state s_k of t_i with s_j is appended before t_j. The shortest connecting sequence is the empty sequence if the final state of u_i is equal to s_j. The cost of an edge (u_i, u_j) corresponds to the minimal length of sequence v from s_i to s_j such that t_i is a prefix of $v \cdot t_j$. There is one exception. Edges to the node $u_0 = (s_0, d_0)$ cost the length of sequence of the starting node because we look for a path from u_0 so the last visited node appends only its sequence to the created checking sequence. The cost of a tour over all nodes is the length of related checking sequence.

The costs of edges are the most important part of task. They can be written in a matrix $C_{I \times I}$, where I are indexes of elements of U, i.e. $I = \{0, \ldots, np\}$. An entry c_{ij} thus corresponds to the cost of edge (u_i, u_j). Entries of C are defined as follows:

$$c_{ij} = \begin{cases} \infty & \text{if } i = j, \\ |t_i| & \text{if } j = 0, \\ |v_i| & \text{if } u_i \text{ overlaps } u_j \text{ by a } v \text{ and } t_i = v_i \cdot v, \\ |t_i| + |w_{kj}| & \text{otherwise.} \end{cases}$$

where w_{kj} is the shortest sequence from the t_i's final state $s_k = \delta^*(s_i, t_i)$ to the state s_j. If s_k and s_j coincide, i.e. $k = j$, the shortest sequence w_{kj} is the empty string ε hence the related cost reduces to $|t_i|$. Notice that self-loops are not considered from a logical reason so costs c_{ii} are infinity (or a reasonably big value).

We are now able to pose an Integer Linear Programming (ILP) formulation for the M-method.

$$\min \quad \sum_{i=0}^{np}\sum_{j=0}^{np} c_{ij}x_{ij}$$

$$\begin{aligned}
\text{s.t.} \quad & \sum_{i=0}^{np} x_{ij} = 1 && j \in I && \text{(enter once)}\\
& \sum_{j=0}^{np} x_{ij} = 1 && i \in I && \text{(leave once)}\\
& o_i - B(1 - x_{ij}) < o_j && i \in I,\ j \in I \setminus \{0\} && \text{(cycle indivisibility)}\\
& x_{ij} \in \{0,1\} && i,j \in I\\
& o_i \in \mathbb{R}_0^+ && i \in I
\end{aligned}$$

It is one of standard formulation of TSP. A binary variable x_{ij} is 1 if and only if the edge (u_i, u_j) is in the found tour. Checking sequence must contain each test segment therefore each node u_j has to be entered once. From each reached node u_i, only one leaving edge is chosen to be in the tour. Another constraint is needed due to the fact that the previous two constraints enable to create several tours over subsets of nodes. A time ordering variables o_i are introduced to force creating of one tour, and checking sequence as well. o_i is assigned to each node and indicates time of visit the related node u_i by the created tour. The variable o_0 of the initial node u_0 has got the lowest value of all o_i's. The node visited right after the initial one has got the second lowest value and so on. The constraint 'cycle indivisibility' captures such ordering. We use a constant B set to infinity (or another reasonably big value) to turn on and off the constraint according to x_{ij}. If an edge (u_i, u_j) is chosen, i.e. $x_{ij} = 1$, the constraint reduces to $o_i < o_j$. Otherwise, the constraint holds because of great value of B.

8.1.1 Idea of proof

We demonstrate that the M-method creates a checking sequence experimentally, see Chapter 11. However, a formal proof is still missing. We propose our idea how a proof could be made in this section.

It is clear that if a proof was found, the D- and AD- methods would be proven as well. They contain all test segments that the M-method requires. Notice that we join the D- and AD- methods because we know that preset and adaptive distinguishing sequences have got the same power of unique state identification. An adaptive distinguishing sequence as the shorter one is thus considered if we use the notion of distinguishing sequence. Test segments, or cells, form β-sequences in the D-method, see Section 6.2.2.

The C-method without output-confirmed sequences improvement, i.e. its first proposal in [Si08], also contains all test segments in fact. It can be proven employing the first two rules for addition to a confirmed set, see the C-method in Section 6.2.7. A transition (s, x) cannot be verified unless a sequence to s and its extension by x, the sequence to $\delta(s, x)$, are in the confirmed set. The latter sequence can be added to the confirmed set only by the first rule so the checking sequence contain the related test segment. An improvement by output-confirmed sequences is discussed in the following section.

A distinguishing sequence is applied in each state so each state is uniquely identified in the experiment. Figure 8.1 displays such a checking sequence on a machine with

states A, B and C. The checking sequence starts in the state A which is the initial state of the machine. Distinguishing sequence d_A takes the machine into an unidentified state denoted by a question mark. Then an input sequence (a downward dotted edge) is applied followed by distinguishing sequence d_C. The state where d_C was used is recognized as different from the state A so we can label it C. After d_C, an input sequence is appended again and A is reached. It cannot be claimed in this phase that the state is really A unless d_A follows and the last state B is also identified, i.e. d_B is employed. Of course, we must know that there are only three states in the implementation. d_A is thus added and then d_B as well. Input and distinguishing sequences are appended in the same manner till each test segment is included in the created checking sequence.

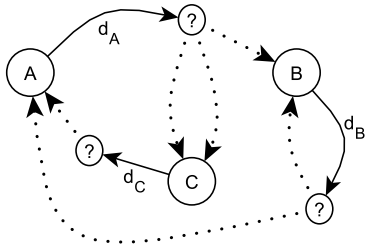


Figure 8.1. A Checking Sequence of a 3-state Machine

A shortest checking sequence is desired. Therefore, we can assume that distinguishing sequence is used as soon as possible. Thus, some input sequences (dotted edges) in Figure 8.1 are shortened. Moreover, distinguishing sequences can be found to overlap. Figure 8.2 shows more realistic model of a checking sequence generated by the M-method. Notice that a state reached by d_A is merged with B which enables to locate C inside d_B. Similarly, the state A is situated within d_C due to overlapping of d_C and d_A.

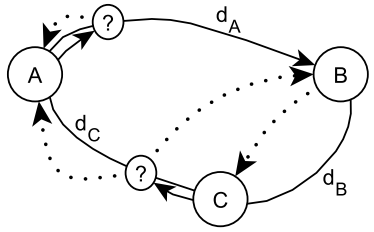


Figure 8.2. An Optimal Checking Sequence by the M-method

An interesting thing is shown in Figure 8.2. Distinguishing sequences follow (or overlap) each other. The D-method introduces α-sequences that verify distinguishing sequence in the implementation. A concatenation of two distinguishing sequences, i.e. dd, has to be included for each state in the experiment. Nevertheless, distinguishing sequence does not only identify a state which it is applied in but it also identifies each state along its path uniquely. Therefore, every distinguishing sequence is a homing sequence. Identification of a state inside distinguishing sequence is comparably important.

The proposed notions are neither sufficient nor constructive. We thus employ Theorem 4.2 that states a sufficient condition for n-completeness of a test suite using confirmed set. A transition (s, x) is verified if a sequence v leading to s and a sequence $v \cdot x$ are in confirmed set. Note that the mentioned sequences are prefixes of checking sequence. Sequences $v \cdot x$ are in confirmed set as they are appended by distinguishing sequence. Therefore, if it is shown that for each transition such a sequence v can be add to confirmed set, the M-method will be proven to generate n-complete test suite.

Simão and Petrenko stated the following sufficient condition for the existence of a confirmed set in [Si10].

Theorem 8.1. Let T be a test suite of FSM M with n states and $L \subseteq T$ be a set of k sequences, $k \geq n$. For an arbitrary ordering of the sequences $t_1, \ldots, t_k \in L$, let $L_i = \{t_j \in L \mid 1 \leq j \leq i\}$. Then L is a confirmed set if there exists an ordering $t_1, \ldots, t_k$, such that the corresponding $L_1, \ldots, L_k$ satisfy the following conditions:

1. L_n is a minimal state cover such that every two sequences are T-distinguishable.
2. If $k > n$, then for each t_i, $n < i \leq k$, it holds that either
   a) for each $s \in S \setminus \{\delta^*(s_0, t_i)\}$, there exists $u \in L_{i-1}, \delta^*(s_0, u) = s$, such that t_i and u are T-distinguishable; or
   b) there exists u, v and w, such that $t_i = uw$, and $vw, v, u \in L_{i-1}$, $\delta^*(s_0, v) = \delta^*(s_0, u)$.

Two sequences $t_i, t_j \in T$ are T-distinguishable if the states they reach, s_i and s_j respectively, are distinguishable by a set U of sequences that extend t_i and t_j in T, i.e. $s_i \nsim_U s_j$ and $U = \{u \mid t_i u, t_j u \in T\}$.

We are able to create L_{np+1} that includes the empty sequence and all sequences appended by a distinguishing sequence, i.e. the last input is a transition for verification. The first condition of Theorem 8.1 thus holds. Some states at the end of distinguishing sequences are identified due to minimization of the length of constructed sequence. The condition 2b) is employed and confirmed set can be extended. Notice that Theorem 8.1 can be handled as an algorithm for design of confirmed set. If one has got a confirmed set L_k with k sequences, $k > n$, and a sequence t that satisfies the second condition, a set $L_{k+1} = L \cup \{t\}$ becomes confirmed. The problem we deal with is how to prove that application of the second condition several times on L_{np+1} is sufficient to create desired confirmed set, i.e. each transition is verified.

Another point of view we had, is based on ordering of transitions' verifications. Sets L_i indirectly define such an order. Similarly to the SPY-method (Section 6.1.11), a transition is verified so the next state can be used as a source of unverified transition. That is, if a transition is to be verified, each preceding transition from an identified state (the initial state for the SPY-method) has to be verified first. It implies that the start state of the unverified transition is identified as well. There are several identified states in the sequence generated by the M-method. Hence, it could be easier to find such an order of transitions to verification. Unfortunately, there are two issues: the choice of the first transition and cyclic dependency.

At the beginning, each identified state is followed by related distinguishing sequence in the experiment. A transition (s, x) is verified if there is ud_s and $uxd_{s'}$ in the created sequence, where $s = \delta^*(s_0, u)$ and $s' = \delta(s, x)$. However, this requires d_s to overlap $d_{s'}$ except one input symbol which is unlikely when a distinguishing sequence is not short. A more promising way to find the first verified transition is employing the condition 2b) of Theorem 8.1. Let u, u' be different sequences reaching the same identified state q, i.e. ud_q and $u'd_q$ are in the test suite T. If T also contains sequences uvd_s and $u'vxd_{s'}$ and $s = \delta^*(q, v), s' = \delta^*(q, vx)$, the transition (s, x) is verified. How can one be sure that such sequences are always present so a first transition is verified?

Let assume that the first transition is verified and consequently a few next transitions are checked as well. Unfortunately, then there is a possibility of cyclic dependency of transitions. An unverified transition $e_i = (s_i, x_i)$ needs another transition $e_j = (s_j, x_j)$ to be verified before it but e_j requires e_i to be verified first. This seems not to be possible as the length of checking sequence is minimized. Test segment for a transition

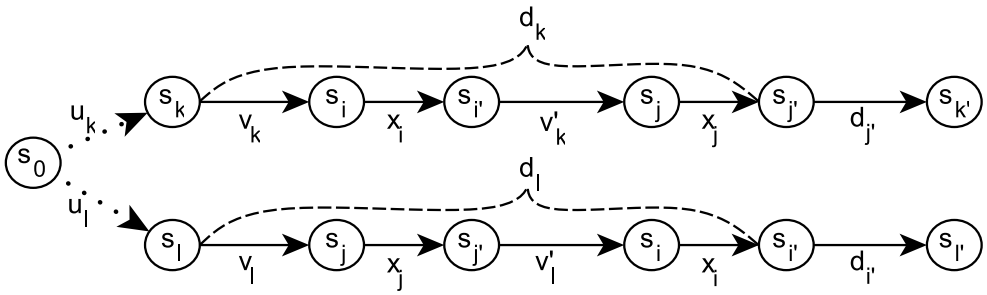


Figure 8.3. Cyclic Dependency in Verification of Transitions in the M-method

should be used as soon as possible. Nevertheless, one or both transitions e_i and e_j can be inside a test segment of other transition and the suffix of such a test segment does not have to overlap the distinguishing sequence related to x_i (or x_j). An example of such a case is shown in Figure 8.3. There are two prefixes of formed checking sequence: $u_k d_k d_{j'}$ and $u_l d_l d_{i'}$. Distinguishing sequence d_k contains transition $e_i = (s_i, x_i)$ and its last transition is $e_j = (s_j, x_j)$. Distinguishing sequence d_l contains transition e_j and its last transition is e_i. Assuming that suffix of d_k (d_l) after e_i (e_j) does not overlap the related distinguishing sequence $d_{i'}$ ($d_{j'}$) and no state is identified in d_k (d_l) leads to that a compliant ordering for Theorem 8.1 cannot be derived from these two discussed sequences. How to show that such cases are resolved in the sequence generated by the M-method (if they really occur), it is still not clear.

8.1.2 Improvements

We discuss two possible improvements of the M-method in this section. The first one is the choice of adaptive distinguishing sequence and the second is reduction of test segments in their length.

The more test segments overlap, the shorter checking sequence can be found. Therefore, distinguishing sequence consisting of a same input could help to create a minimal checking sequence. There is a design method polynomial in the number of states that produces suboptimal adaptive distinguishing sequence [Le94]. It is an open question how to adjust the method to create ADS formed mainly from one input symbol. However, exact influence of the choice of ADS on the length of checking sequence is not easy task and should be further researched.

The C-method reduces ADS so that the last symbols are omitted if they are found to be unnecessary. This is determined using the notion of an output-confirmed sequence, see Section 4.3. The same approach can be employed for the M-method. However, several difficulties need to be solved first. The C-method shortens particular d_i based on knowledge of created sequence so far and thus there could be any sequence output-confirmed. On the contrary, the M-method knows only test segments to be in a resulting sequence. We could imagine that all test segments and its subsequences are output-confirmed. Then we shorten some of them according the C-method's rule. That is, let $S_{s',l} \subseteq S \setminus \{s'\}$ be a set of states that are not distinguished from $s' \in S$ by the prefix w_{l-1} of $d_{s'}$ of length $l-1$, i.e. $l-1 = |w_{l-1}|$, but they are distinguished from s' by the prefix w_l of $d_{s'}$ of length l, i.e. $l = |w_l|$. If a transition (s, x) is a prefix of d_s and there is a q-output-confirmed prefix of v, $d_s = x \cdot v$, for each state $q \in S_{s',l}$ and each l, $|d_{s'}| - k < l \leq |d_{s'}|$, that distinguishes q and $s' = \delta(s, x)$, then the last k symbols of $d_{s'}$ in the related test segment can be omitted. This would be nice because it reduced test segments quite well. Nevertheless, such a reduction of a test segment can break the assumption that the resulting sequence contains all d_s's.

During our experiments we found a machine that the C-method produced an n-complete checking sequence which did not display all states using their entire distinguishing sequences d_i's. The machine is attached in Appendix B. The problem is that there is a state s' with only one entering transition $(s, x), s' = \delta(s, x)$. In addition, input x is a prefix of distinguishing sequence d_s so $d_{s'}$ may be shortened. The resulting sequence of the C-method was checked to be checking sequence. Nevertheless, we realized that the M-method needs to be proven formally as it is proposed first and then an improvement by output-confirmed sequences may be employed.

8.1.3 Example

This section describes the M-method on an example from [Si09c]. Using the same example enables us a comparison with the C-method. The sample Mealy machine is shown in Figure 8.4. The M-method is NP-complete in general. Therefore, we employ the M^a-method for describing. The M^a-method approximates the M-method and it is very similar to the C-method because it appends distinguishing sequences when it is possible, see Section 10.4.10.

	a	b
A	B / 0	A / 1
B	E / 0	A / 1
C	D / 1	E / 0
D	D / 0	E / 0
E	A / 1	C / 1

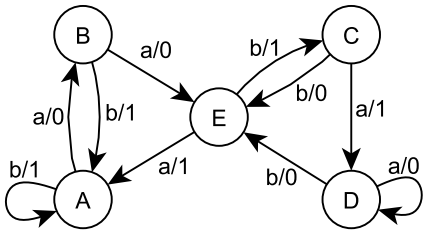


Figure 8.4. An Example Mealy machine from [Si09c]

The authors of [Si09c] uses the following distinguishing sequences in the example:

$$d_A = d_B = aba \qquad d_C = d_D = d_E = ab$$

that we also use to be consistent. After 7 iterations of the C-method (Section 6.2.7) the sequence t_7 is created as follows:

$$t_7 = \underline{\varepsilon}\,{}^{1}_{A} \mid a\,{}^{5}_{B}\,\underline{b}\,{}^{2}_{A}\,a\,{}^{6}_{B} \mid b\,{}^{3}_{A}\,\underline{a}\,{}^{4}_{B} \mid a\,{}^{14}_{E}\,\underline{b}\,{}^{7}_{C}\,a\,{}^{30}_{D} \mid \underline{b}\,{}^{8}_{E} \mid a\,\overline{{}^{25}_{A}\,\underline{b}\,{}^{9}_{A} \mid a\,{}^{10}_{B}\,b\,{}^{11}_{A}\,a\,{}^{12}_{B}} \mid \underline{a}\,{}^{13}_{E}\,a\,{}^{26}_{A}\,b\,{}^{15}_{A}$$

Vertical bars divide the sequence according to iterations of the algorithm. After each sequence u there are a subscript denoting the state $\delta^*(s_0, u)$ and a superscript denoting the sequence number when u becomes confirmed. For example, the sequence after 2 iterations is $t_2 = ababa$, the last state is B and confirmed sequences are $\varepsilon, ab, abab$. Note that the sequence numbers are not related to the number of iterations. See the original paper [Si09c] for a detailed description. The maximal sequence number is 15 after 7 iterations. Hence, sequences $ababaaba^{30}_{D}$, $ababaababa^{25}_{A}$ and $ababaababababaaa^{26}_{A}$ are still not confirmed.

The first 7 iterations of the M^a-method produces the same t_7. Notice that the verified transitions, i.e. related test segments are used, are underline. The difference between the C-method and the M^a-method appears in the 8th iteration. According to confirmed set, transition (A, b) has not been verified yet so the C-method applies this transition and distinguishing sequence of the next state, i.e. $d_A = aba$ because $A = \delta(A, b)$. The C-method creates the following checking sequence of length 36, or 37 if we count the initial setting, i.e. the defined len(T) is used.

$$t_7\,{}^{15}_{A} \mid b\,{}^{16}_{A}\,a\,{}^{17}_{B}\,b\,{}^{18}_{A}\,a\,{}^{19}_{B} \mid a\,{}^{20}_{E}\,a\,{}^{21}_{A}\,a\,{}^{22}_{B}\,b\,{}^{23}_{A}\,a\,{}^{24}_{B} \mid a\,{}^{27}_{E}\,b\,{}^{28}_{C}\,a\,{}^{29}_{D}\,a\,{}^{31}_{D} \mid a\,{}^{32}_{D}\,b\,{}^{33}_{E} \mid b\,{}^{34}_{C}\,b\,{}^{35}_{E}\,a\,{}^{36}_{A}\,b\,{}^{37}_{A}$$

The M^a-method does not verify transition (A, b) in the 8th iteration because its related test segment has been used; it is overlined in t_7. Thus, the M^a-method appends the shortest transfer sequence to a state with an unverified transition, i.e. sequence aa that transfers the machine to state E. Test segment related to the unverified transition (E, a) is applied so t_8 is created.

$$t_8 = t_{7\,A} \mid a_{\,B}\, a_{\,E}\, \underline{a}_{\,A}\, a_{\,B}\, b_{\,A}\, a_{\,B}$$

Then three unverified transitions remain: $(C, a), (C, b), (D, a)$. The basic version of the M^a-method produces the following checking sequence of length 39 because each transition needs to be extended by the entire related distinguishing sequence.

$$t_{8\,B} \mid a_{\,E}\, b_{\,C}\, \underline{a}_{\,D}\, a_{\,D}\, b_{\,E} \mid b_{\,C}\, \underline{b}_{\,E}\, a_{\,A}\, b_{\,A} \mid a_{\,B}\, a_{\,E}\, b_{\,C}\, a_{\,D}\, \underline{a}_{\,D}\, a_{\,D}\, b_{\,E}$$

However, if we employed output-confirmed sequences, the resulting checking sequence would be:

$$t_{8\,B} \mid a_{\,E}\, b_{\,C}\, \underline{a}_{\,D}\, \underline{a}_{\,D} \mid a_{\,D}\, b_{\,E} \mid b_{\,C}\, \underline{b}_{\,E}\, a_{\,A}\, b_{\,A}$$

The sequence has got 33 input symbols. Notice that the sequence corresponds to the one produced by the C-method without bab in the 8th iteration. The C-method decides to verify (A, b) in the 8th iteration. However, if it verified (E, a) first like the M^a-method does, then the transition (A, b) would become verified as well owing to the function of confirmed set. This shows the drawback of the C-method.

The authors stated in [Si09c] that the length of a created checking sequence by the version of the D-method from [Hi06] is 64, by the version of the D-method from [Ch05] is 44, and by the C-method without use of output-confirmed sequences [Si08] is 43. The C-method using output-confirmed sequences [Si09c] creates a checking sequence of length 36. The M-method produces a checking sequence of 33 or 39 inputs for the given ADS if it uses output-confirmed sequences or not.

Our implementation of the M^a-method produces other checking sequence for the machine in Figure 8.4. It is due to the fact that the algorithm from [Le94] found a different adaptive distinguishing sequence. With this ADS

$$d_A = d_D = aaa \qquad d_B = aa \qquad d_C = d_E = ab$$

the M^a-method without employing output-confirmed sequences creates the following checking sequence of length 35.

$$\varepsilon_{\,A} \mid \underline{a}_{\,B}\, \underline{a}_{\,E}\, a_{\,A} \mid \underline{b}_{\,A} \mid a_{\,B}\, a_{\,E}\, \underline{a}_{\,A} \mid a_{\,B}\, a_{\,E}\, a_{\,A} \mid a_{\,B}\, \underline{b}_{\,A}\, a_{\,B}\, a_{\,E}\, a_{\,A} \mid a_{\,B}\, a_{\,E}$$

$$\underline{b}_{\,C}\, a_{\,D}\, \underline{b}_{\,E} \mid a_{\,A}\, b_{\,A} \mid a_{\,B}\, a_{\,E}\, b_{\,C}\, \underline{a}_{\,D}\, \underline{a}_{\,D}\, a_{\,D}\, a_{\,D} \mid a_{\,D} \mid b_{\,E}\, b_{\,C}\, \underline{b}_{\,E}\, a_{\,A}\, b_{\,A}$$

We were able to reduce this checking sequence to 32 input symbols by hand using output-confirmed sequences. Nevertheless, formulation of rules for employing output-confirmed sequences is a future work.

8.2 Resettable FSMs

Checking sequence can be shorten using reset r when the given machine possesses reliable one for each state. The M-method is adjusted so that shortest connecting sequences w_{kj} can be replaced by a sequence $r \cdot w_{0j}$ if it is shorter. We call this modified method the *Mr-method* because reset can be employed and so the resulting test suite contains several sequences. A similar discussion is proposed for the C-method in [Si08].

Chapter 9
Fault Coverage Checker

There are a few methods for checking fault coverage of a given test suite as we mention in Chapter 5. The CSP-based method described in [Zh93] became a basis for us. However, after reading a survey of methods [Pe96] and 'domain reasoning' [Ka12n], we improved the method so the *Fault Coverage Checker* (FCC) was created. It is worth mentioning that methods checking fault coverage of a test suite and generating all machines that response to the test suite equally as the specification do *passive learning* in fact. Such a method gets a test suite and a desired response to the test suite and it tries to reconstruct a model which produces this input-output suite.

The method from [Zh93] as well as the FCC creates a testing tree from a given test suite. A testing tree is a successor tree with specialized nodes. Each node represents a state of the specification. The initial state coincides with the root node. Each test of a given test suite forms a path from the root. Edges are labeled by exactly one input symbol. An example of a testing tree is in Figure 9.2 or in Section 8.1.3. There is a checking sequence interleaving with reached states in Section 8.1.3, i.e. a testing tree with exactly one path from the root to a leaf. A node consists of a domain of states and a set of different nodes in both methods. A domain of a node n_i represents a set of states that can be reached by the sequence of path from the root to n_i in the implementation. The method from [Zh93] denotes domain of node n_i by D_i and set of different nodes by NEQ_i. The FCC uses denotations n_i.domain and n_i.different for domain and set of different nodes of node n_i, respectively.

The method from [Zh93] sorts nodes by the distance from the root after creating a testing tree. Then, each node n_i is compared with the previous ones. A previous node $n_j, j < i$, is added to a set NEQ_i if they are distinguished in the tree, i.e. there is a common sequence from nodes in the tree that produces different responses for both node. A common sequence u means that there are paths labeled by u from both nodes in the testing tree. After this creation of unequal nodes' sets, nodes are again passed through and reference nodes are found. Each state has got one reference node. A node n_i becomes a reference node if it differs from all reference nodes found so far; the difference is checked by set inclusion, i.e. the set of reference nodes has to be a subset of NEQ_i of new reference node n_i. In the beginning, the root is set to be the only one reference node. Each node can be assigned to any state initially so the domain D_i of each node n_i is equal to S, or Q if we deal with extra states in the implementation. When a node n_i becomes a reference one, it is instantiated which means that a state s is assigned to it and its domain reduces to this state, $D_i = \{s\}$. The domains of nodes are reduced using the set of reference nodes and related NEQ_i. After reduction of domains, almost standard CSP search is started. That is, the first uninstantiated node n_i is chosen, i.e. $|D_i| > 1$ and $\forall j < i : |D_j| = 1$, and it is instantiated with a value of D_i that is consistent with NEQ_i and a partial machine created from previously instantiated nodes. Such a search function is repeated unless all nodes try every value of their domains. When all nodes are instantiated, the solution is checked against the specification. All indistinguishable machines are found this way.

Our first straightforward implementation of the method exploded in the number of stored partial machines needed for each estimated state of a node. Even a redesign which stored only changes in the partial machine did not help. In general, the method has time complexity of $O(m^l)$, where l is the number of nodes, $l \leq \text{len}(T)$. Notice that m stands for the number of states in the implementation and the size of a domain as well. Hence, the number of nodes that are to be estimated needs to be minimized as much as possible. We found out in [Pe96] that another approach to fault coverage checking was proposed. It is based on minimization of a testing tree. For example, the method proposed in [Ya94] is based on reduction partial FSM. However, it is very hard task like CSP. We decided to combine these two approaches so we can create all indistinguishable machines and take advantages of minimizing a testing tree that provides new knowledge of testing nodes.

9.1 Motivating example

The main idea of the FCC is reduction of given testing tree, i.e. merging of its nodes. In this section we outline a procedure of the method from [Zh93] and the FCC on a small example. The example is 2-state Mealy machine captured in Figure 9.1. An exact implementation of the FCC is proposed in the following section.

	a	b
A	B / 0	B / 0
B	B / 1	A / 0

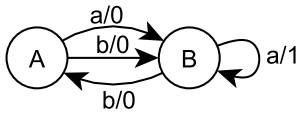


Figure 9.1. A Mealy machine for Fault Coverage Checking Methods' Example

A testing method creates a test suite $T = \{ba, aaa, aba\}$ for the machine in Figure 9.1. Both fault coverage checking methods build a testing tree of T shown in Figure 9.2. There are nodes labeled with their sequence number and related state of the specification. Domains and sets of different nodes are not shown for clarity. Besides input symbols labels of edges contain related outputs that are needed for a distinguishing of nodes. Note that the test suite T is produced by the ADS-method (Section 6.1.2) with sequence a as a ADS, for example.

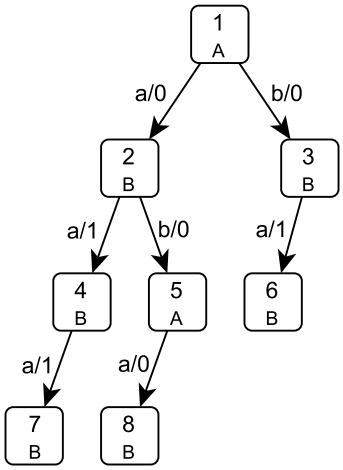


Figure 9.2. A Testing Tree of a Test Suite $T = \{ba, aaa, aba\}$

The method from [Zh93] passes all nodes two times. The first time it compares each node with every node that has got a lower sequence number so sets of unequal nodes NEQ_i's are formed. The second passage reduces domains D_i's that are set to $\{A, B\}$ at the beginning. The root node n_1 is fixed to the initial state A so D_1 reduces to $\{A\}$ and a set of reference nodes R is initialized to $\{1\}$. Then the second node n_2 is found to be different from n_1 and thus it becomes the second reference node due to the fact that $R \subseteq NEQ_2$. No other node can be a reference node because the implementation is assumed to have 2 states as the specification (Figure 9.1). Domains of nodes 3, 4 and 5 are then reduced owing to their sets NEQ_i. The second and third columns of Table 9.1 show the contents of the sets related to nodes after both passages through all nodes. Notice that the domains of reference nodes are emphasized.

	Method of [Zh93]		Fault Coverage Checker		
n_i	NEQ_i	D_i	n_i.domain	n_i.different	Compared with
1	$\emptyset$	A	A	not used	$\emptyset$
2	1	B	B	not used	1
3	1	B	B	not used	1
4	1	B	B	not used	1
5	2,3,4	A	A	not used	1,2
6	$\emptyset$	A,B	A,B	$\emptyset$	1-5
7	$\emptyset$	A,B	A,B	$\emptyset$	1-6
8	$\emptyset$	A,B	A,B	$\emptyset$	1-7

Table 9.1. Procedure of the Fault Coverage Checking Methods

The FCC reduces domains in very similar way as the method from [Zh93]. However, one passage through all nodes is sufficient. In addition, if a node can become instantiated, i.e. its domain is singleton, during domains reduction procedure, then its set of different nodes is unnecessary. Some comparisons of nodes are also needless. It will be clearer after introduction of domains reduction procedure that is captured in Algorithm 1.

After reduction of domains, the FCC merges instantiated nodes into related reference nodes. In our example, nodes 3, 4 and 5 became instantiated during domains reduction procedure. We store instantiated nodes in a stack so an instantiated node with the greatest sequence number is merged first. The procedure of merging is captured for all three instantiated nodes in Figure 9.3. The reference nodes of states A and B are nodes 1 and 2, respectively. Merging of nodes is a recursive procedure so that if two nodes are merged, then their successors on a common path are merged as well. Therefore, when node 5 is merged into its related reference node 1, node 8 as a successor of node 5 is merged with node 2. Similarly, merging of node 4 into its related reference node 2 implies merging of node 7 into node 2 as well. After merging nodes 3 and 6 into node 2, a machine equivalent to the specification is created so no CSP search is needed and the test suite is demonstrated to be n-complete.

The method from [Zh93] creates a partial machine from the testing tree after domains reduction. The partial machine has got no transition at the beginning. If there is an instantiated node having an instantiated child in the testing tree, then the related transition is added to the partial machine. All transitions is verified in the given test suite T hence the method from [Zh93] does not need CSP search like the FCC. However, if the test suite was $T' = \{aaa, baba\}$, then the method from [Zh93] would need to estimate one node unlike the FCC. The FCC would eliminate the uncertainty in the

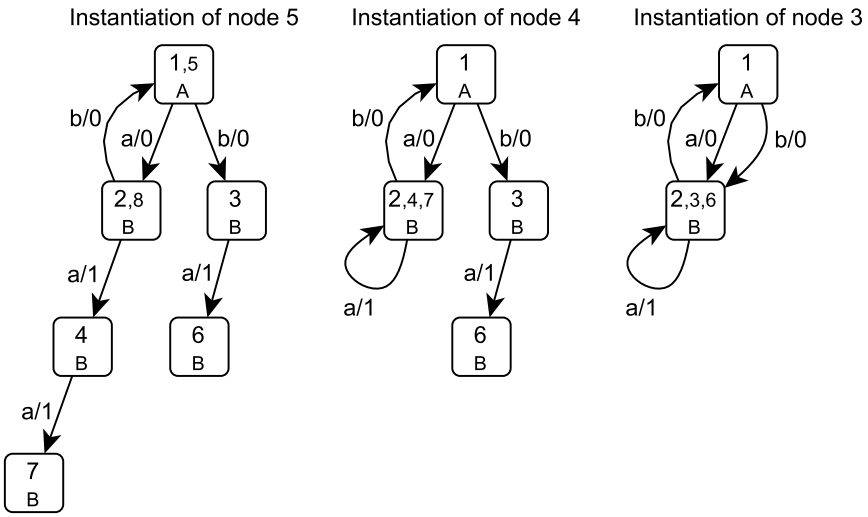


Figure 9.3. Instantiation and Merging of Nodes by the FCC

domains reduction procedure so it would demonstrate n-completeness of T' without CSP search. The proposed example aimed to show notion of merging and improvement in domains reduction procedure that are described in the following section in detail. Note that T' could be created by the Mr-method (Section 8.2).

9.2 Implementation

This section proposes a description of the Fault Coverage Checker with a comparison to the method from [Zh93]. The Fault Coverage Checker first builds a testing tree from the given test suite and sorts the nodes so each node gets a sequence number. The order of nodes can be arbitrary but we keep the order of the method from [Zh93], i.e. the far the node is from the root, the bigger number it gets. This order is preferable due to the fact that nodes closer to the root have higher probability to be reference nodes in test suites with several sequences generated for resettable machines (Section 6.1). The FCC reduces domains of nodes and then it starts a search which produces all machines that have the same response to the test suite as the specification.

There are several differences between the method from [Zh93] and the FCC. The first one is a merging of generating of *NEQ* sets and subsequent domains' reduction into one function so set inclusion operation and *NEQ* sets of instantiated nodes are not needed. The function is called reduceDomains() and it is described in Algorithm 1. The aforementioned minimization, or reduction, of the testing tree is the main difference. That is, if a node is instantiated, it is merged with the related reference node and their successors as well. The merging creates a machine that complies with the desired response to the test suite. Moreover, the merging brings new information to the testing nodes. Some merged nodes are instantiated due to the fact that the domain of merged node gets intersection of domains of both nodes to merge. It is captured in the function mergeNodes() in Algorithm 7. Besides instantiation of a merged node, the reduction of domain by intersection can influence some predecessors.

Two nodes are different if they have a common outgoing sequence with different responses or there are successor nodes reached by equal sequence without a same state in successors' domains. Formally, let u be a common sequence leading from nodes n_i and n_j and $n_{i'}, n_{j'}$ are nodes reached by u from n_i, n_j, respectively, if $\lambda^*(n_i, u) \neq \lambda^*(n_j, u)$

or $D_{i'} \cap D_{j'} = \emptyset$ then nodes n_i and n_j are distinguished. For simplification, we extend the state-transition function δ and the output function λ to nodes such that a node as a parameter of both functions represents its state and the result of the transition function is a successor node instead of a state, i.e. $n_{i'} = \delta^*(n_i, u)$ in the previous definition. We found this notion of different nodes in [Ka12n] and it is a very important improvement in the FCC.

In our implementation of the FCC each node n_i includes:

- n_i.domain - a set of possible states,
- n_i.different - a set of different nodes,
- n_i.state - an instantiated state when $|n_i.\text{domain}| = 1$, i.e. $n_i.\text{domain} = \{n_i.\text{state}\}$,
- a pointer to a successor node $\delta(n_i, x)$ for each $x \in X$ if the successor is defined,
- an output by itself or by each outgoing transition according to the type of machine, i.e. Moore or Mealy type,
- a mark if the node is a reference node,
- a pointer to the predecessor (parent node) for simplification of implementation.

Two arrays of nodes are employed. An array *refNodes* contains reference nodes. For each state s there is always at most one node instantiated to s in *refNodes*. An array *Instantiated* includes instantiated nodes that are to be merged into a related reference node. It can be represented as a queue as well as a stack, it is a choice of order of merging. We use the representation by a stack.

The function reduceDomains() is proposed in Algorithm 1. Each node n_i is first compared to each reference node. The domain is reduced if the nodes are distinguished. In contrast to the method from [Zh93], a node can be instantiated even if it is not different from each reference node. This is possible only when there is a reference node for each state and no extra state is considered. Then, if a node is distinguished from $(n-1)$ reference nodes, its domain is singleton and it is instantiated. Moreover, the node does not have to store different nodes because domain of such a different node is reduced by the instantiated state only once (line 14-16). Consequently, such a different node can be found to have a singleton as domain and thus it is instantiated and possibly it reduces domains of other nodes (line 17). Instantiated nodes need not to be compared so that the number of compared pairs of nodes is less or equal to the one of the method from [Zh93] that always compares all pairs. If a node cannot be instantiated after comparison to reference nodes, it is compared to previous nodes, i.e. nodes with lower index, or sequence number. Instantiated nodes have not been merged into related reference node yet so they can provide a domain reduction (line 22) or even instantiation (line 26). If both nodes n_i, n_j are uninstantiated and they are distinguishable, their different sets are extended by value of each other. This is another difference from the method from [Zh93], its different set NEQ_i stores only lower node's index, i.e. $n_j \in NEQ_i$ implies $j < i$. It is sufficient because the CSP search then instantiates nodes in numerical order. The nodes are necessary in both different sets because of merging and different order of estimated nodes in the FCC.

Notice that we do not need to fix the root as a reference node, it is derived implicitly; the choice of reference nodes is influenced by the order of nodes.

Each pair of nodes is compared by isNodesDifferent() at most once so reduceDomains() runs in $O(l^3)$ if we assume the comparison of nodes in $O(l)$. The FCC benefits over the method from [Zh93] in domains reduction if all n reference nodes are known and no extra states are considered. Then, another nodes can be instantiated and sets

Algorithm 1. Reduction of domains according to the testing tree

```
1  Function reduceDomains()
2      foreach node n_i in the testing tree T (1 ≤ i ≤ l) do
3          unique ← true
4          foreach n_ref ∈ refNodes do
5              if isNodesDifferent(n_ref, n_i, false) then
6                  n_i.domain ← n_i.domain \ {n_ref.state}
7              else unique ← false
8          if unique or |n_i.domain| = 1 then
9              n_i.state ← a value of n_i.domain
10             if unique then
11                 n_i.domain ← {n_i.state}
12                 refNodes.push(n_i)
13             else Instantiated.push(n_i)
14             foreach node n_j (1 ≤ j < i) do
15                 if |n_j.domain| > 1 and isNodesDifferent(n_j, n_i, false) then
16                     n_j.domain ← n_j.domain \ {n_i.state}
17                     if |n_j.domain| = 1 then instantiate(n_j)
18         else
19             foreach node n_j (1 ≤ j < i) do
20                 if isNodesDifferent(n_j, n_i, false) then
21                     if n_j is instantiated then
22                         n_i.domain ← n_i.domain \ {n_j.state}
23                     else
24                         n_j.different ← n_j.different ∪ {n_i}
25                         n_i.different ← n_i.different ∪ {n_j}
26             if |n_i.domain| = 1 then instantiate(n_i)
```

of different nodes are smaller because an instantiated node reduces domain of different nodes. Moreover, it does not have to appear in the set of different nodes and it does not need to be compared with the following instantiated nodes.

A node is added to *Instantiated* just in a simple function instantiate() that is proposed in Algorithm 2. After assigning a state to given node, it reduces domains of different nodes if it is possible. It is worth mentioning that Algorithms 1-7 are illustrative only, they do not provide full logic needed to avoid an infeasible solution, e.g. there should be a check that *diffNode* is not instantiated to the same state as *node* in Algorithm 2.

Algorithm 2. Instantiation of given node with different nodes' domain reduction

```
1  Function instantiate(node)
       input : node - a node to instantiate
2      node.state ← the value of node.domain
3      update refNodes if there is no node of node.state
4      Instantiated.push(node)
5      foreach diffNode ∈ node.different do
6          if node.state ∈ diffNode.domain then
7              diffNode.domain ← diffNode.domain \ {node.state}
8              if |diffNode.domain| = 1 then instantiate(diffNode)
```

Algorithm 3 captures the mentioned checking of nodes' difference. We use this function in two places. The first one is in reduction of domains at the beginning when the testing tree is handled and nodes are passed from the root so intersection of domains does not arise a new information. The second use is after merging some nodes. The structure of nodes thus can contain a cycle. We want to differ the first given node n_i from a reference node so the comparison can be stopped when an instantiated successor of n_i is reached. Notice that the following always holds. Either there is no common input of given nodes or an instantiated node is reached. Therefore, the function isNodesDifferent() stops after a finite number of steps.

Algorithm 3. A check whether the given nodes are distinguished

```
1  Function isNodesDifferent(n_i, n_j, cyclic)
       input  : n_i, n_j - nodes to compare
                cyclic - true if the structure of nodes could contain a cycle
       output: true if the given nodes are distinguishable, false otherwise
2      if cyclic and n_i is instantiated then
3          return (n_i.domain ∩ n_j.domain) = ∅
4      if Moore and λ(n_i) ≠ λ(n_j) then return true
5      foreach x ∈ X such that δ(n_i, x) and δ(n_j, x) are defined do
6          if Mealy and λ(n_i, x) ≠ λ(n_j, x) then return true
7          if isNodesDifferent(δ(n_i, x), δ(n_j, x), cyclic) then return true
8      return false
```

The array *Instantiated* is filled by instantiated (not reference) nodes after reduceDomains(). Hence, the merging process can start. We call function proccessInstantiated() followed by function checkUninstantiated(). Equivalently, we can just call function search() on an instantiated node. The function search() is proposed in Algorithm 4. It comprises both functions needed to call. Calling search() on an instantiated node is safe if the function instantiate is treated against processing an instantiated node twice. Nevertheless, there is a possibility that *Instantiated* is empty after domains reduction. Then search() is called on a node with the smallest domain. Such a node is returned by the function checkUninstantiated().

Algorithm 4. CSP search with domain reasoning

```
1  Function search(node)
       input  : node - a test node to instantiate
2      foreach state ∈ node.domain do
3          node.domain ← {state}
4          instantiate(node)
5          while Instantiated is not empty do
6              processInstantiated()
7              uninstantiatedNode ← checkUninstantiated()
8          if uninstantiatedNode exists then search(uninstantiatedNode)
9          else check solution
10         revert changes
```

The function search() simply tries each state from the given node's domain. Each instantiation implies a reduction of domains of other nodes. If there is an uninstantiated node after processing of all instantiated nodes, the function search() is recursively called on such a node whose domain is not singleton (line 8). Otherwise, a solution is checked against the previous ones (line 9). Only non-isomorphic solution is saved. For simplicity, we assumes n-complete testing method on completely-specified FSMs in this thesis so solutions are complete machines again. Therefore, we can check isomorphism by minimization of the solution and then straight comparison of reduced machines, i.e. the initial states are fixed so machines are isomorphic if there is a bijective function mapping states of one machine to states of the other.

After each attempt to instantiate the given node with a state, all instantiation, domain reduction and merging implied by the instantiation of the node are returned to the previous state. We create a copy of nodes' structure before each try to handle these changes reverting. Note that an instantiation can lead to an infeasible solution, i.e. different nodes are instantiated with the same state or a domain of node is empty. Such a fail can be detected in each of functions instantiate(), processInstantiated() and checkUninstantiated(). If this happens, search() revert changes immediately and start another attempt to instantiate the node (if there is an untested state in the domain).

Algorithm 5. Merging all instantiated into related reference nodes

```
1 Function processInstantiated()
2     while Instantiated is not empty do
3         node ← Instantiated.pop()
4         n_ref ← n_ref ∈ refNodes and n_ref.state = node.state
5         if node ≠ n_ref then mergeNodes(node, n_ref)
```

Algorithm 5 describes the function processInstantiated() that simply passes *Instantiated* and calls a recursive merging of a node of *Instantiated* and its related reference node. After all instantiated nodes are processed so the nodes' structure (the testing tree at the beginning) gets smaller due to merging, all uninstantiated nodes are checked according to search(). The function checkUninstantiated() in Algorithm 6 looks at each uninstantiated node n_i and checks whether a state of its domain can be eliminated. The function isNodesDifferent() (Algorithm 3) is employed. Notice 'true' as the third parameter (line 5) because some previous merging could create a cycle which contains n_i. That is, isNodesDifferent() would loop forever because n_i and its successor would have a successor to compare if a 'cyclic' condition were not in isNodesDifferent().

Algorithm 6. Possible reduction of domains of uninstantiated nodes

```
1 Function checkUninstantiated()
      output: an uninstantiated node with the smallest domain if exists
2     U ← ∅
3     foreach node n_i with |n_i.domain| > 1 do
4         foreach n_ref ∈ refNodes such that n_ref.state ∈ n_i.domain do
5             if isNodesDifferent(n_i, n_ref, true) then
6                 n_i.domain ← n_i.domain \ {n_ref.state}
7         if |n_i.domain| = 1 then instantiate(n_i)
8         else U ← U ∪ {n_i}
9     return n_i from U with the smallest domain if U ≠ ∅
```

Algorithm 7. Merging a given node to another one

```
1  Function mergeNodes(fromNode, toNode)
       input : fromNode, toNode - nodes to be merged
2      toNode.domain ← fromNode.domain ∩ toNode.domain
3      toNode.different ← fromNode.different ∪ toNode.different
4      if |toNode.domain| = 1 then
5          if toNode is not instantiated then instantiate(toNode)
6          else if fromNode is not instantiated then
7              fromNode.domain ← toNode.domain
8              instantiate(fromNode)
9      foreach x ∈ X such that δ(fromNode, x) is defined do
10         if δ(toNode, x) is defined then
11             if δ(fromNode, x) is a reference node then
12                 if δ(toNode, x) is not instantiated then
13                     δ(toNode, x).domain ← δ(fromNode, x).domain
14                     instantiate(δ(toNode, x))
15             else mergeNodes(δ(fromNode, x), δ(toNode, x))
16         else δ(toNode, x) ← δ(fromNode, x)
```

The last thing to describe is merging two nodes by the function mergeNodes() proposed in Algorithm 7. For saving a computer memory, we take the first node as a node which transfers its informations into the second node instead of creating a new node that represents a merged node. Moreover, the first node can be deleted because it is contained in the second node in the end of merging. The merged node, i.e. the second given node, updates its domain and different set with the information of the first node. The given nodes represent the same state therefore intersection of domains and union of different sets are done. If one of given nodes is not instantiated but the merged one is, instantiate() is called. Consequently, successors of nodes are merged. If the first node possesses a successor on an input x and the second node does not any on x, the successor is simply appended to the merged node (line 16). The second case that is needed to handle is when there is a common input for both nodes. The successors are merged if the successor of the first node is not a reference node. This is due to the fact that the first node passed to mergeNodes() is removed in the end of the function and we do not want to remove a reference node. The successors can have been merged or will be merged. Therefore, line 12 contains another condition dealing with instantiation of the successor of the second node. That is, if the successor of the second node is not a reference (have not been merged) and is not in *Instantiated* (is not going to be merged), it is instantiated.

9.3 Hint of Reference Nodes

The testing methods based on minimal state cover set (see Section 6.1) produce an m-complete test suite with several sequence. States in such test suite T are uniquely identified so the FCC does not need to estimate any node on T if no extra states are considered.

The methods generating a checking sequence are more complicated. The order of nodes processing in reduceDomains() and related identifying of reference nodes is crucial. Consider the following example. The C-method on a 3-state FSM with binary input and output alphabets and an ADS aa creates a checking sequence sketched in

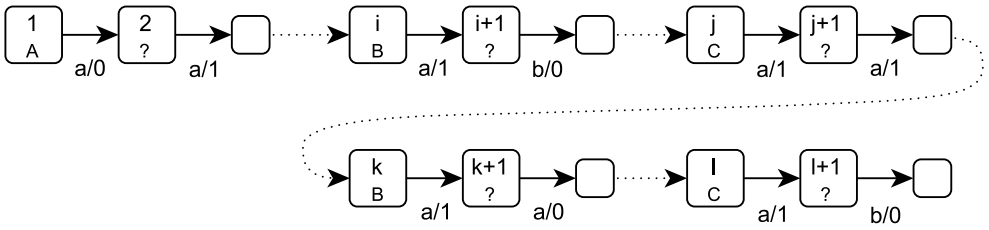


Figure 9.4. A Checking Sequence displayed the Need of a Hint of Reference Nodes

Figure 9.4. The FCC does not know that sequence aa is needed to distinguish all reference nodes. It simply passes through nodes in given order. Sequence aa leads from the first processed node n_1 with response 01. The node n_1 is set as the first reference node. Then, a node n_i is distinguished from n_1 because a transition $a/1$ leads from n_i. Thus, n_i is set as the second reference node. No other node can be chosen as the third reference node because the third state C responds equally to sequence ab as state B of node n_i. Sequence ab is the only one that can be used for a distinguishing from n_i. If the FCC started with nodes n_1, n_j and n_k that are appended with distinguishing sequence aa, then all reference nodes would be found and thus domains would be reduced more.

We resolve this problem by providing a hint for the FCC. Based on knowledge of the testing method, we choose nodes of the testing tree that are to be processed in reduceDomains() first. In our example, nodes n_1, n_j and n_k are chosen. The function reduceDomains() then marks the chosen nodes as reference nodes implicitly. Notice that this 'hint' does not break correctness or completeness of the FCC but it speeds up the search enormously.

Two observations conclude this chapter. Searching for reference nodes is similar to finding of n-clique in distinguishability graph. A distinguishability graph is built in the first step of the fault coverage checking method from [Si10]. The second observation is that the FCC creates a confirmed set in fact. It follows Theorem 8.1. Reference nodes represent minimal state cover so the condition 1 of Theorem 8.1 holds. Then, merging and instantiations provide next confirmed sequences.

Chapter 10
Implementation

We compare some testing method of Chapter 6 in the following chapter and we want to propose replicable experiments. Therefore, we describe our implementation of the methods in this chapter. It is needed because each method's design contains some nondeterministic choice that influences the result. An example of such nondeterminism is an order of transitions' verification or a choice of one of several shortest distinguishing sequences. Thus, we need to state how we handle these nondeterministic parts to be able to make a valuable comparison. There are several very good comparisons of testing methods [Do05e, En13] however they did not state how a family of harmonized state identifiers is obtained, for example. Hence, the results could be different if one chooses other implementation of the same method.

This chapter begins with a discussion how separating sequences can be obtained and how a state characterizing set, a characterizing set and a set of harmonized state identifiers can be formed from separating sequences. Then a description of the state and transition cover design methods is proposed. In addition, a special structure for storing prefix-closed set is introduced in the same section. Subsequently, we will use proposed algorithms and structures to implement the testing methods.

States, input symbols and outputs symbols are numbered from zero so that the transition and the output function can be easily stored in arrays. Thus, we use $s_i \in S$ and $x_j \in X$ as indexes and it allows us to do standard arithmetic operations with states or inputs. More about representation of FSMs in our C++ library and how sample machines for experiments are generated is proposed in [So14].

10.1 Separating sequences

A separating sequence w of states s_i and s_j produces different responses when it is applied to both states, i.e. $\lambda^*(s_i, w) \neq \lambda^*(s_j, w)$. Knowledge of separating sequences is significant for state identification. Note that distinguishing sequence and state verifying sequence are special cases of separating sequences. We focus on creating separating sequences in general, i.e. for each pair of states in a complete reduced FSM, because such sequences always exist. When one has a separating sequence for each states pair, state characterizing sets, characterizing set and a family of harmonized state identifiers can be easily arranged as it will be shown in the following section.

Separating sequences are equal for states pair (s_i, s_j) as for (s_j, s_i) therefore it is sufficient to store just for one pair. We use a *pairs array* [So14] for storing sequences. It is a one-dimensional array of length $K = n(n-1)/2$ and the bijective relation of a states pair (s_i, s_j) and the index $\text{cell}(s_i, s_j)$ in pairs array follows:

$$\text{cell}\,(s_i, s_j), s_i < s_j \;\longleftrightarrow\; s_i \cdot n + s_j - 1 - \frac{s_i \cdot (s_i + 3)}{2} \tag{1}$$

Four methods are proposed in the following sections. Then an example and appropriate steps of algorithms are demonstrated. All methods are very simple. States are distinguished by outputs first and then by the next states.

10.1.1 Shortest sequences

We proposed the design method finding shortest separating sequences for each pair of states in [So14] (Section 3.5.1):

1. For each pair of states $(s_i, s_j) \in S \times S, s_i \neq s_j$, apply each input $x \in X$. If states s_i and s_j produce on a x different output, store this input x as shortest separating sequence for pair (s_i, s_j). For Moore machines, distinguish pairs of states first by ε.
2. For each pair $(s_i, s_j), s_i \neq s_j$, distinguished by the input sequence w in the previous step try to find an undistinguished pair $(s_k, s_l), s_k \neq s_l$, and an input x such that the pair of the next states $(\delta(s_k, x), \delta(s_l, x)) = (s_i, s_j)$. If there are such pair and input, store $x \cdot w$ as a separating sequence for the pair (s_k, s_l).
3. Repeat step 2 until there is a pair of undistinguished states $(s_k, s_l), s_k \neq s_l$.

The design method contains a few uncertainties so the implementation is proposed in Algorithm 8. Three data structures are employed:

- V is a pairs array to store a separating sequence of the related states pair,
- *Distinguished* is a queue of states pairs that have already set the separating sequence,
- *Link* contains a list of previous states pairs and transferring inputs for each states pair

States and input symbols are numbered so the 'foreach' cycles are determined by numerical order. Notice that line 16 prevents a self-loop adding and lines 6, 11, 12 and 18 shorten searching because only one of shortest separating sequences is to be found.

Algorithm 8. Design method of shortest separating sequences

```
1  foreach pair of states (s_i, s_j) do
2      idx ← index of (s_i, s_j) in V
3      if type is Moore and λ(s_i, ε) ≠ λ(s_j, ε) then
4          V[idx] ← ε
5          Distinguished.push(idx)
6          continue
7      foreach x ∈ X do
8          if λ(s_i, x) ≠ λ(s_j, x) then
9              V[idx] ← x
10             Distinguished.push(idx)
11             break
12     if V[idx] is set then continue
13     foreach x ∈ X do
14         if δ(s_i, x) ≠ δ(s_j, x) then
15             nextIdx ← index of (δ(s_i, x), δ(s_j, x)) in V
16             if idx ≠ nextIdx then
17                 Link[nextIdx].push((idx, x))
18             if V[nextIdx] is set then break
19 while Distinguished is not empty do
20     idx ← Distinguished.pop()
21     foreach (prevIdx, x) ∈ Link[idx] do
22         if V[prevIdx] is not set then
23             V[prevIdx] ← x · V[idx]
24             Distinguished.push(prevIdx)
```

The proposed Algorithm 8 runs in $O(n^2 \cdot |X|)$. It passes through all states pairs and in the worst case all inputs are checked at first. The number of states pairs is $K = n(n-1)/2$ which is in $O(n^2)$. The queue *Distinguished* can contain at most $K \cdot |X|$ elements. Therefore, both main cycles (line 1 and 19) are processed in $O(K \cdot |X|)$. Thus, the time complexity of the algorithm, $O(n^2 \cdot |X|)$, is proved.

10.1.2 Parallel approaches

Parallelization of an algorithm is able to reduce running time. As the design method finding shortest separating sequences processes states pairs relatively independently, the method should be parallelized quite well. We propose two parallel approaches adapted to run on GPU. GPU can be imagine as multiprocessor with many elementary cores. For simplicity, each core handles one thread and cores are grouped into blocks. A function called *kernel* runs on a specific number of blocks and every thread in such a block follows the same kernel but each thread handles different data. More about programming in parallel and on GPU can be found in [Ki12], for example.

Basic idea for parallelization the design method is handling each states pair by one thread. However, a couple of problems have to be resolved at first.

On GPU it is very hard to work with linked list which we use as a structure of sequence. Therefore, two pairs arrays, *SepInput* and *SepNextPair*, are employed instead. *SeqInput* stores only the first input symbol of sequence in a cell. *SepNextPair* links the cell with the next states pair cell. If a pair is distinguished by the input stored in the related cell in *SepInput*, the cell in *SepNextPair* contains a link to itself as a mark of sequence's end; a link is the index of cell in pairs array.

Each thread has a unique identifier that corresponds to the index in pairs array in our case. The derivation of state indexes from the index of pairs array needs to find a square root. This is a very costly operation if each thread has to do at the beginning of its run. Therefore, another array is employed. We called it *Mapping*, it has the length as previous mentioned pairs arrays and it stores lower state index related to given cell. The second state index is easily computed according to Relation (1) of cell(s_i, s_j).

Straightforward approach

The first proposed parallel approach simply follows the design method in Section 10.1.1. Step 1. is implemented by the kernel in Algorithm 9 or Algorithm 10 according to the type of given machine. The kernels distinguish pair of states that produces different output on some input symbol, or on the empty string in case of Moore machine. The kernel in Algorithm 11 represents step 2. and it is repeated if there is an undistinguished states pair (step 3.). Algorithm 11 distinguishes an undistinguished pair of states that transfers to distinguished states pair on some input.

Algorithm 9. Kernel Separation by output for Moore machines

```
1 idx ← global identifier of thread
2 s_i ← Mapping[idx]
3 s_j ← idx = cell(s_i, s_j)
4 if λ(s_i, ε) ≠ λ(s_j, ε) then
5 |   SepInput[idx] ← ε
6 |   SepNextPair[idx] ← idx
7 |   SepSeqLength[nextIdx] ← 0
```

We want to create shortest separating sequences. Therefore, we need to consider following property of parallel algorithm. A thread can distinguish related pair of states

Algorithm 10. Kernel Separation by output for Mealy machines

```
1 idx ← global identifier of thread
2 s_i ← Mapping[idx]
3 s_j ← idx = cell(s_i, s_j)
4 foreach x ∈ X do
5     if λ(s_i, x) ≠ λ(s_j, x) then
6         SepInput[idx] ← input
7         SepNextPair[idx] ← idx
8         SepSeqLength[nextIdx] ← 1
9         break
```

if the next states pair has been distinguished. However, the next states pair can become distinguished in the same repetition of Algorithm 11. Although threads shall run in parallel, it is not ensured exactly when they set a certain value unless they are synchronized. In general, we use several blocks of threads so we can only synchronize the entire GPU. We solved this issue by keeping the length of separating sequence for each states pair in the array *SepSeqLength*. We also need to know the number of distinguished pairs. Therefore, each thread atomically increment the global counter *DistinguishedCount* once the related pair is distinguished. If the value of this counter is less than the number of states pairs, Algorithm 11 is processed again.

Algorithm 11. Kernel Separation by the next states

```
   input: len - the length of longest found separating sequence
1  idx ← global identifier of thread
2  if SepNextPair[idx] is not set then
3      s_i ← Mapping[idx]
4      s_j ← idx = cell(s_i, s_j)
5      foreach x ∈ X do
6          s'_i ← δ(s_i, x)
7          s'_j ← δ(s_j, x)
8          if s'_i ≠ s'_j then
9              nextIdx ← cell(s'_i, s'_j)
10             if SepNextPair[nextIdx] is set and
               SepSeqLength[nextIdx] = len then
11                 SepInput[idx] ← x
12                 SepNextPair[idx] ← nextIdx
13                 SepSeqLength[idx] ← len + 1
14                 begin atomic
15                     DistinguishedCount ← DistinguishedCount + 1
16                 break
```

Consider a set W of all found distinguishing sequences, i.e. a union of shortest separating sequences over all states pairs. Let w be the longest distinguishing sequence of W. The length of w is bounded by the number of states, i.e. $|w| < n$, or $|w| \in O(n)$. A thread goes at most through all inputs, so its running time is $O(|X|)$ for each of three kernels. Either Algorithm 9 or Algorithm 10 runs once according to the type of machine. Algorithm 11 is started $|w|$-times. Therefore, time complexity of the Straightforward parallel approach is $O(|w| \cdot |X|) = O(n \cdot |X|)$.

Approach using a queue

The first parallel approach always uses all $K = n(n-1)/2$ threads although most of them have no work because their related pairs have already been distinguished. As a consequence of this disadvantage, we developed another method.

The second approach parallelizes the sequential method captured in Algorithm 8. Each thread first distinguishes pair of states by an output, see Algorithm 9 and 10. The difference is that *SepSeqLength* is not needed and in this place we push *idx* into queue *Distinguished*. A queue is represented on GPU as a buffer and an index pointer to the last element. The pointer *DistinguishedCount* of *Distinguished* is atomically incremented by a thread just before the thread stores its index in the queue. An *atomic* operation is irreducible so only one thread can change the content at a time.

Algorithm 8 uses *Link* as pairs array of lists which is again a problem on GPU. Hence, arrays *LinkInput* and *LinkPrevPair* are introduced in place of *Link*. They contain the same data as *Link* but in one dimensional array. Therefore, pairs arrays *LinkIdx* and *LinkSize* have to be employed to store a start index and length of list in *LinkInput* and *LinkPrevPair* for a particular entry of *Link*. The first element of $Link[idx]$ can be then accessed as $(LinkPrevPair[k], LinkInput[k])$, where $k = LinkIdx[idx]$, and the length of $Link[idx]$ is $LinkSize[k]$, for example.

Sizes of *LinkInput* and *LinkPrevPair* are not known apriori so we fill *LinkIdx* and *LinkSize* at first. Algorithm 9 and Algorithm 10 are extended in the following way. If a pair with index *idx* is not possible to distinguish, then $LinkIdx[nextIdx]$ is atomically incremented by one for each valid next states pair with index *nextIdx*. A next states pair is valid if the next states are not equal and its index is different from the source pair's index, i.e. $idx \neq nextIdx$. The same procedure will be done in Algorithm 12 with the array *LinkSize*. *LinkIdx* contains the number of links for each states pair when the first kernel finishes.

The exclusive all-prefix-sums operation on *LinkIdx* is then employed to derive start indexes and sizes of *LinkInput* and *LinkPrevPair*. The all-prefix-sums operation on an array of data is known as *scan* [La80]. It recalculates entries of *LinkIdx* such that $LinkIdx[i]$ will contain $\sum_{j<i} LindIdx[j]$. Moreover, it calculates the total sum of *LinkIdx* which is needed for allocation of *LinkInput* and *LinkPrevPair* on GPU.

Algorithm 12. Kernel Filling previous pair's Link

```
1  idx ← global identifier of thread
2  if SepNextPair[idx] is not set then
3      s_i ← Mapping[idx]
4      s_j ← idx = cell(s_i, s_j)
5      foreach x ∈ X do
6          s'_i ← δ(s_i, x)
7          s'_j ← δ(s_j, x)
8          if s'_i ≠ s'_j then
9              nextIdx ← cell(s'_i, s'_j)
10             if idx ≠ nextIdx then
11                 begin atomic
12                     linkIdx ← LinkSize[nextIdx]
13                     LinkSize[nextIdx] ← LinkSize[nextIdx] + 1
14                 linkIdx ← linkIdx + LinkIdx[nextIdx]
15                 LinkInput[linkIdx] ← x
16                 LinkPrevPair[linkIdx] ← idx
```

After scan finishes and '*Link* structures' are allocated, Algorithm 12 starts. It fills *LinkInput* and *LinkPrevPair* with values like Algorithm 8. *LinkSize* as other structures is initialized with zeros so when Algorithm 12 begins, $LinkSize[i] = 0$ for each i.

The last kernel is described in Algorithm 13. It works on one block of threads and one thread handles one distinguished pair of states in the queue *Distinguished*. Pairs are processed in waves according to the length of separating sequence of given states pair. That is, the pairs distinguished by output directly are processed simultaneously first, then the pairs distinguished by sequence of length 2. These pairs may have links to some undistinguished pairs of states. Such undistinguished pairs are again simultaneously treated in the next round.

Algorithm 13. Kernel Processing the Distinguished

```
1  idx ← global identifier of thread
2  base, count ← 0
3  repeat
4      base ← count
5      count ← DistinguishedCount
6      while idx < count − base do
7          idx ← Distinguished[base + idx]
8          for k = 0 to LinkSize[idx] do
9              prevIdx ← LinkIdx[idx] + k
10             prev ← LinkPrevPair[prevIdx]
11             begin atomic
12                 if SepNextPair[prev] is not set then
13                     isDist ← true
14                     SepNextPair[prev] ← idx
15                 else isDist ← false
16             if isDist then
17                 SepInput[prev] ← LinkInput[prevIdx]
18                 begin atomic
19                     distIdx ← DistinguishedCount
20                     DistinguishedCount ← DistinguishedCount + 1
21                 Distinguished[distIdx] ← prev
22         base ← base + number of threads
23     synchronize threads
24 until DistinguishedCount < number of state pairs
```

Calculation of time complexity can be divided into four parts according to kernels. The first kernel runs in $O(|X|)$ because each thread goes through all inputs and either a separating sequence (one input symbol) is found or next states pair's link counter is increased. Then scan is employed and it takes $O(\log_2(K))$ [La80], where $K = n(n-1)/2$. Algorithm 12 needs $O(|X|)$ for filling *Link*'s structures. Under the assumption that each states pair has approximately $|X|$ previous states pair, i.e. $|X|$ entries in *Link*, Algorithm 13 runs in $O(|w| \cdot |X|) = O(n \cdot |X|)$, where w is the longest distinguishing sequence, see time complexity derivation of the first parallel method. Estimation of the number of links for each states pair is reasonable due to the fact that $|Link| < n \cdot |X|$ and all *Link*'s entries are processed in $|w|$ cycles of 'repeat' cycle in Algorithm 13. That is, a thread treats $\frac{|Link|}{|w|} \in O(\frac{n \cdot |X|}{n}) = O(|X|)$ entries per cycle in average. Thus, total time complexity is $O(|X| + \log_2(K) + |X| + n \cdot |X|) = O((n+2) \cdot |X|)$.

10.1.3 All sequences

The methods proposed so far find some of shortest separating sequences for a particular pair of states. Nonetheless, it could be beneficial to know all separating sequences sometimes. For example, the H-method (Section 6.1.8) chooses a separating sequence on the fly. Listing of all separating sequences takes a lot of computer time and space therefore we create a special linked structure L instead. Each separating sequence can be then easily obtained from the L. Particularly, L is a pairs array that contains two components in each cell. The former component called *next* is an array of indexes to L for each input symbol. The latter component is a number *minLength* determining the length of shortest separating sequence for given pair of states. Both components are initialized with values of -1.

L can be also imagine as a directed graph. Nodes are cells of L and edges are given by *next* of particular cell. A self-loop indicates that pair of states is distinguished by related input, i.e. label of the self-loop edge. In other words, we use the cell's index as mark of separating input symbol. Moreover, *minLength* is 0 if the related pair produces different outputs and the type of machine is Moore.

The method for finding all separating sequences shown in Algorithm 14 is very similar to Algorithm 8. The difference is that searching is not shortened so each input is processed for each states pair and links are thus set. Notice that the second main cycle (line 16) is needed only for determining the minimal length of separating sequence for each pair of states.

Algorithm 14. Finding of all separating sequences

```
1  foreach pair of states (s_i, s_j) do
2      idx ← index of (s_i, s_j) in L
3      if type is Moore and λ(s_i, ε) ≠ λ(s_j, ε) then
4          L[idx].minLength ← 0
5      foreach x ∈ X do
6          if λ(s_i, x) ≠ λ(s_j, x) then
7              L[idx].next[x] ← idx
8              if L[idx].minLength = −1 then
9                  L[idx].minLength ← 1
10                 Distinguished.push(idx)
11         else if δ(s_i, x) ≠ δ(s_j, x) then
12             nextIdx ← index of (δ(s_i, x), δ(s_j, x)) in L
13             if idx ≠ nextIdx then
14                 L[idx].next[x] ← nextIdx
15                 Link[nextIdx].push((idx, x))
16 while Distinguished is not empty do
17     idx ← Distinguished.pop()
18     foreach (prevIdx, x) ∈ Link[idx] do
19         if L[prevIdx].minLength = −1 then
20             L[prevIdx].minLength ← L[idx].minLength + 1
21             Distinguished.push(prevIdx)
```

The proposed Algorithm 14 runs in $O(n^2 \cdot |X|)$ like Algorithm 8. We can even state that time complexity of the algorithm is $\Theta(n^2 \cdot |X|)$ because each pair of states has to be processed with each input symbol.

10.1.4 Example

We proposed four slightly different approaches for creating separating sequences in the previous sections. This section provides a description how each approach works and creates separating sequences of the Moore machine M given in Figure 10.1.

	a	b	ε
A	A	C	0
B	C	A	0
C	B	D	0
D	D	A	1

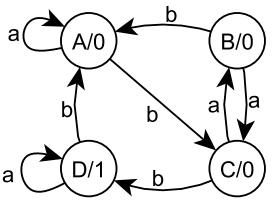


Figure 10.1. A Moore machine for Separating Sequences Example

The first approach described in Algorithm 8 finds some of the shortest separating sequences for each pair of states. Its process of sequence creating is captured in Figure 10.2. There are a distinguishing table on the left and steps of the algorithm on the right. The distinguishing table shows the content of pairs array V more clearly. Rows and columns are labeled with states so there is a separating sequence of related pair of states in a cell. States are denoted A, B, C and D but in fact they are numbered $0, 1, 2$ and 3, respectively. Therefore, the pairs array index can be calculated according Relation (1). Index is then shown in top left of the cell and it is used in the algorithm's steps as reference to particular cell.

	B	C	D
A	^{0} ab $\Rightarrow$	^{1} b	^{2} ε
B		^{3} b	^{4} ε
C			^{5} ε

1. ($\Rightarrow$) $Link[1] \leftarrow (0, a), (0, b)$
2. $1 \leftarrow b$
3. $2 \leftarrow \varepsilon$
4. $3 \leftarrow b$
5. $4 \leftarrow \varepsilon$
6. $5 \leftarrow \varepsilon$
7. $0 \leftarrow ab$

Figure 10.2. Function of Algorithm 8 on the Moore machine (Figure 10.1)

Algorithm 8 processes given Moore machine in 7 steps as it is shown in Figure 10.2. Each step depicts a significant change, e.g. assignment of a separating sequence to certain cell. The first six steps handle each pair of states for the first time, i.e. 'foreach' cycle on line 1. The last step goes through *Distinguished* (line 19) and creates the separating sequence ab of states A and B using $Link[1]$ filled in the very first step.

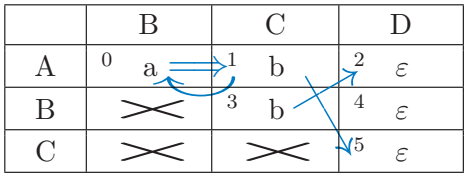
The Straightforward approach parallelizes creating of sequences so it finds some of shortest separating sequences for each states pair just in 3 steps. Steps and a distinguishing table are shown in Figure 10.3. The distinguishing table on the left captures pairs array *SepInput* and blue arrows represent connections of pairs array *SepNextPair*. Next states pair is in brackets after distinguishing input symbol in steps 2 and 3. *SepInput* contains only an input symbol in a cell so *SepNextPair* is needed for separating sequence listing. Therefore, cell 0 of states A and B contains a in the distinguishing table but the resulting separating sequence is ab.

	B	C	D
A	^{0} a $\rightarrow$	^{1} b	^{2} ε
B		^{3} b	^{4} ε
C			^{5} ε

1. $2 \leftarrow \varepsilon,\ 4 \leftarrow \varepsilon,\ 5 \leftarrow \varepsilon$
2. ($\rightarrow$) $1 \leftarrow b(5),\ 3 \leftarrow b(2)$
3. ($\rightarrow$) $0 \leftarrow a(1)$

Figure 10.3. Function of the Straightforward approach on the Moore machine (Figure 10.1)

The Approach using a queue tries to minimize the number of used threads compared to the Straightforward approach. Nevertheless, the preparation of *Link* structures and queue *Distinguished* takes some additional time. The function of the Approach using a queue on the Moore machine can be described in 4 steps as Figure 10.4 shows. In the first step, states pairs with indexes 2, 4, 5 are distinguished and the other pairs increase counters in $LinkIdx$. Pair of states A and C (index 1) has 2 incoming transitions, $(0, a)$ and $(0, b)$ in particular, so $LinkIdx[1] = 2$. The second step runs *scan* that recalculates $LinkIdx$. It also sums $LinkIdx$ before adjustment. The *sum* is needed for allocation of *Link* structures, i.e. both $LinkInput$ and $LinkPrevPair$ have the length of 5. Algorithm 12 then fills *Link* structures as step 3 shows. Notice the correspondence of $LinkIdx$ entries and use of comma and semicolon between elements of *Link* structures. For instance, $LinkIdx[2] = 3$ so that links of cell 2 start on index 3 and Algorithm 12 computes $LinkSize[2] = 1$ that means there is one such a link. This link is $(3, b)$ because $LinkPrevPair[3] = 3$ and $LinkInput[3] = b$. The last step processes queue *Distinguished*, see Algorithm 13. In the first round, it distinguished states pairs with indexes 1 and 3. The second round, or cycle, sets distinguishing input a to cell 0.



1. $2 \leftarrow \varepsilon,\ 4 \leftarrow \varepsilon,\ 5 \leftarrow \varepsilon$
   $LinkIdx \leftarrow 1, 2, 1, 0, 0, 1$
2. $LinkIdx \leftarrow 0, 1, 3, 4, 4, 4;\quad sum \leftarrow 5$
3. $Link \leftarrow (1, a); (0, a), (0, b); (3, b); (1, b)$
4. $3 \leftarrow b(2),\ 1 \leftarrow b(5);\quad 0 \leftarrow a(1)$

Figure 10.4. Function of the Approach using a queue on the Moore machine (Figure 10.1)

The last approach finds all separating sequences and it is proposed in Algorithm 14. Its function on the Moore example captured in Figure 10.5 is very similar to Algorithm 8 described as the first in this section. However, the distinguishing table contains *next* array and *minLength* in a cell in this case. Variable *minLength* is abbreviated to m on the bottom left of a cell. The inputs and values on the right of a cell represent particular entries of *next* array.

	B	C	D
A	0 m 2 a 1 b 1	1 m 1 a 0 b *1*	2 m 0 a -1 b 1
B		3 m 1 a -1 b *3*	4 m 0 a *4* b -1
C			5 m 0 a *5* b *5*

1. 0: $a \leftarrow 1,\ b \leftarrow 1$
2. 1: $a \leftarrow 0,\ b \leftarrow \mathit{1},\ m \leftarrow 1$
3. 2: $m \leftarrow 0,\ b \leftarrow 1$
4. 3: $b \leftarrow \mathit{3},\ m \leftarrow 1$
5. 4: $m \leftarrow 0,\ a \leftarrow \mathit{4}$
6. 5: $m \leftarrow 0,\ a \leftarrow \mathit{5},\ b \leftarrow \mathit{5}$
7. 0: $m \leftarrow 2$

Figure 10.5. Function of Algorithm 14 on the Moore machine (Figure 10.1)

The first six steps shown in Figure 10.5 handle each states pair. For example, step 4 distinguishes states B and C (index 3). It tries the input a at first. The states transfer to the same pair of states on a so the algorithm leaves $next[a]$ with the default value. The input b distinguishes the states, $\lambda(B, b) = \lambda(A) \neq \lambda(D) = \lambda(C, b)$, therefore $next[b]$ is marked and *minLength* is set to 1. The index of cell is used as the mark hence $next[b]$, or simply b, gets 3. The last step goes through the queue *Distinguished* and sets *minLength* using provided backward links stored in *Link*. The cell 0 is the last one without set *minLength* because the pair of states A and B is the only one with shortest separating sequence of length 2.

10.2 State characterization

General testing methods generating several test sequences are based on state verification by a set of separating sequences, see the W-method or the HSI-method in Section 6.1, for example. The performance of methods also relies on choosing such a set that is not unique in general. Unfortunately, the testing methods do not pose a design method for state characterizing set. Therefore, this section proposes methods to obtain a state characterizing set as well as a characterizing set, to reduce the set and to create a family of harmonized state identifiers. All methods are based on separating sequences. Shortest separating sequences constructed in the previous section are shown in Table 10.1 and they will be used for a description of proposed methods.

Pair of states	A,B	A,C	A,D	B,C	B,D	C,D
Separating sequence	ab	b	ε	b	ε	ε

Table 10.1. Shortest Separating Sequences of the Moore machine (Figure 10.1)

10.2.1 SCSet

The set of output sequences to state characterizing set W_i of state s_i must be different for each state $s_j \neq s_i$ according to Definition 3.30. Consequently, there is a separating sequence for each s_j in W_i. So, it is sufficient to take all constructed separating sequences related to given state s_i to form its SCSet. Table 10.2 shows state characterizing set of each state of our sample machine.

State	State Characterizing Set
A	ε, b, ab
B	ε, b, ab
C	ε, b
D	ε

Table 10.2. State Characterizing Sets of the Moore machine (Figure 10.1)

Definition 3.32 states the notion of characterizing set W. It can be easily obtain as union of all constructed separating sequences. Then, there is a separating sequence for each pair of states in W so W distinguishes every two states. In our example, the state characterizing set of state A (or B) is also characterizing set of the machine M, i.e. $W = \{\varepsilon, b, ab\}$.

10.2.2 Reduction

Neither state characterizing set nor characterizing set is unique. We want to construct minimal length checking experiment so we need to minimize SCSet and CSet as well. Nevertheless, there are two criteria. One can minimize the number of sequences and/or the lengths of sequences in the set. It is very hard optimization problem. The former minimization approach leads to finding state verifying sequence (Definition 3.29) which does not have to exists. We have decided for the latter approach as the first step. The set of minimal length separating sequences was formed in the previous section. The second step tries to reduce the number of sequences in the set.

We proposed the reduction method in [So14]. A similar approach was introduced in [Mi10] however it removes only sequences that are proper prefixes of another ones from the set. Our reduction method is based on knowledge of design of initial characterizing

set. We form a characterizing set as union of shortest separating sequences. Let u be a shortest separating sequence of states s_i and s_j in characterizing set W. There is a possibility that W contains another sequence v that is separating sequence of states s_i and s_j as well. If we find such a v, then u can be removed from W. We pass through the set and analyze which sequence distinguishes which states pair. The key idea is sorting of sequences by the length and then go from longest. Sequence v must be at least long as u, i.e. $|v| \geq |u|$.

Our method [So14] passes the entire set twice: at fist from longest and the second time from shortest sequences. We found that the second passage through is not needed because sequences with the same length are the main issue. Imagine a set of the same length sequences G and a set of states pairs $V \subseteq S \times S$. Each sequence $u \in G$ distinguishes some states pairs of V and all sequences of G together distinguish all states pairs of V. Finding smallest subset of G that distinguish all states pairs of V is NP-complete problem [Ho06]. The described task is equal to NP-complete Set Cover Problem. Therefore, we propose a suboptimal approach only.

The new method is described in Algorithm 15. The sequences of given characterizing set W are sort and grouped by the length. We start with the group G of longest sequences. Two sets of pair indexes are initialized for each sequence $u \in G$. All_u contains all yet undistinguished pairs that u separates. $Last_u$ is a subset of All_u and consists of states pairs that are distinguished by the last input symbol of u. Pairs of $Last_u$ are called u's related pairs. If $Last_u$ is empty, there are $v_i \in W, |v_i| > |u|$, that distinguish all u's related pairs of states and so u can be removed. The heuristic then comes (line 10 in Algorithm 15). We choose a sequence w of G that remains in W. The sequence with most related states pairs is chosen. If there are more sequences with the same $|Last_u|$, they are sorted by $|All_u|$ and even if they are equal, alphabetic order decides. This heuristic assumes that a sequence with more related or all distinguished pairs has better chance to replace a sequence with less pairs. After the best sequence u is extracted from G sequences remaining in G update their *Last* and *All* sets due to u distinguishes some pairs.

Algorithm 15. Reduction of characterizing set W in the number of sequences

```
   input: W - characterizing set to reduce
 1 Distinguished ← ∅
 2 foreach G_len = {u ∈ W | |u| = len} and len ← n to 0 do
 3     foreach u = vx ∈ G_len, x ∈ X do
 4         All_u ← {cell(s_i, s_j) | λ(s_i, u) ≠ λ(s_j, u)} \ Distinguished
 5         Last_u ← {idx ∈ All_u | λ(s_i, v) = λ(s_j, v)}
 6         if Last_u is empty then
 7             erase u from W and G_len
 8     while G_len is not empty do
 9         sort G_len by |Last_u| and |All_u|
10         w ← G_len.pop_best()
11         Distinguished ← Distinguished ∪ All_w
12         foreach u ∈ G_len do
13             All_u ← All_u \ Distinguished
14             Last_u ← Last_u \ Distinguished
15             if Last_u is empty then
16                 erase u from W and G_len
```

Algorithm 15 runs in $O(n^5)$ like the one in [So14]. W contains at most $n(n-1)/2$ sequences at the beginning of the reduction. Let w be the longest separating sequence in W. Output sequences are compared for each pair of states. That is $C = n(n-1)/2$ comparisons and one comparison takes at most $|w| < n$. Thus, time complexity is $O(|w| \cdot |W| \cdot |C|) \subseteq O(n \cdot n^2 \cdot n^2) = O(n^5)$. Notice that the heuristic part is not so time consuming. In each 'while' cycle (line 8) at least one sequence is removed from G and the inner 'for' cycle (line 12) takes the time of $|G|$. Together with sorting algorithm $O(n \log n)$, it is $O(n^2 + n \cdot n \log n)$.

Adaptation of Algorithm 15 to reduce state characterizing set is done as follows. Input to the algorithm is a state characterizing set W_i of state s_i. State s_i is fixed on lines 4 and 5. All_u contains states s_j that are distinguished from the state s_i; line 4 is adjusted to '$All_u \leftarrow \{s_j \mid \lambda(s_i, u) \neq \lambda(s_j, u)\} \setminus Distinguished$'. Time complexity decreases to $O(n^3)$, because of $|W_i| < n$, $|C| < n$ and $|w| < n$.

Far more complex machine than the Moore one in Figure 10.1 would be needed to show all properties of Algorithm 15. Nevertheless, we continue with our example. Table 10.2 shows initial state characterizing sets to reduce. Output responses are listed for each state in Table 10.3.

Moore machines require special handling due to outputs by states. The empty string is used to obtain the output of a state. Testing methods usually concatenates characterizing sets to a nonempty sequence u. The last output symbol on such a sequence u is equal to the one obtained to the empty string applied after u, i.e. in Moore machines it holds $\lambda(\delta^*(s_0, v), x) = \lambda(\delta^*(s_0, u), \varepsilon)$ where $u = vx, x \in X$. Thus, the empty sequence is included in a test implicitly. Therefore, we do not count with states distinguished by the empty string ε. When it is needed, the empty sequence precedes another sequence of characterizing set.

State	ε	b	ab	Reduced SCSet
A	0	0	00	b, ab
B	0	0	01	ab
C	0	1	00	b
D	1	0	10	ε

Table 10.3. Reduction of SCSets of the Moore machine (Figure 10.1)

Table 10.3 shows reduced state characterizing set for each state. For example, the output of state B to ab is 01 which is unique over all states. Therefore, the $Last_b$ and $Last_\varepsilon$ are empty and b, ε are removed from W_B. The algorithm found a state verifying sequence of state B by reduction of W_B. Sequences b and ab are sufficient to distinguish all states so reduction of characterizing set produces $W = \{b, ab\}$. On the other hand, there is a distinguishing sequence that could be found if the order of the input alphabet was different. Algorithm 8 would construct the separating sequence bb for states A and B. Then the sequence εbb is distinguishing sequence, i.e. it produces a different response for each state.

10.2.3 HSI

Harmonized state identifiers are a special case of state characterizing sets. According to their Definition 3.31 each pair of harmonized state identifiers has to contain common prefix that separates related states. Therefore, it is sufficient to take each separating sequence related to given state s_i as in Section 10.2.1. We reduce such a formed set by removing proper prefixes of other sequences. In other words, we take maximal sequences

of pref(W_i) to create harmonized state identifier H_i of state s_i. Moreover, we add the empty string at the beginning of any separating sequence of the set explicitly if we handle with a Moore machine. A family of harmonized state identifiers of our Moore sample is shown in Table 10.4.

State	Harmonized State Identifier
A	εb, ab
B	εb, ab
C	εb
D	ε

Table 10.4. Harmonized State Identifiers of the Moore machine (Figure 10.1)

10.3 Other input sequences

Testing methods use special input sequences for state verification. Characterizing sets are described in the previous section. We proposed design methods for other sequences in [So14]. There are algorithms for preset and adaptive distinguishing sequence, state verifying sequence and homing sequence. Construction of PDS is PSPACE-complete [Le94]. Therefore, we posed a heuristic approach. ADS is created in polynomial time (in the number of machine states) in price of a possible suboptimal solution, i.e. a found sequence does not have to be the shortest. Methods for SVS and homing sequence are stated as exponential algorithms like the PDS algorithm. However, there is a polynomial method for homing sequence that can produce a suboptimal solution [Gi62].

A transfer sequence to state s is needed to apply before verification of transitions from s can start. In this section we describe the straightforward methods creating *state* and *transition cover* set of sequences and then we introduce the structure for handling prefix-closed set.

10.3.1 State cover

State cover is a set of transfer sequences according to its Definition 3.34. A transfer sequence for state s is a defined sequence from the initial state to state s. Algorithm 16 creates a set of shortest transfer sequences for each state. It is simple breadth-first search using array *covered* to mark visited nodes.

Algorithm 16. Design method of state cover set SC

```
1  SC ← {ε}
2  covered[s0] ← true
3  FIFO.push(s0, ε)
4  while FIFO is not empty do
5      (s, u) ← FIFO.pop()
6      foreach x ∈ X do
7          nextState ← δ(s, x)
8          if not covered[nextState] then
9              SC ← SC ∪ {ux}
10             covered[nextState] ← true
11             FIFO.push(nextState, ux)
```

10.3.2 Transition cover

Transition cover is a set of defined sequences that contains each transition according to Definition 3.35. Algorithm 17 creates a state cover SC and each sequence of SC extends with each input symbol so a transition cover set is produced. It employs breadth-first search as state cover set's Algorithm 16. It uses array *covered* to mark visited nodes as well. The difference is adding a sequence to the set; it is before a checking if the next state is visited for design of transition cover (line 8 in Algorithm 17) and it is after the checking for state cover (line 9 in Algorithm 16).

Algorithm 17. Design method of transition cover set TC

```
1  TC ← {ε}
2  covered[s0] ← true
3  FIFO.push(s0, ε)
4  while FIFO is not empty do
5      (s, u) ← FIFO.pop()
6      foreach x ∈ X do
7          nextState ← δ(s, x)
8          TC ← TC ∪ {ux}
9          if not covered[nextState] then
10             covered[nextState] ← true
11             FIFO.push(nextState, ux)
```

10.3.3 Prefix set

Definition 4.1 states that test suite is a prefix-closed set and only maximal test cases are needed for testing. For a representation of test suite one can extend a set structure provided as a basis for most programming languages or design new structure. We decided to the latter case because only a few functions on the structure are needed. Particularly, a sequence can be inserted into the set, one can ask whether the set contains a sequence and all maximal sequences can be easily obtained. We call such a structure a prefix set.

Definition 10.1. A *prefix set* of sequences is a tree such that label of each node is an input symbol but the root is labeled with the empty string ε. Paths from the root represent sequences of the prefix set.

There are plenty of options how to implement prefix set. If $|X|$ is big and a lot of sequences are assumed to be in prefix set then a node of prefix set can be represented by an associative array (map in C++ terminology). Key and value would be an input symbol and a pointer to the next node, respectively.

Another approach is representation of node by a triplet of an input symbol, i.e. label of node, and two pointers to other nodes. The first pointer is to a child, or a successor, in the tree and the second one is to sibling, i.e. a node that has the same parent in the prefix set. The approach handles prefix set by levels of its tree structure. For example, if one wants to access the last child of a node, all children have to be passed. Therefore, this approach is suitable when low branching factor is assumed. Nonetheless, the required functionality is very easy to implement. Linked structure enables simple removal of sequences. This is a big advantage of the approach.

10.4 Testing Methods

The methods creating test suite with several sequences use sets P and R as they are defined in Section 6.1, i.e. P is the transition cover (Section 10.3.2) if there is no extra state and R are then sequences from TC that are not in the state cover (Section 10.3.1). Some of them employ rather a testing tree, see Figure 9.2 for example.

Description of methods' implementation is proposed as a technical documentation without explanation of all details. Therefore, a study of provided pseudocodes is required for fully understanding of some methods. The H-, SPY- and C- methods get the most focus owing to their complexity. It is needed to specify especially a choice of separating sequences in the H-method, a way of handling of convergent sequences in the SPY-method and updating of confirmed set in the C-method. An implementation of our new testing method, the M-method, concludes this section. It also includes a proposal of two suboptimal approaches, the M^a-method and the M^g-method, that approximate the M-method.

10.4.1 PDS-method

The PDS-method defined in Section 6.1.1 states creating of a test suite T as appending a preset distinguishing sequence to each sequence of P. We obtain a PDS using the method described in [So14]. Each sequence from P is then concatenated with the PDS and inserted into the prefix set PS. Test suite is formed by maximal sequences of PS. If there is no PDS, the method produces empty test suite T as a sign of PDS's absence.

If given machine is type of Moore, i.e. the PDS starts with the empty string ε to obtain the output of the initial state, the following treatment is employed. The empty sequence is erased from the beginning of the PDS because the last output of a sequence u of P is the same as it would be obtained by applying the empty sequence after u. Notice that P contains ε so the entire PDS is applied in the initial state.

10.4.2 ADS-method

The ADS-method (Section 6.1.2) is equal to the PDS-method in the implementation point of view. Use of an adaptive distinguishing sequence instead of PDS is the only difference. We obtain an ADS by polynomial algorithm but the ADS does not have to be the shortest one, see [So14] or [Le94].

10.4.3 SVS-method

Section 6.1.3 defines design of a test suite using the SVS-method. An implementation is similar to the PDS-method in aspect of handling Moore machines and use of a prefix set. If there is no state verifying sequence of a state s, reduced state characterizing set W_s (Section 10.2.2) is used for state verification. The method always creates a test suite T but the number of states without SVS is attached to T for a comparison.

10.4.4 W-method

The W-method is the oldest method producing an m-complete test suite for each resettable machine. It is formally defined in Section 6.1.4 and implemented in a straightforward manner. Each sequence of P is concatenated with each sequence of reduced characterizing set W. Creating characterizing set and its reduction in the number of sequences are described in Section 10.2. A prefix set is again employed to remove redundant tests that are contained in longer tests. If a given machine is type of Moore and there is an extra state or W contains the empty sequence, one test sequence of T needs to start with ε to distinguish the initial state.

10.4.5 Wp-method

Characterizing set W and state characterizing sets W_i are used in the Wp-method. Each W_i must be subset of W according to the definition of the method in Section 6.1.5. Therefore, we reduce each state characterizing set using Algorithm 15 at first. Characterizing set is then created as union of reduced state characterizing sets W_i. We use a prefix set to obtain only maximal sequences of such union of W_i. Test sequences are formed according to the definition and only maximal sequences are selected. If we deal with a Moore machine, the same condition as for the W-method holds. If there is an extra state or W contains the empty string ε, a test sequence has the empty string ε at the beginning.

10.4.6 HSI-method

The HSI-method stated in Section 6.1.6 is very similar to the W-method from the implementation point of view. The difference is the use of harmonized state identifiers H_i in place of characterizing set. Creating of harmonized state identifier is proposed in Section 10.2.3. The condition for Moore machines is adjusted that if H_0, i.e. state identifier of the initial state, contains the empty sequence or there is an extra state, then one test starts with the empty sequence ε.

10.4.7 H-method

Section 6.1.8 provides a description of the H-method. This method uses harmonized state identifiers to state verification. However, it chooses separating sequences for two states on-the-fly unlike the HSI-method that chooses separating sequences in advance.

At first, all shortest separating sequences are obtained using Algorithm 14. They are stored in states pairs array L. The length of shortest separating sequence for each states pair is very important. We use it to estimate the number of needed symbols for separation of two states.

A testing tree is constructed such that each sequence of P is a path from the root. The root represents the initial state and has the output of the initial state as label if we deal with a Moore machine. Edges are labeled by an input symbol. A node $n_i = (s_i, y_i)$ of testing tree having a parent node $n_j = (s_j, y_j)$ contains a state s_i and an output symbol y_i as label. The state and the output are obtained by applying the input x of the edge from the parent in n_j, i.e. $s_i = \delta(s_j, x)$ and $y_i = \lambda(s_j, x)$. Moreover, nodes contain a variable to mark the best input for separation of two nodes in question.

Nodes of testing tree are divided into groups of SC nodes that are reached by state cover set's sequences and the others, called $Extra$. This helps to pass steps 2-4 of the H-method. The testing tree is created in breath first search as the transition cover (Section 10.3.2). Nodes are added in the list of SC nodes or $Extra$ nodes as soon as they are created. The lists are then passed according to the steps of the method.

A separating sequence has to be chosen if two given nodes are not distinguished. Both subtrees of given nodes are compared at first whether they are different and so the nodes are distinguished. During comparison of subtrees we get estimate of how many input symbols are needed to append to distinguish two related nodes. Then, if it is found out that given nodes are not distinguished, we simply take minimal estimate and append related separating sequence.

Our simple estimation function is proposed in Algorithm 18. The function assumes that the first given node n_i has a leaving edge on given input x and the second node n_j does not provide such an edge. Therefore, the function returns an estimate of 1 if x

distinguishes given nodes; x has to be appended to n_j. There is a possibility that x can not be used as the first symbol of a separating sequence of the nodes. That is, states of the nodes transfer to the same state or remain in the same states pair on the input x. In such case, a mark of inapplicability must be returned. We use the number of states (line 4) because each shortest separating sequence is shorter than this value and we find a minimal estimate. If x can be the initial symbol of a separating sequence, doubled the length of shortest separating sequence of the next states pair increased by 1 for appending x in n_j is returned as an estimate. Note that the value is really an estimate because the next node of n_i on x can have the related separating sequence appended (or its prefix) so the number of needed symbols is less than the estimated value.

Algorithm 18. Estimation of needed symbols to distinguish given nodes

```
1 Function getEstimate(n_i, n_j, x)
      input  : n_i = (s_i, y_i), n_j = (s_j, y_j) - test nodes to estimation
               x - the first input symbol of estimated separating sequence
      data   : L - pairs array of all separating sequences
2     idx ← index of (s_i, s_j) in L
3     nextIdx ← L[idx].next[x]
4     if nextIdx = −1 then return |S|    // x can not distinguish nodes
5     if nextIdx = idx then return 1     // x is distinguishing input
6     return 2 · L[nextIdx].minLength + 1
```

Algorithm 19 describes the recursive function that checks whether given nodes are distinguished and it marks an input that should be used as the initial symbol of their separating sequence. The function compares subtrees of nodes that are to be distinguished according to the H-method. If the machine is the type of Moore, the outputs of nodes are checked before the function is called. For each common input symbol the outputs of successors are checked whether they differ. If they do not, the comparison is recursively called on the successor nodes. Notice that the states of successors must be different; a pair of equal states can not be distinguished. Some nodes do not have a leaving edge on certain input symbol. For such a pair of nodes we get an estimate of appending a common separating sequence in these nodes using Algorithm 18. If both nodes do not have an edge on certain input, one is added to the estimate (line 19) due to the design of the estimate function. We want to reduce branching of testing tree, i.e. minimize the number of tests. Therefore, we prefer appending in successor nodes. The best estimate so far is updated even if it is equal to the estimate of successor, see line 11. In the end, the function marks the best input (line 21).

A separating sequence of nodes from lists SC and $Extra$ to be distinguished is created using marked inputs after Algorithm 19 finds that the nodes are not distinguished. The following rule is recursively applied. If the best input is common for both nodes, the successor nodes are considered. Otherwise a separating sequence starting with the best input is appended to both nodes. The shortest separating sequence is derived from provided pairs array L. There can be more shortest separating sequence. Then the one with the first symbol coinciding with a leaving edge of one node is chosen or the lexicographically lowest sequence is employed if neither node has such a leaving edge.

The last step is to arrange a test suite. It is the same process as obtaining maximal sequences from prefix set. That is, sequences of paths from the root node to leaves form test cases.

Algorithm 19. Finding the best way to distinguish given nodes

```
1  Function areTestNodesDistinguished(n_i, n_j)
       input  : n_i = (s_i, y_i), n_j = (s_j, y_j) - test nodes to compare
       output: 0 if given nodes are distinguished,
               the number of inputs needed to distinguishing otherwise
2      minEst ← (|S|, ε)    // a pair of minimal value and related input
3      foreach x ∈ X do
4          if n_i has an edge on x then
5              if n_j has an edge on x then
6                  if λ(s_i, x) ≠ λ(s_j, x) then return 0
7                  if δ(s_i, x) = δ(s_j, x) then continue
8                  n'_i, n'_j ← the next nodes of n_i, n_j on x
9                  est ← areTestNodesDistinguished(n'_i, n'_j)
10                 if est = 0 then return 0
11                 if minEst.first ≥ est then minEst ← (est, x)
12             else
13                 est ← getEstimate(n_i, n_j, x)
14                 if minEst.first > est then minEst ← (est, x)
15         else if n_j has an edge on x then
16             est ← getEstimate(n_j, n_i, x)
17             if minEst.first > est then minEst ← (est, x)
18         else
19             est ← getEstimate(n_i, n_j, x) + 1
20             if minEst.first > est then minEst ← (est, x)
21     mark minEst.second as the best input to distinguish given nodes
22     return minEst.first
```

10.4.8 SPY-method

The SPY-method reduces test branching using knowledge of verified transition. This makes the method one of the best methods for generating test suite with several sequences. Nevertheless, it uses harmonized state identifiers obtained in advance so the results are comparable to the H-method, see Section 11.3.

A formal description is in Section 6.1.11 or in original paper [Si12]. The method deals with a partition of sequences derived by their convergence. The authors suggest to use Union-Find structure to handle partitions so time complexity of the method remains polynomial in the length of test suite T.

We found out that only blocks containing sequences of SC are needed due to the design of the method and thus we do not have to handle with all blocks of the current partition. The idea of implementation is similar to the H-method. At first, a testing tree is created according to the first step of the SPY-method. That is, paths from the root node are sequences of $SC \circ H_i$. We use array $confirmedNodes[s_i]$ to handle the block of partition Π that contains sequences convergent with the transfer sequence $u \in SC$ to state s_i. During the creation of the testing tree, *confirmedNodes* arrays are filled with singletons of nodes that relate to transfer sequences of state cover set SC. Moreover, unverified transitions are stored when they are touched in breath-first search; the process of creating testing tree is similar to Algorithm 17 however nodes of SC are appended with related H_i.

Selection of the convergent sequence that is extended by given separating sequence is done as Section 6.1.11 states. Given transition (s, x), we pass *confirmedNodes*$[s]$ and then *confirmedNodes*$[\delta(s, x)]$. For each node in particular array we check whether a path from the node to a leaf is a prefix of the sequence w that is to be added. It is clear that if w is a prefix of a path from the node, nothing is added. If there are more maximal sequences (paths) that are prefixes of w, the one minimizing added symbols is chosen. Otherwise, i.e. no path is a prefix of w, the transfer sequence from SC is extended because the test suite grows least. During appending the sequence w to a node of *confirmedNodes*$[\delta(s, x)]$, the closure procedure is performed. If a prefix of w consists of verified transitions, each node reached by such a prefix is added to related *confirmedNodes* array. Nodes reached by appending to a node of *confirmedNodes*$[s]$ do not have to be checked because the transition (s, x) is still unverified so the successor nodes cannot be in *confirmedNodes* arrays. After each separating sequence of particular H_i is applied, the transition becomes verified and the *confirmedNodes* arrays are updated as follows. Each successor node of a node n_i of *confirmedNodes*$[s]$ reached by x from n_i is added to *confirmedNodes*$[\delta(s, x)]$ and its successors are also added (to related *confirmedNodes* array) if there is a verified path to them. As an unverified transition occurs in a path from n_i, no successor nodes are added to *confirmedNodes* arrays.

Test cases are again formed from the maximal sequences, i.e. paths from the root to leaves. Our approach with *confirmedNodes* arrays remains polynomial in the length of test suite. Handling partition and the closure function are the only differences from the authors' approach. The length of test suite T, i.e. len(T), generated by aforementioned methods for resettable machines can be bounded by $O(np^{m-n+1} \cdot (n-1) \cdot (m+n-1))$ because P (an extended SC defined in Section 6.1) has up to np^{m-n+1} sequences of length at most $m = |Q|$ and each of them is appended with up to $(n-1)$ separating sequences that are shorter than $n = |S|$. In addition, the number of extra states is assumed to be strictly less than n, i.e. $n \leq m < 2n$ so the bound of $O(n^3 p^{m-n+1})$ is obtained. The SPY-method can produce test cases with the length greater than $(m+n)$ but the number of sequences is reduced and the bound holds.

10.4.9 C-method

Section 6.2.7 proposes a formal description of the C-method. There is a quite straightforward algorithm however two difficulties need to be resolved. The first one is the maintaining of a confirmed set $C(t)$ and the second problem concerns output-confirmed sequences that are needed for the step 2.(ii).

An approach to the former problem is proposed in [Si08]. The authors found out that it is sufficient to maintain the shortest verified sequences for each state, i.e. there is no verified proper prefix of them. We use a prefix set ps_i (see Definition 10.1) for each state s_i. It handles all shortest verified sequences and a new function over this structure is employed. When a sequence v becomes verified in a state s_i, it replaces all sequences of the prefix set ps_i which v is a prefix of. That is, v is a maximal sequence in ps_i after removal of sequences. There is also a possibility that no sequence is removed. Suffixes of removed sequences are then added to the prefix set of state $\delta^*(s_i, v)$. This is comprised in function update() and function processNewlyConfirmed() proposed below.

We store a particular state with creating (checking) sequence t_i. A testing tree with one path, or a linear linked list of test nodes, is created. Each node (reached by u, a prefix of t_i) contains related *state* ($s = \delta^*(s_0, u)$), a confirmation mark *isConfirmed*, i.e. whether u is s-confirmed in t_i, and an input x leading to the next node. Inputs of all nodes then form sequence t_i and a checking sequence at the end of design. When

u becomes is s-confirmed, mark of related node n_i is set and n_i is added to the array *confirmedNode*[s] that represents nodes of convergent sequences. We say n_i with state s is confirmed as the sequence to n_i is s-confirmed. A queue *newlyConfirmed* stores confirmed nodes whose leaving sequences need to be checked for a possible confirmation of successor nodes.

An adaptive distinguishing sequence, or distinguishing set of verifying sequences d_i, is obtained by polynomial algorithm as in the ADS-method (Section 10.4.2).

Algorithm 20. Updating verified sequences from confirmed node

```
1  Function update(node)
       input : node - a confirmed test node
2      v ← the shortest verified sequence from node
3      if |v| = 1 then the transition (node.state, v) is verified
4      shorten verified sequences in the prefix set of node.state to v
5      foreach convNode ∈ confirmedNodes[node.state] do
6          if there is a path from convNode with the label of v then
7              newConfNode ← the node reached by v from convNode
8              if not newConfNode.isConfirmed then
9                  newConfNode.isConfirmed ← true
10                 newlyConfirmed.push(newConfNode)
```

The function update() in Algorithm 20 is called on the last confirmed node preceding newly confirmed one in step 1 of the C-method (Section 6.2.7) and on node related to state s with unverified transition in step 2. The next confirmed node nc is found (line 2) and the number of unverified transition decreases if a sequence v from *node* to nc is a transition only. Another use of update() is in function processNewlyConfirmed() described in Algorithm 21. There is update() called on a newly confirmed node which a verified sequence not covered in the related prefix set leads from.

Algorithm 21. A check of newly confirmed nodes and their verified sequences

```
1  Function processNewlyConfirmed()
2      while newlyConfirmed is not empty do
3          node ← newlyConfirmed.pop()
4          confirmedNodes[node.state].push(node)
5          u ← the longest common prefix of the path from node and
6                a verified sequence of node.state
7          v ← the shortest verified sequence from node, or
8                ε if there is no confirmed successor node
9          if u is a maximal sequence of node.state's verified prefix set and
           (|u| ≤ |v| or v = ε) then
11             newConfNode ← the node reached by u from node
12             if not newConfNode.isConfirmed then
13                 newConfNode.isConfirmed ← true
14                 newlyConfirmed.push(newConfNode)
15         else if v ≠ ε then update(node)
```

Function processNewlyConfirmed() looks at successors of a newly confirmed *node* and confirms one if there is a verified sequence to it, or it adds a verified sequence to the prefix set of *node*'s state if a successor is found to be confirmed. Conditions in the function ensure that only node reached earlier (closer successor) is handled and a verified sequence is not added to the related prefix set if the set contain a nonempty prefix of such a sequence. In other words, the function finds the first confirmed successor *cs* and a node *nv* reached by a verified sequence of the prefix set of *node*'s state. The function then handles the one of *cs* and *nv* which is reached first. There is a possibility that one or both nodes are not found, then no action is performed.

The confirmed set is updated using the function update() followed by a call of the function processNewlyConfirmed() after each extension of the constructed sequence t_i, i.e. in the end of steps 1 and 2. We store a pointer *lastConf* to the last confirmed node. It is used for update() call in step 1 and it can be shifted in processNewlyConfirmed() if a successor node is confirmed. However, the pointer *lastConf* has to point before the node confirmed in step 1, i.e. it points to a node within the sequence u, so *lastConf* is shifted in processNewlyConfirmed() conditionally. Step 1.(i) defines u as the shortest unverified prefix of t_i that can be appended by its related distinguishing sequence $d_{\delta^*(s_0,u)}$.

Algorithm 22. A check of possibly reduction of verifying sequence

```
1  Function getVerificationSequence(s, x)
       input  : (s, x) - a transition to verify
       output: a prefix of d_{δ(s,x)} needed for verification
2      s' ← δ(s, x)
3      if x is not a prefix of d_s then return d_{s'}
4      v ← d_s = x · v
5      U ← {ε}
6      for q ∈ S \ {s'} do
7          u ← the shortest prefix of v such that λ*(s', u) ≠ λ*(q, u), or
8                v if s' and q are indistinguishable by v
9          if u ≠ v then
10             if there is a path u from a cn ∈ confirmedNodes[q] then
11                 continue    // distinguished by q-output-confirmed
12         u_q ← the shortest prefix of d_{s'} such that λ*(s', u_q) ≠ λ*(q, u_q)
13         U ← U ∪ {u_q}
14         if u_q = d_{s'} then break   // entire d_{s'} is needed so stop searching
15     return the longest sequence of U
```

A shortest verified sequence in step 2.(i) is found using breadth-first search on the machine's state-diagram from the current state $\delta^*(s_0, t_i)$. Algorithm 22 describes function getVerificationSequence() that returns a sequence w verifying a given transition according step 2.(ii) of the C-method. It is a straightforward implementation of the design step with employing array *confirmedNodes*. Each state q is first tried to distinguish from s' by v, a suffix of d_s. If it is distinguished, the distinguishing prefix of v is checked to be q-output-confirmed. Otherwise (or it is not q-output-confirmed), the shortest prefix u_q of $d_{s'}$ distinguishing q and s' is determined. If a state q is found to need the entire distinguishing sequence $d_{s'}$, the search among all states can be stopped (line 14) and $d_{s'}$ is returned. Otherwise, the longest prefix of $d_{s'}$ over all determined u_q's is chosen.

10.4.10 M-method

The M-method is a NP-complete problem in general. However, an optimal solution can be found for small instances. We denote the method's implementation that produces optimal solution the *M*-method*. We further propose two polynomial suboptimal approaches. We call them the *M^a-method* and the *M^g-method*. The approaches are similar to design of minimal spanning tree by Prim's and Kruskal's algorithms, but a directed path is constructed in this case.

All approaches create an adaptive distinguishing sequence, if it exists, at first. We use the algorithm from [Le94] that is polynomial in the number of states and it can produce a suboptimal ADS.

M*-method

We create set U according to its definition, see Section 8.1. It takes $O(pn)$ time, where $p = |X|$ and $n = |S|$, because each transition has test sequence in U and concatenation with related distinguishing sequence d_i takes a constant time. Notice that $|U| = pn + 1$ because verification of the initial state s_0 must be included. Each element of U forms a node in a graph G.

We assume that not all sequences can overlap so connecting shortest sequences are needed. Therefore, shortest sequences between each states are computed at first. We use Floyd algorithm that runs in $O(n^3)$ [Fl62]. Then we calculate the costs of edges in G. A test sequence $(s_i, t_i) \in U$ can overlap with up to $|t_i|$ other tests. One comparison of such sequences takes $O(|t_i|)$. Let t be the longest sequence of U, i.e. $|t|$ is the length of adaptive distinguishing sequence plus one. Thus, checking of overlapping of all test sequences is in $O(|t|^2(np + 1))$. Filling the rest of costs, i.e. edges related to non-overlapped tests, requires to pass through all edges which takes $O((np + 1)^2)$. Therefore, time complexity of computing costs is $O(n^3 + |t|^2(np + 1) + (np + 1)^2)$.

An ILP task is then formed. We use optimization tool by Gurobi[1]) for solving the given ILP task. The last thing is to construct a checking sequence from a solution provided by Gurobi solver. We start in node $u_0 = (s_0, d_0)$ and find a position j such that $x_{0j} = 1$. According to the related cost c_{0j}, a sequence is appended to the sequence created so far, i.e. the empty sequence at the beginning. If $c_{0j} \leq |d_0|$, the first c_{0j} symbols of d_0 are added. Otherwise, we append the entire d_0 followed by shortest connecting sequence w_{kj}; $|w_{kj}| = c_{0j} - |d_0|$. The sequence constructed so far reaches a node u_j which the related test segment is to be applied in. We set i with a value of j and focus on row i. A new index j is again found such that $x_{ij} = 1$. We append the sequence related to u_i of the length of c_{ij}. Then we update i by j and look at row related to the next node. We continue this way until all nodes are passed, i.e. the index j of the next node is 0. Thus, a checking sequence is created.

M^a-method

The first heuristic approach is similar to the C-method. We call it the M^a-method due to the fact that it gradually appends ADS (or its suffix) at the end of (checking) sequence created so far. In contrast to the C-method, the M^a-method does not employ confirmed set, it only stores which test segment was used. More specifically, we pass through the created sequence and if there is an unverified transition and the distinguishing sequence d_i of the next state can be applied then certain suffix of d_i is appended to the created sequence. It is equal to step 1 in the C-method (Section 6.2.7) but instead of checking t_i and u whether are in $C(t_i)$ the last transition of u has to be unverified

[1]) `www.gurobi.com`

in the meaning that the related test segment was not used so far. If we reach the end of created sequence, a shortest transfer sequence to state with unverified transition is added. Then the related test segment is append and checking of possible overlapping starts again.

This implementation does not need neither set U nor a formulation as ATSP using costs c_{ij}. Time complexity is equal to the C-method, i.e. polynomial in the length of checking sequence, but it is generally faster because confirmed set does not need to be handled. Moreover, it gives better results than the C-method in most cases, see Section 11.2.

M^g-method

The M^a-method optimize locally so the solution can be far from optimum. We thus propose another suboptimal approach called the M^g-method that tries to approximate an optimal solution from the global point of view. It chooses best connections in G till all test segments are included.

More precisely, the M^g-method merges minimal subpaths so that no cycle is produced. The edges of G are sorted by their costs at first. Then, they are passed from the minimal one. An edge is added to the resulting path if it does not create a cycle or a branch of subpaths formed so far. A branch means that the edge connects to an inner node of a subpath. We use arrays $Prev$ and $Next$ to handle the 'branching' constraint. Both have the length of the size of U, i.e. $np + 1$. If an edge (u_i, u_j) is chosen for addition, array are updated as follows: $Prev[u_j] = u_i$ and $Next[u_i] = u_j$. An edge (n_i, n_j) cannot be added if $Prev[u_j]$ or $Next[u_i]$ is set.

Union-Find structure is used for checking of creating of a cycle. Every node has different color in the beginning. If an edge (u_i, u_j) is added, the subpaths containing nodes u_i and u_j get the same color, i.e. we apply union on u_i and u_j. An edge (u_i, u_j) cannot be added if both nodes u_i and u_j have the same color, i.e. they are in the same subpath. However, this implies that this approach cannot create a tour because nodes of the last edge that should be added have got the same colour. We handle this in a way that edges to the initial node u_0 are not considered so a path from u_0 is always created. Both union and find operation have amortized time of a small constant in Union-Find structure. We apply np union operations and at most $(np)^2$ find operation. Time complexity of creating a checking sequence is $O((np)^2)$. However, sorting has to be considered so time complexity of the second approach is $O((np)^4)$ when a quadratic sort is used. This comes with price of possibly shorter created checking sequence.

Mr-method

Section 8.2 proposes modification of the M-method employing reset. We extend all three implementation of the M-method so reset can shorten resulting sequence. The Mr*-method computes optimal checking sequence. State cover SC (Section 10.3.1) is first created. Then, when the cost of an edge between test segment is calculated, a small change appears. If test segments do not overlap, shortest transfer sequence u of SC to a state with unverified transition is compared to a shortest connecting sequence v from the end state to a state with unverified transition. If $|u| + 1 < |v|$, the cost is set to $|u| + 1$. Plus one stands for the use of reset. Notice that the condition forces to create longer tests rather than more tests. When the related edge is chosen to be in resulting sequence, a new test starting with u is created. The Mr^g-method and the Mr^a-method is modified in similar way.

Chapter 11
Experiments

This chapter describes our experiments that show performance of discussed testing method and proposed approaches. At first, we compare our parallel approaches for creating separating sequences. Then we deal with design of checking sequence by the M-method and the last experiment focuses on a comparison of all implemented testing methods. The experiments were performed on computer with the following properties:

- System: MS Windows 7 Ultimate 64-bit
- Processor: Intel Core i7-2620M @ 2.70GHz
- Memory: 6.0 GB RAM
- GPU: NVIDIA GeForce 610M with CUDA specifications:
  - Compute Capability: 2.1
  - Driver Model: WDDM
  - Max. number of threads per block: 1024

11.1 Separating sequences

Section 10.1 proposes algorithms for creation of a shortest separating sequences. Besides the sequential one there are two parallel approaches. Time complexity is derived for each of three methods. Parallel approaches have asymptotic complexity linear in the number of states while the sequential approach has complexity a bit worse, quadratic in particular. This should be empirically proved. Therefore, we propose an experiment comparing all three methods from the perspective of running time.

We generated 10 reduced FSMs with binary input and output alphabets randomly. The machines differ in the number of states. The numbers are multiplies of 100, i.e. the smallest machine has 100 states and the largest one has 1000 states. Each method was running on a machine 8 times. Then the average time was concerned for comparison.

All three methods were compared by their running time measured on CPU. Average values for three particular machines are captured in Table 11.1. Times of the sequential method (Section 10.1.1) are in the second column. The last two columns contains values of parallel approaches, the Straightforward approach and the Approach using a queue respectively. Figure 11.1 shows entire comparison provided by our test.

n	Sequential m.	Straightforward a.	A. using a queue
100	181.704779	83.505128	89.677751
500	7920.376424	649.436105	668.850658
1000	50491.689204	1655.177018	1705.624504

Table 11.1. Design Methods of Shortest Separating Sequences
Average Running Time in Milliseconds

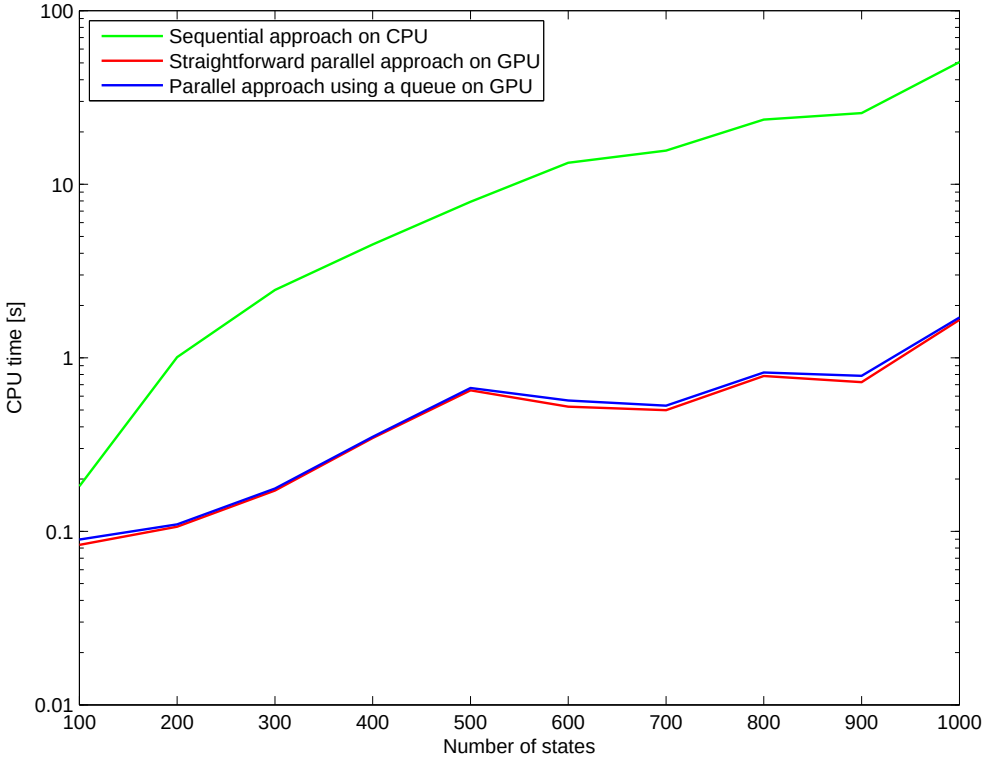


Figure 11.1. Comparison of Design Methods of Shortest Separating Sequences
Average Running Time in Seconds measured on CPU

Obtained results demonstrate acceleration by employing parallelization in searching for shortest separating sequences. Besides measurement of time on CPU we also recorded times of parallel approaches but they were measured on GPU. Several parts of methods were tracked. The first one is time needed to load data to GPU memory. The second time measures processing all kernels of a certain method. In both methods there are some data transfer between kernels. Thus, the processing time also includes copying memory between CPU and GPU. The last captured time is total running time of given parallel method. This time is almost the same as the one measured on CPU. However sometimes it is a quite lower because of GPU initialization. Total time includes reconstruction of found sequences; sequences are transfered from buffers to lists on CPU. Therefore, total time is much bigger than sum of loading and processing times. Some captured values are in Table 11.2 and entire comparison of parallel methods is shown in Figure 11.2.

Method	n	Loading	Processing	Total
Straightforward a.	500	2.456272	11.6082075	581.060669
A. using a queue	500	2.549728	30.0669005	595.017334
Straightforward a.	1000	5.971008	63.8658525	1582.890503
A. using a queue	1000	6.307640	130.3542405	1633.812439

Table 11.2. Parallel Methods - GPU Time in Milliseconds

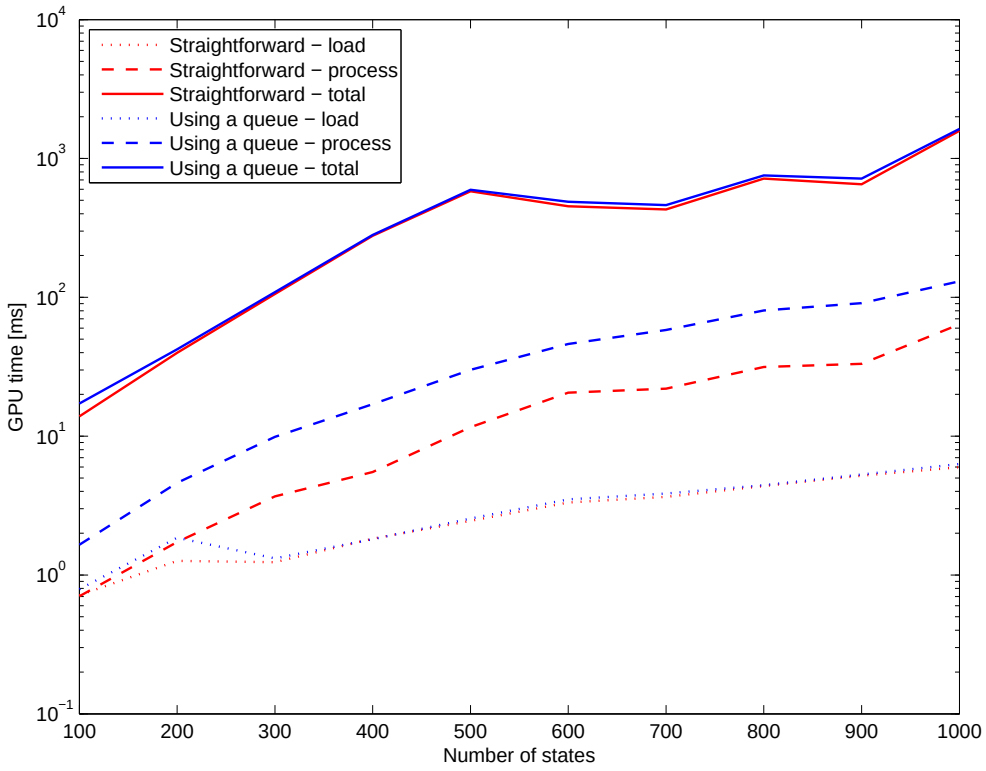


Figure 11.2. Efficiency of the Parallel Approaches measured on GPU

The Approach using a queue were developed to reduce the number of working threads in contrast to the Straightforward approach. Nevertheless, employing of a queue leads to a slower parallel algorithm according to experiment's result captured in Figure 11.2. The largest tested machine has got 1000 states so 499500 threads were used by each parallel approach. Performance of methods is likely to change when one deals with bigger machines and the number of threads is restricted. Then one thread would have to handle more states pairs. Thus, elimination of solved pairs as in the Approach using a queue could be beneficial. Dependence of the approaches performance on the number of threads needs further study. Our experiment only aimed to prove a parallel approach can create shortest separating sequence in time linear to the number of states.

11.2 Checking sequence

We proposed a new testing method, the M-method, in Chapter 8. Its experimental verification and comparison to the state-of-the-art method, the C-method, is described in this section.

The first experiment ran on 60 Moore machines and 60 Mealy machines. All machines have got binary input and output alphabets and they differ in the number of states. There are groups of machines with 10, 20, 30, 40, 50 and 60 states. Each group has 10 samples for both types of machine. Neither the M-method nor the C-method can run on an arbitrary machine so all machines in the experiment are reduced, strongly connected and possess an ADS.

The experiment was to consist of machines with states up to 100. However, we were not able to generate such machines that meet the requirements. Our generator

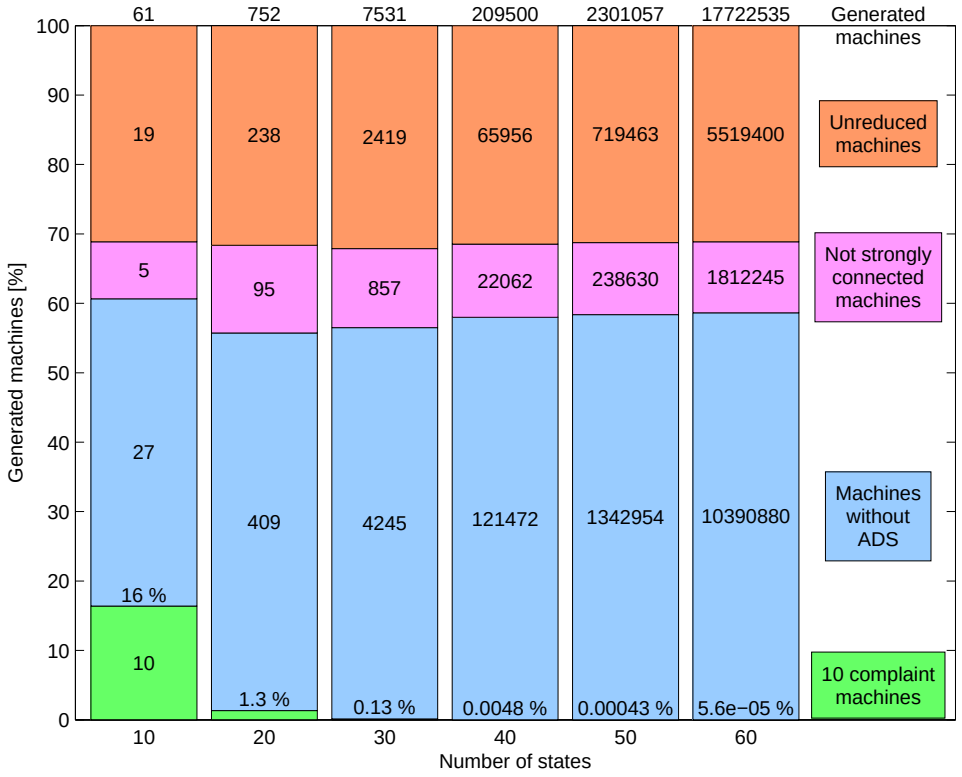


Figure 11.3. Numbers of Generated Moore Machines for the Experiment

described in [So14] randomly creates initially connected machines with given numbers of states, inputs and outputs. We generated machines with binary input and output alphabets until it was found 10 machines for each state group that can be used for the experiment. A generated machines is first checked whether it is reduced. If it is not reduced, another machine is generated. Otherwise, the machine is checked for strongly connectedness. Strongly connected machines are consequently checked whether they have got an ADS. The numbers of generated machines that fail on each condition are shown in Figure 11.3 for Moore type and in Figure 11.4 for Mealy type. We needed to generate 17 722 535 initially connected Moore machine with 60 states to obtain 10 machines satisfying the conditions, i.e. only $5.6 \cdot 10^{-5}$ % out of generated machines meet the requirements. Rapidly decreasing rate of complaint machines to generated ones implies the following correlation. The more states the generated machine has, the lower probability of meeting the condition is. Therefore, we were not able to obtain complaint machines with more states in reasonable time.

The proportions in Figure 11.3 and Figure 11.4 show another interesting information besides how hard is to obtain a machine for testing the C- and M- method. Assume that our generator uniformly samples the space of initially connected machines with given numbers of states, inputs and outputs. It seems that unreduced machines take the same proportion of the space for all classes with different numbers of states. It is similar in case of not strongly connected machines. They take approximately equal proportion of the spaces. Comparison of FSMs' classes with properties, such as connectedness and minimality, is discussed in Section 3.4. The main observation is the enormous space of strongly connected, reduced machines that have got no adaptive

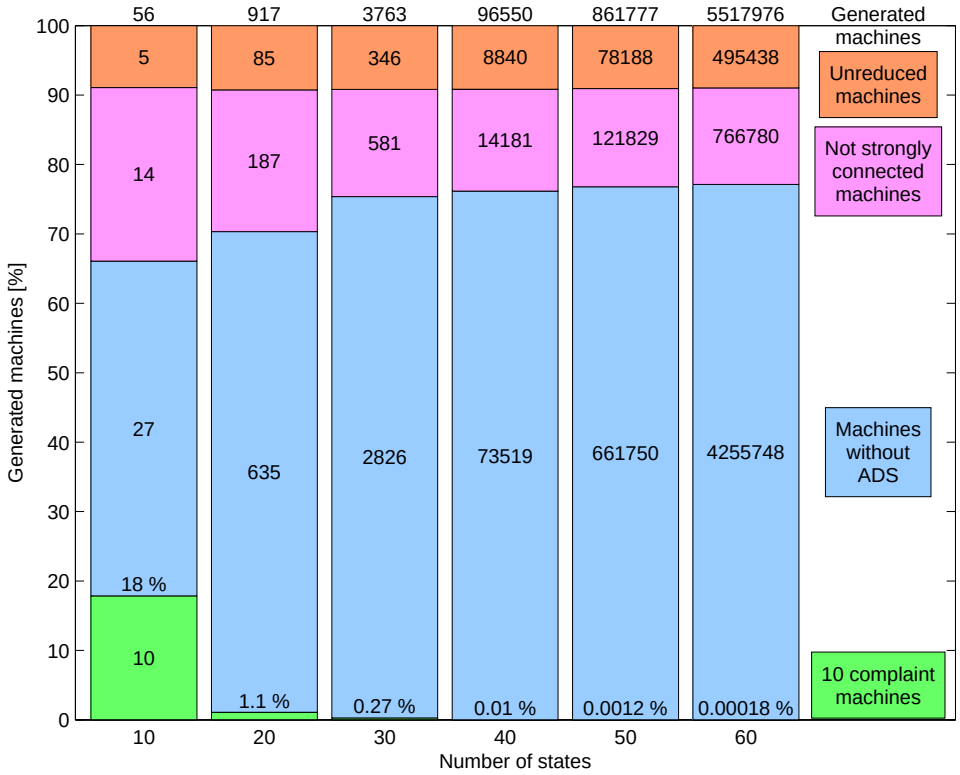


Figure 11.4. Numbers of Generated Mealy Machines for the Experiment

distinguishing sequence. This is a huge limitation of most testing methods creating a checking sequence. They cannot be used for great number of machines.

Mealy machines have got outputs by transition therefore they possess more distinguishing power than Moore ones having outputs by states. This is seen in comparison of the numbers of generated machines for Moore machines in Figure 11.3 and for Mealy machines in Figure 11.4. It is easier to find 10 complaint Mealy machines than 10 complaint Moore machines. Two Mealy states can be distinguished by each transition or by each next state, i.e. there are $2p$ possibilities for distinguishing ($p = |X|$). Two Moore states can be distinguished by their own output or by each next state, i.e. they have got only $p + 1$ distinguishing possibilities.

Lengths of checking sequences generated by the C-, M^a-, M^g- and M*- methods are captured using boxplot[1]) for each machine in the experiment in Figure 11.5 and in Figure 11.6. In our experiment, each box deals with 10 values, i.e. the lengths of checking sequences generated by a related method for a group of machines with the same number of states. Note that minimum, maximum and all three quartiles' values of the methods usually relate with the same machine. For example, outliers in group of Moore machines with 50 states (Figure 11.5) correspond to a machine that is more complex than the others in the group; this machine has longer adaptive distinguishing sequence so its checking sequences are longer as well.

Figure 11.5 and Figure 11.6 show that the M-method outperforms the C-method in most cases. The M^g-method seems to be a very good approximator of the M-method as

[1]) Boxplot is a statistical tool for visualization. Blue rectangle contain values withing the first and third quartiles. Red line is the second quartile (the median). Lines outgoing from blue box are so-called whiskers and capture other values. If a value is too far from the others, it is marked by a red cross as an outlier.

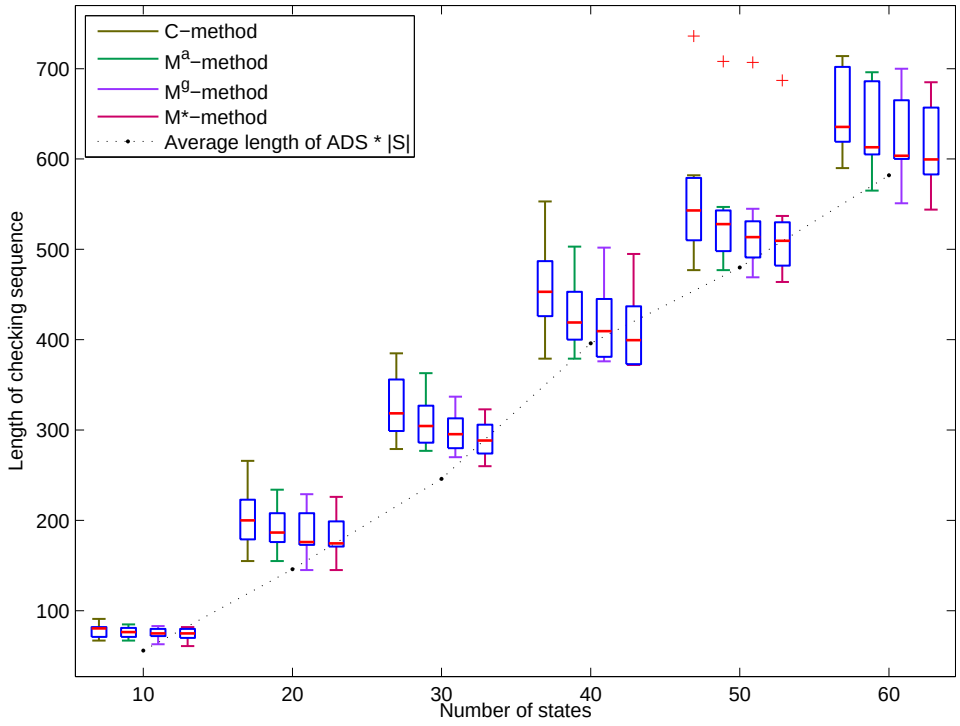


Figure 11.5. Comparison of Lengths of Checking Sequences of Moore Machines

it creates checking sequences of length close to the optimal values. The optimal value is shown by the M*-method. The M^a-method is a quite worse than the M^g-method but it is still usually better than the C-method. All created checking sequences were proved by the FCC to be n-complete. In addition, no testing node need to be estimated owing to providing reference nodes in advance as a hint, see Section 9.3.

There is another information captured in Figure 11.5 and Figure 11.6 besides boxplots. Each checking sequence contains adaptive distinguishing sequence appended in each state. Therefore, an interesting factor could be a comparison of the lengths of ADS and checking sequence. The length of ADS is the length of the longest distinguishing sequence d_i in ADS. We counted average a_n of the lengths of ADSs over every 10 samples with the same number of states n. Then, these averages are multiplied by the related number of states and shown for each group of machines in the figures. We discuss dependence of lengths of ADS and checking sequence after the second experiment's description.

The second experiment is introduced to demonstrate the need of suboptimal approaches. We generated machines with the number of states in multiples of hundreds randomly. Probability of sampling such a machine with small input and output alphabet, that holds strongly connectedness, is reduced and has got an ADS, is very low. Therefore, we decided to set sizes of input and output alphabets to be proportional to the number of states. The larger input alphabet is, the more connected the machine is. The size of output alphabet influences presence of a distinguishing sequence. We experimentally chose sizes of both alphabets to be one twentieth of the number of states. In such way, ten Mealy machines with 100-1000 states were created.

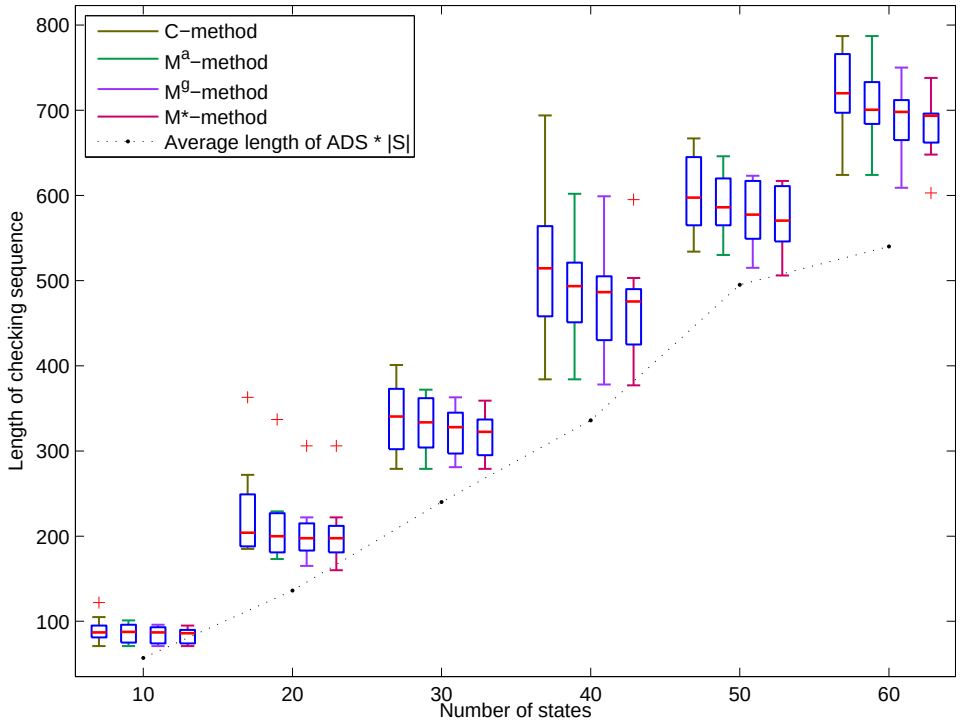


Figure 11.6. Comparison of Lengths of Checking Sequences of Mealy Machines

We use Gurobi solver to obtain an optimal solution, i.e. it is employed in the M*-method. Unfortunately, the solver was not able to find a solution for the machines of the second experiment due to computer memory constraints. Therefore, the M*-method is not included in the experiment. The M^g-method had similar problem. It is very memory consuming so results was not obtained for machines with more than 600 states. Table 11.3 contains the lengths of checking sequences generated by the C-method, the M^a-method and the M^g-method. Moreover, there are lengths of ADSs and percentage rates of improvement the C-method using the M^a-method or the M^g-method.

n	$\|X\| = \|Y\|$	|ADS|	C	M^a	M^g	C/M^a	C/M^g
100	5	6	2468	2448	2429	0.817 %	1.606 %
200	10	5	9010	8992	8851	0.200 %	1.796 %
300	15	4	19619	19600	19307	0.097 %	1.616 %
400	20	4	34451	34413	33990	0.110 %	1.356 %
500	25	4	52777	52745	51955	0.061 %	1.582 %
600	30	4	72721	72713	71649	0.011 %	1.496 %
700	35	4	99762	99737	*	0.025 %	*
800	40	4	127209	127160	*	0.039 %	*
900	45	4	159725	159725	*	0.000 %	*
1000	50	4	197936	197939	*	-0.002 %	*

Table 11.3. Length of Checking Sequence of the C-method and the M-method

The second experiment with results shown in Table 11.3 confirms observation from the previous experiment. However, it found out the limitation of the M-method. As the M-method is an NP-complete problem, it is infeasible to find an optimal solution for large machines. Besides the M*-method, we could not obtain a solution of the M^g-method for machines with more than 600 states and 30 inputs. The FCC was able to prove n-completeness of the checking sequences generated for machines with the number of states up to 400. Testing tree with nodes domain and different set did not fit in the computer memory for machines with more than 400 states and 20 inputs.

Notice that the C-method generates shorter checking sequence of the machine with 1000 (the last row of Table 11.3) than the M^a-method. This is due to its ability to shorten adaptive distinguishing sequence d_i using output-confirmed sequences.

The correspondence of the lengths of ADS and checking sequence is an interesting factor for a discussion. Each sequence of U needs to be included in created checking sequence. If one does not count with overlapping and connecting sequences, the length of checking sequence is bounded by $(np + 1)(|d| + 1)$, where $|d|$ is the length of ADS. The results of the first experiment show that the length of checking sequence is close to $n|d|$ due to overlapping and the fact that most distinguishing sequences d_i in ADS are shorter than d, i.e. $|d_i| \leq |d|$. Nevertheless, Table 11.3 indicates a little different correspondence. The length of checking sequence roughly matches to $np|d|$. Notice that the machines in the first experiment have got binary input and output alphabets so that $p = 2$. However, the bound $np|d|$ is over the lengths of checking sequences in the first experiment. More accurate bound may be $n(p - 1)|d|$ that compensates overlapping, connecting sequences and different lengths of distinguishing sequences in ADS. It is a topic for a future study.

11.3 Resettable machines

The last proposed experiment compares all implemented methods. Descriptions of particular method implementations are in Section 10.4. We consider the PDS-, ADS-, SVS-, W-, Wp-, HSI-, H-, SPY-, C-, M- and Mr- methods. Some methods require a specific property of the specification, i.e. the machine which a created test suite is based on. We pick the first machine of each group of the first experiment of Section 11.2. However, Moore machines with 10 and 30 states have no preset distinguishing sequence so other 10-state and 30-state machines having PDS were chosen from the related groups. The comparison of all testing methods were performed on 6 Moore and 6 Mealy machines with binary input and output alphabets that differ in the number of states.

Figure 11.7 shows lengths of test suites created by the methods on Moore machines and the same is captured for Mealy machines in Figure 11.8. Comparison on 6 Moore machines and on 6 Mealy machines is to have an informative character only. Two machines with the same numbers of states, inputs and outputs can possess completely different characteristics so a comparison of testing methods could provide different results on such machines. In case of checking sequence testing methods, we provided a characteristic of a group of machines using boxplot in Figure 11.5 and Figure 11.6. However, we found out that a comparison on a single machine has got similar informative value. Therefore, the second experiment of checking sequence design methods and this experiment of all testing methods are performed on one machine per group of machine with the same number of states. In addition, it would be harder to compare more values when we have got 11 testing methods to compare. Another possible approach is use of average value of group of machines but it is not so accurate due to the mentioned different characteristics of machines.

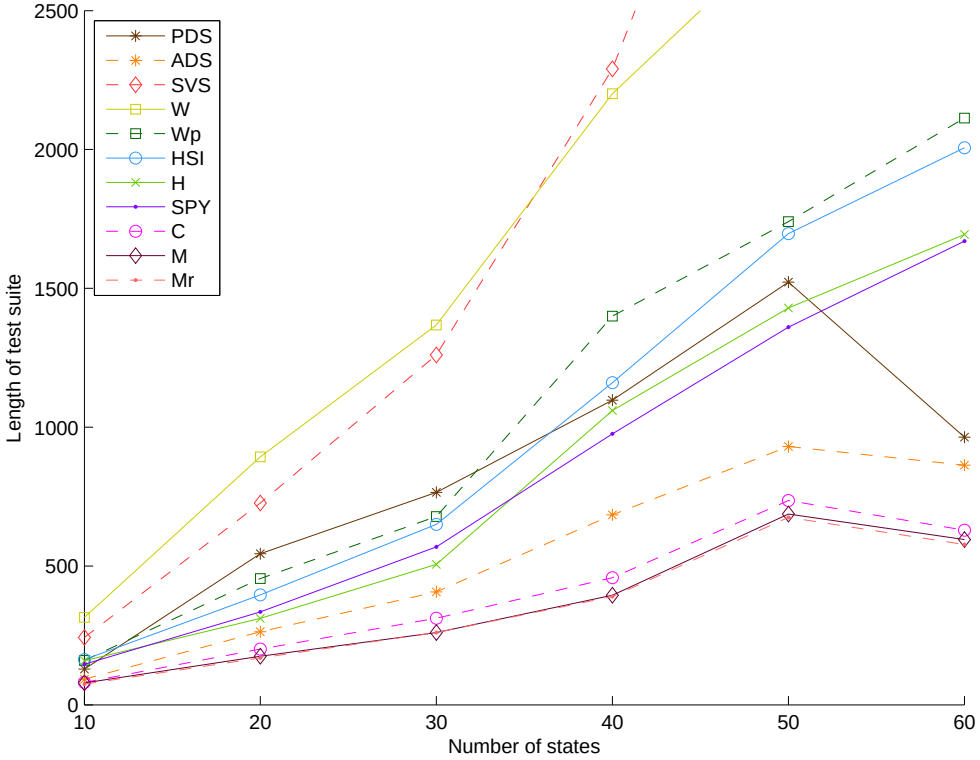


Figure 11.7. Comparison of Lengths of Test Suites of Moore Machines

We constrained the maximum length of test suite for visualization to 2500 input symbols because the SVS- and W- methods produce significantly longer test suites than the other methods so the results would not be distinguishable if the y axis scales according to the longest test suites. Table 11.4 contains lengths of test suites created for the Moore and Mealy machines with 60 states. As a reminder, the length of a test suite T is sum of lengths of all tests plus size of T that is the number of uses of reset. Formally, $\text{len}(T) = \sum_{t \in T} |t| + |T| = \sum_{t \in T}(|t| + 1)$. Values for the M-method and the Mr-method are obtained using the M*-method, i.e. using Gurobi solver.

Method	Moore	Mealy
SVS	4953	7657
W	2572	4190
Wp	2113	2382
HSI	2006	2298
H	1694	1915
SPY	1670	1732
PDS	964	1013
ADS	863	956
C	629	712
M	595	648
Mr	577	636

Table 11.4. Lengths of Test Suites of two 60-state Machines

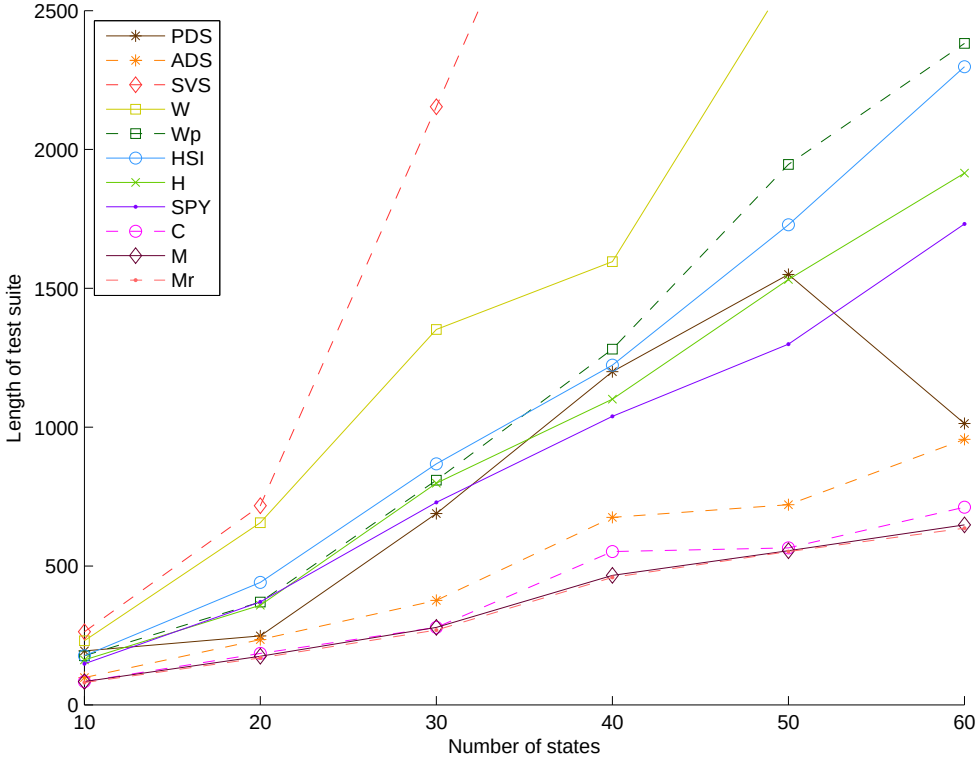


Figure 11.8. Comparison of Lengths of Test Suites of Mealy Machines

Results on both types of FSM capture known properties of testing methods. The W-method always produces a longer test suite than the Wp-method. The HSI-method is comparable to the Wp-method. In addition, the H-method is generally better than the HSI-method and is comparable to the SPY-method (or a little worse - it depends on given machine). The PDS-method always creates a longer test suite than the ADS-method due to the fact that ADS is shorter than PDS. If one replaces some tests of the ADS-method with reasonable extension of other tests as the Mr-method describes, the test suite is even shorter. The Mr-method always produces shorter or equal test suite than the M-method and on given 12 machines the C-method did not create shorter checking sequence than the M-method. As for the previous experiment, the FCC proved n-completeness of all created test suites.

It is worth mentioning that the experiment was performed on reduced, strongly connected machines that posses PDS. Therefore, we were able to obtain a test suite by each testing method. Nevertheless, we shown how small portion of machines meet such conditions in the experiment of checking sequence design methods, see Section 11.2. A general conclusion can be made. If a given machine is reduced, strongly connected and has got ADS, then we use the M-method, or the Mr-method when reset is available. Otherwise, i.e. we deal with a reduced, initially connected machine, we employ the SPY-method (or the H-method) which requires reliable reset. What method should be used when given machine has no ADS and no reset? This problem needs further research. The DW-methods (Section 6.2.4) can be used if the machine is strongly connected but it produces exponentially long checking sequence.

Chapter 12
Conclusion

The field of testing finite-state machines is very extensive. Therefore, we have got a lot of things to study. We researched classes of FSMs and their relationships in Chapter 3. Then we stated test suite's properties (Chapter 4), we described known testing methods (Chapter 6) and their interference in terms of design and time of proposal (Chapter 7).

Consequently, we were able to propose a new testing method, the M-method (Chapter 8). The M-method was experimentally proven to create an n-complete test suite. A new proposed method for checking fault coverage of given test suite was used for verification of the M-method. We call the checking method the Fault Coverage Checker and it was described in Chapter 9. In experiments in Chapter 11 there was shown that the M-method produces the shortest test suite from all 10 tested testing methods. The M-method has got two drawbacks. The former one is that the method is applicable only on reduced strongly connected machines having an adaptive distinguishing sequence. The latter one is that finding an optimal solution of the M-method is an NP-complete problem. Therefore, we propose two suboptimal approaches that approximates the optimal value quite well and they are polynomial in the number of states.

In Chapter 10 we described our implementations of testing methods. Therefore, the results of experiments are uniquely determined owing to the proposed implementations. Testing methods for resettable machines usually need a characterizing set, or a set of separating sequences, but they do not specify how it could be obtained. Thus, we proposed algorithms for creating separating sequences and characterizing sets in Chapter 10. New parallel approaches for design shortest separating sequences and new method for reduction of characterizing sets are discussed as well.

We studied a lot of scientific papers as Chapter 5 shows so we had to become familiar with many different notations. Hence, we propose an unification of denotations in Section 3.7. It sets reasonable rules of use of symbols for notations.

There are a plenty of topics that we only touched in this thesis and they need a further research. The essential one is a formal proof of the M-method that it creates an n-complete checking sequence. Then test segments can be optimized, using output-confirmed sequences, for example. Another problem to deal with is finding optimal harmonized state identifiers or characterizing sets. A research of structure of finite-state machines and relations to particular machine's properties, e.g. having adaptive distinguishing sequence, is an example of more theoretical task that can be found in this thesis. Nevertheless, we became familiar with the field of testing finite-state machines and understanding of this field now help us to continue with our research in automata active learning.

References

[Ah91] Alfred V Aho, Anton T Dahbura, David Lee, and M Umit Uyar. An optimization technique for protocol conformance test generation based on UIO sequences and rural Chinese postman tours. *Communications, IEEE Transactions on*. 1991, 39 (11), 1604–1615.

[An87] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*. 1987, 75 (2), 87–106.

[Bo74] RT Boute. Distinguishing sets for optimal state identification in checking experiments. *Computers, IEEE Transactions on*. 1974, 100 (8), 874–877.

[Ch89] Wendy YL Chan, CT Vuong, and MR Otp. An improved protocol test generation procedure based on UIOs. *ACM SIGCOMM Computer Communication Review*. 1989, 19 (4), 283–294.

[Ch05] Jessica Chen, Robert M Hierons, Hasan Ural, and Husnu Yenigun. *Eliminating redundant tests in a checking sequence*. 2005.

[Ch06] Kai Chen, Fan Jiang, and Chuan-dong Huang. *A new method of generating synchronizable test sequences that detect output-shifting faults based on multiple uio sequences*. In: *Proceedings of the 2006 ACM symposium on Applied computing*. 2006. 1791–1797.

[Ch03] Wen-Huei Chen, and others. An optimization technique for protocol conformance testing based on the wp method. *International Journal of Applied Science and Engineering 1 (1)*. 2003, 45–54.

[Ch95i] Wen-Huei Chen, Chuan Yi Tang, and Son T Vuong. Improving the UIOv-method for protocol conformance testing. *Computer Communications*. 1995, 18 (9), 609–619.

[Ch95s] Wen-Huei Chen, and Hasan Ural. Synchronizable test sequences based on multiple UIO sequences. *IEEE/ACM Transactions on Networking (TON)*. 1995, 3 (2), 152–157.

[Ch78] Tsun S. Chow. Testing software design modeled by finite-state machines. *Software Engineering, IEEE Transactions on*. 1978, (3), 178–187.

[De94] RG Deshmukh, and GN Hawat. *An algorithm to determine shortest length distinguishing, homing, and synchronizing sequences for sequential machines*. In: *Southcon/94. Conference Record*. 1994. 496–501.

[Do05e] Rita Dorofeeva, Khaled El-Fakih, Stephane Maag, Ana R Cavalli, and Nina Yevtushenko. *Experimental evaluation of FSM-based testing methods*. In: *Software Engineering and Formal Methods, 2005. SEFM 2005. Third IEEE International Conference on*. 2005. 23–32.

[Do05i] Rita Dorofeeva, Khaled El-Fakih, and Nina Yevtushenko. *An improved conformance testing method*. 2005.

[Du07] Lihua Duan, and Jessica Chen. *Reducing test sequence length using invertible sequences*. 2007.

[Du09] Lihua Duan, and Jessica Chen. Exploring alternatives for transition verification. *Journal of Systems and Software*. 2009, 82 (9), 1388–1402.

[En13] Andre Takeshi Endo, and Adenilso Simao. Evaluating test suite characteristics, cost, and effectiveness of FSM-based testing methods. *Information and Software Technology*. 2013, 55 (6), 1045–1062.

[Fl62] Robert W Floyd. Algorithm 97: shortest path. *Communications of the ACM*. 1962, 5 (6), 345.

[Fu91] Susumu Fujiwara, F Khendek, M Amalou, A Ghedamsi, and others. Test selection based on finite state models. *Software Engineering, IEEE Transactions on*. 1991, 17 (6), 591–603.

[Gh92] Abderrazak Ghedamsi, and Gv Bochmann. *Test result analysis and diagnostics for finite state machines.* In: *Distributed Computing Systems, 1992., Proceedings of the 12th International Conference on*. 1992. 244–251.

[Gh93] Abderrazak Ghedamsi, GV Bochmann, and Rachida Dssouli. *Multiple fault diagnosis for finite state machines.* In: *INFOCOM'93. Proceedings. Twelfth Annual Joint Conference of the IEEE Computer and Communications Societies. Networking: Foundation for the Future, IEEE*. 1993. 782–791.

[Gi62] Arthur Gill. *Introduction to the Theory of Finite-state Machines*. McGraw-Hill Book Company, 1962.

[Go70] Guney Gonenc. A method for the design of fault detection experiments. *Computers, IEEE Transactions on*. 1970, 100 (6), 551–558.

[Go07] Sezer Gören, and F Joel Ferguson. On state reduction of incompletely specified finite state machines. *Computers & Electrical Engineering*. 2007, 33 (1), 58–69.

[He64] FC Hennie. *Fault detecting experiments for sequential circuits.* In: *Switching Circuit Theory and Logical Design, 1964 Proceedings of the Fifth Annual Symposium on*. 1964. 95–110.

[Hi09] Robert M Hierons, G-V Jourdan, Hasan Ural, and Husnu Yenigun. *Checking sequence construction using adaptive and preset distinguishing sequences.* In: *Software Engineering and Formal Methods, 2009 Seventh IEEE International Conference on*. 2009. 157–166.

[Hi02] Robert M Hierons, and Hasan Ural. Reduced length checking sequences. 2002,

[Hi06] Robert M Hierons, and Hasan Ural. Optimizing the length of checking sequences. *Computers, IEEE Transactions on*. 2006, 55 (5), 618–629.

[Hi96] Robert M. Hierons. Extending test sequence overlap by invertibility. *The Computer Journal*. 1996, 39 (4), 325–330.

[Hi97] Robert M. Hierons. Testing from a finite-state machine: extending invertibility to sequences. *The Computer Journal*. 1997, 40 (4), 220–230.

[Ho71] John Hopcroft. An n log n algorithm for minimizing states in a finite automaton. 1971,

[Ho06] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. *Introduction to Automata Theory, Languages, and Computation*. Prentice Hall, 2006.

[Hs71] EP Hsieh. Checking Experiments for Sequential Machines. *Computers, IEEE Transactions on*. 1971, 100 (10), 1152–1166.

[Jo10] Guy-Vincent Jourdan, Hasan Ural, Hüsnü Yenigün, and Ji Chao Zhang. Lower bounds on lengths of checking sequences. *Formal aspects of computing*. 2010, 22 (6), 667–679.

[Ka10] M Kapus-Kolar. A better procedure and a stronger state-recognition pattern for checking sequence construction. *Jožef Stefan Institute Technical Report*. 2010, 10574

[Ka12n] Monika Kapus-Kolar. New state-recognition patterns for conformance testing of finite state machine implementations. *Computer Standards & Interfaces*. 2012, 34 (4), 390–395.

[Ka12e] Monika Kapus-Kolar. On "Exploring alternatives for transition verification". *Journal of Systems and Software*. 2012, 85 (8), 1744–1748.

[Ka14] Monika Kapus-Kolar. On the global optimization of checking sequences for finite state machine implementations. *Microprocessors and Microsystems*. 2014, 38 (3), 208–215.

[Ke94] Michael Kearns, and Umesh Virkumar Vazirani. *An introduction to computational learning theory*. The MIT Press, 1994.

[Ki12] David B Kirk, and W Hwu Wen-mei. *Programming massively parallel processors: a hands-on approach*. Newnes, 2012.

[Ko10] Zvi Kohavi, and Niraj K Jha. *Switching and finite automata theory*. Cambridge University Press, 2010.

[Ku94] BP Vijay Kumar, and Pallapa Venkataram. An optimization technique for Protocol conformance test sequence generation based on MUIOS using Hopfield Neural Networks. 1994,

[La80] Richard E Ladner, and Michael J Fischer. Parallel prefix computation. *Journal of the ACM (JACM)*. 1980, 27 (4), 831–838.

[La76] Eugene L Lawler. *Combinatorial optimization: networks and matroids*. Courier Corporation, 1976.

[Le96c] David Lee, Krishan K Sabnani, David M Kristol, and Sanjoy Paul. Conformance testing of protocols specified as communicating finite state machines-a guided random walk based approach. *Communications, IEEE Transactions on*. 1996, 44 (5), 631–640.

[Le94] David Lee, and Mihalis Yannakakis. Testing finite-state machines: State identification and verification. *Computers, IEEE Transactions on*. 1994, 43 (3), 306–320.

[Le96p] David Lee, and Mihalis Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*. 1996, 84 (8), 1090–1123.

[Lu94] Gang Luo, Gregor v Bochmann, and Alexandre Petrenko. Test selection based on communicating nondeterministic finite-state machines using a generalized Wp-method. *Software Engineering, IEEE Transactions on*. 1994, 20 (2), 149–162.

[Lu95] Gang Luo, Alexandre Petrenko, and Gregor v Bochmann. *Selecting test sequences for partially-specified nondeterministic finite state machines*. 1995.

[Mi10] Huaikou Miao, Pan Liu, and Jia Mei. *An improved algorithm for building the characterizing set*. In: *Theoretical Aspects of Software Engineering (TASE), 2010 4th IEEE International Symposium on*. 2010. 67–74.

[Mo56] Edward F Moore. Gedanken-experiments on sequential machines. *Automata studies*. 1956, 34 129–153.

[Pe91] Alexandre Petrenko. *Checking experiments with protocol machines*. In: *Proceedings of the IFIP TC6/WG6. 1 Fourth International Workshop on Protocol Test Systems IV*. 1991. 83–94.

[Pe96] Alexandre Petrenko, Gv Bochmann, and M Yao. On fault coverage of tests for finite state specifications. *Computer Networks and ISDN Systems*. 1996, 29 (1), 81–106.

[Pe92] Alexandre Petrenko, and Nina Yevtushenko. *Test suite generation from a fsm with a given type of implementation errors*. In: *Proceedings of the IFIP TC6/WG6. 1 Twelth International Symposium on Protocol Specification, Testing and Verification XII*. 1992. 229–243.

[Pe05] Alexandre Petrenko, and Nina Yevtushenko. Testing from partial deterministic FSM specifications. *Computers, IEEE Transactions on*. 2005, 54 (9), 1154–1165.

[Po13] Faimison Rodrigues Porto, Andre Takeshi Endo, and Adenilso Simao. *Generation of Checking Sequences Using Identification Sets*. 2013.

[Re95] Ali Rezaki, and Hasan Ural. Construction of checking sequences based on characterization sets. *Computer Communications*. 1995, 18 (12), 911–920.

[Sa85] Krishan Sabnani, and Anton Dahbura. A new technique for generating protocol test. *ACM SIGCOMM Computer Communication Review*. 1985, 15 (4), 36–43.

[Sa88] Krishan Sabnani, and Anton Dahbura. A protocol test generation procedure. *Computer Networks and ISDN systems*. 1988, 15 (4), 285–297.

[Sh92] Y-N Shen, Fabrizio Lombardi, and Anton T Dahbura. Protocol conformance testing using multiple UIO sequences. *Communications, IEEE Transactions on*. 1992, 40 (8), 1282–1287.

[Si88] Deepinder Sidhu, and T-k Leung. *Fault coverage of protocol test methods*. In: *INFOCOM'88. Networks: Evolution or Revolution, Proceedings. Seventh Annual Joint Conference of the IEEE Computer and Communcations Societies, IEEE*. 1988. 80–85.

[Si89] Deepinder Sidhu, and T-K Leung. Formal methods for protocol testing: A detailed study. *Software Engineering, IEEE Transactions on*. 1989, 15 (4), 413–426.

[Si08] Adenilso Simão, and Alexandre Petrenko. *Generating checking sequences for partial reduced finite state machines*. 2008.

[Si09c] Adenilso Simão, and Alexandre Petrenko. *Checking sequence generation using state distinguishing subsequences*. In: *Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on*. 2009. 48–56.

[Si09f] Adenilso Simão, and Alexandre Petrenko. Fault coverage-driven incremental test generation. *The Computer Journal*. 2009, bxp073.

[Si10] Adenilso Simão, and Alexandre Petrenko. Checking completeness of tests for finite state machines. *Computers, IEEE Transactions on*. 2010, 59 (8), 1023–1032.

[Si12] Adenilso Simão, Alexandre Petrenko, and Nina Yevtushenko. On reducing test length for fsms with extra states. *Software Testing, Verification and Reliability*. 2012, 22 (6), 435–454.

[So14] Michal Soucha. Finite-State Machine State Identification Sequences. 2014,

[Ur92] Hasan Ural. Formal methods for test sequence generation. *Computer communications*. 1992, 15 (5), 311–325.

[Ur97] Hasan Ural, Xiaolin Wu, and Fan Zhang. On minimizing the lengths of checking sequences. *Computers, IEEE Transactions on*. 1997, 46 (1), 93–99.

[Ur06] Hasan Ural, and Fan Zhang. *Reducing the lengths of checking sequences by overlapping*. 2006.

[Va73] MP Vasilevskii. Failure diagnosis of automata. *Cybernetics and Systems Analysis*. 1973, 9 (4), 653–665.

[Vu90] Son T Vuong, and Kai C Ko. *A novel approach to protocol test sequence generation.* In: *Global Telecommunications Conference, 1990, and Exhibition. 'Communications: Connecting the Future', GLOBECOM'90., IEEE*. 1990. 1880–1884.

[Wa10] Qiang Wang, Shuai Wang, and Yindong Ji. *A test sequences optimization method for improving fault coverage.* In: *Information Management and Engineering (ICIME), 2010 The 2nd IEEE International Conference on*. 2010. 80–84.

[Xi06] Jitian Xiao, Chiou Peng Lam, Huaizhong Li, and Jun Wang. *Reformulation of the generation of conformance testing sequences to the asymmetric travelling salesman problem.* In: *Proceedings of the 8th annual conference on Genetic and evolutionary computation*. 2006. 1933–1940.

[Ya95] Mihalis Yannakakis, and David Lee. Testing finite state machines: fault detection. *Journal of Computer and System Sciences*. 1995, 50 (2), 209–227.

[Ya94] Mingyu Yao, Alexandre Petrenko, and Gv Bochmann. *Fault coverage analysis in respect to an FSM specification.* In: *INFOCOM'94. Networking for Global Communications., 13th Proceedings IEEE*. 1994. 768–775.

[Zh93] Jinsong Zhu, and Samuel T Chanson. *Fault coverage evaluation of protocol test sequences*. University of British Columbia, Department of Computer Science, 1993.

Appendix A
Abbreviations and Symbols

A.1 Abbreviations

FSM	Finite-state machine
DS	Distinguishing sequence
ADS	Adaptive DS d_i
PDS	Preset DS d
SVS	State verifying sequence
SVSet	States verifying set of sequences
SCSet	State characterizing set of sequences
CSet	Characterizing set W of sequences
HSI	Harmonized state identifier H_i
HS	Homing sequence
SS	Synchronizing sequence
CS	Checking sequence
SepSeq	Separating sequence

A.2 Symbols

S	a set of states; $n = \|S\|$, $S = \{s_0, \ldots, s_{n-1}\}$
ε	the empty symbol with zero length
X	an input alphabet; $p = \|X\|$, $X = \{x_1, \ldots, x_p\}$
X_ε	an input alphabet extended with ε
X^*	a set of all strings over the input alphabet; $t, u, v, w \in X^*$
Y	an output alphabet; $q = \|Y\|$, $Y = \{y_1, \ldots, y_q\}$
Y_ε	an output alphabet extended with ε
Y^*	a set of all strings over the output alphabet; $z \in Y^*$
δ	the state-transition function
λ	the output function
D	a domain of defined transition; $D \subseteq S \times X$
$\Omega(s)$	a set of all defined input sequences for state s
r	reliable reset
M	a finite-state machine $(S, X, Y, \delta, \lambda, s_0)$
N	a finite-state machine $(Q, X', Y', \Delta, \Lambda, q_0)$; $m = \|Q\|$

Appendix B
Checking Sequence Example

We found a machine that caused troubles for the Fault Coverage Checker when the C-method produced a checking sequence on it. The machine has got 40 states, binary input and output alphabets and it is type of Mealy. Its state diagram is shown on the next page. Verification of state 36 is the problem part. There is only one transition to this state, $(26, 0)$ in particular. This transition is a prefix of the distinguishing sequence d_{26} of state 26 so the C-method shortens distinguishing sequence d_{36} when it verifies the transition $(26, 0)$. The produced checking sequence with 491 input symbols follows:

$$\varepsilon\,0\,0_{21}\,0_{15}\,0_{14}\,1_{37}\,0_{7}\,0_{2}\,0_{9}\,0_{16}\,1_{31}\,0_{32}\,0_{37}\,0_{7}\,0_{2}\,1_{34}\,0_{38}\,0_{28}\,0_{4}\,0_{25}\,0_{19}$$

$$0_{17}\,1_{14}\,0_{24}\,0_{10}\,0_{31}\,0_{32}\,0_{37}\,1_{3}\,0_{29}\,0_{18}\,0_{2}\,0_{9}\,0_{16}\,0_{39}\,0_{8}\,1_{19}\,0_{17}\,1_{14}\,0_{24}\,0_{10}$$

$$0_{31}\,0_{32}\,0_{37}\,0_{7}\,0_{2}\,1_{34}\,1_{31}\,0_{32}\,0_{37}\,0_{7}\,0_{2}\,1_{34}\,0_{38}\,0_{28}\,0_{4}\,0_{25}\,0_{19}\,0_{17}\,0_{23}\,0_{9}$$

$$1_{28}\,0_{4}\,0_{25}\,0_{19}\,0_{17}\,0_{23}\,0_{9}\,1_{28}\,1_{37}\,0_{7}\,0_{2}\,0_{9}\,0_{16}\,1_{31}\,0_{32}\,0_{37}\,0_{7}\,0_{2}\,0_{9}\,0_{16}$$

$$1_{31}\,0_{32}\,1_{38}\,0_{28}\,0_{4}\,0_{25}\,0_{19}\,0_{17}\,1_{14}\,0_{24}\,1_{13}\,0_{35}\,0_{34}\,0_{38}\,0_{28}\,1_{37}\,0_{7}\,0_{2}\,0_{9}\,0_{16}$$

$$1_{31}\,0_{32}\,0_{37}\,0_{7}\,0_{2}\,0_{9}\,0_{16}\,0_{39}\,0_{8}\,1_{19}\,0_{17}\,1_{14}\,1_{37}\,0_{7}\,0_{2}\,0_{9}\,0_{16}\,1_{31}\,0_{32}\,1_{38}$$

$$1_{11}\,0_{1}\,0_{24}\,0_{10}\,0_{31}\,1_{12}\,0_{20}\,1_{25}\,0_{19}\,0_{17}\,0_{23}\,0_{9}\,1_{28}\,0_{4}\,1_{11}\,0_{1}\,0_{24}\,0_{10}\,0_{31}\,1_{12}$$

$$0_{20}\,1_{25}\,0_{19}\,0_{17}\,0_{23}\,0_{9}\,0_{16}\,0_{39}\,1_{29}\,0_{18}\,0_{2}\,0_{9}\,0_{16}\,0_{39}\,0_{8}\,0_{12}\,0_{20}\,0_{26}\,1_{30}\,0_{22}$$

$$0_{26}\,0_{36}\,0_{13}\,0_{35}\,0_{34}\,1_{31}\,0_{32}\,0_{37}\,0_{7}\,0_{2}\,1_{34}\,1_{31}\,1_{12}\,0_{20}\,0_{26}\,0_{36}\,0_{13}\,0_{35}\,0_{34}\,1_{31}$$

$$1_{12}\,1_{5}\,0_{18}\,0_{2}\,0_{9}\,0_{16}\,0_{39}\,0_{8}\,0_{12}\,0_{20}\,0_{26}\,0_{36}\,0_{13}\,0_{35}\,0_{34}\,1_{31}\,1_{12}\,0_{20}\,1_{25}\,0_{19}$$

$$0_{17}\,0_{23}\,0_{9}\,1_{28}\,0_{4}\,0_{25}\,1_{26}\,0_{36}\,0_{13}\,0_{35}\,0_{34}\,1_{31}\,1_{12}\,1_{5}\,1_{7}\,0_{2}\,0_{9}\,0_{16}\,0_{39}\,0_{8}$$

$$1_{19}\,0_{17}\,0_{23}\,0_{9}\,0_{16}\,0_{39}\,1_{29}\,0_{18}\,1_{0}\,0_{21}\,0_{15}\,0_{14}\,1_{37}\,0_{7}\,0_{2}\,0_{9}\,1_{28}\,0_{4}\,0_{25}\,0_{19}$$

$$0_{17}\,0_{23}\,0_{9}\,1_{28}\,0_{4}\,1_{11}\,0_{1}\,0_{24}\,0_{10}\,0_{31}\,0_{32}\,0_{37}\,1_{3}\,0_{29}\,1_{21}\,0_{15}\,0_{14}\,0_{24}\,0_{10}\,0_{31}$$

$$0_{32}\,0_{37}\,0_{7}\,1_{6}\,0_{1}\,0_{24}\,0_{10}\,1_{27}\,0_{33}\,0_{12}\,0_{20}\,0_{26}\,0_{36}\,0_{13}\,1_{13}\,0_{35}\,0_{34}\,0_{38}\,0_{28}\,1_{37}$$

$$0_{7}\,1_{6}\,0_{1}\,0_{24}\,0_{10}\,0_{31}\,0_{32}\,0_{37}\,1_{3}\,1_{16}\,0_{39}\,0_{8}\,0_{12}\,0_{20}\,0_{26}\,1_{30}\,0_{22}\,1_{17}\,0_{23}\,0_{9}$$

$$0_{16}\,0_{39}\,1_{29}\,0_{18}\,1_{0}\,0_{21}\,0_{15}\,0_{14}\,0_{24}\,0_{10}\,0_{31}\,0_{32}\,1_{38}\,1_{11}\,1_{26}\,0_{36}\,0_{13}\,0_{35}\,0_{34}\,1_{31}$$

$$1_{12}\,0_{20}\,0_{26}\,0_{36}\,0_{13}\,1_{13}\,0_{35}\,0_{34}\,0_{38}\,0_{28}\,1_{37}\,0_{7}\,1_{6}\,1_{20}\,0_{26}\,0_{36}\,0_{13}\,0_{35}\,0_{34}\,1_{31}$$

$$0_{32}\,1_{38}\,1_{11}\,0_{1}\,1_{12}\,0_{20}\,0_{26}\,0_{36}\,0_{13}\,0_{35}\,0_{34}\,1_{31}\,1_{12}\,0_{20}\,0_{26}\,0_{36}\,1_{8}\,0_{12}\,0_{20}\,0_{26}$$

$$0_{36}\,0_{13}\,0_{35}\,0_{34}\,0_{38}\,0_{28}\,0_{4}\,0_{25}\,0_{19}\,0_{17}\,1_{14}\,0_{24}\,0_{10}\,1_{27}\,1_{0}\,0_{21}\,0_{15}\,0_{14}\,1_{37}\,0_{7}$$

$$0_{2}\,0_{9}\,0_{16}\,0_{39}\,1_{29}\,0_{18}\,0_{2}\,0_{9}\,0_{16}\,0_{39}\,0_{8}\,1_{19}\,1_{33}\,0_{12}\,0_{20}\,0_{26}\,0_{36}\,0_{13}\,1_{13}\,0_{35}$$

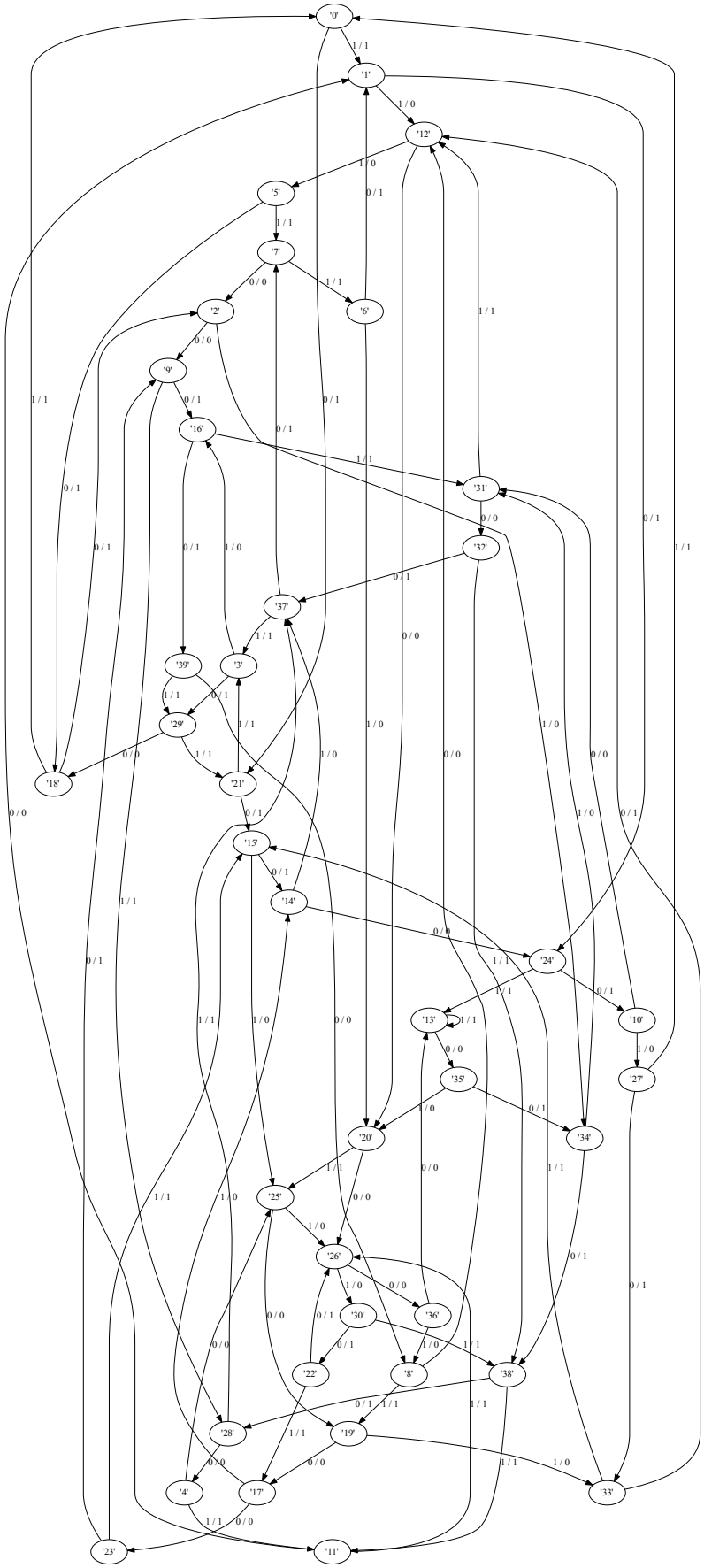
$$1_{20}\,0_{26}\,0_{36}\,0_{13}\,0_{35}\,0_{34}\,1_{31}\,1_{12}\,0_{20}\,0_{26}\,1_{30}\,1_{38}\,0_{28}\,0_{4}\,0_{25}\,0_{19}\,0_{17}\,1_{14}\,0_{24}\,0_{10}$$

$$1_{27}\,0_{33}\,0_{12}\,0_{20}\,0_{26}\,0_{36}\,0_{13}\,0_{35}\,0_{34}\,1_{31}\,1_{12}\,1_{5}\,0_{18}\,1_{0}\,1_{1}\,0_{24}\,0_{10}\,0_{31}\,0_{32}\,0_{37}$$

$$1_{3}\,0_{29}\,1_{21}\,1_{3}\,0_{29}\,0_{18}\,0_{2}\,0_{9}\,0_{16}\,0_{39}\,0_{8}\,1_{19}\,1_{33}\,1_{15}\,0_{14}\,0_{24}\,0_{10}\,0_{31}\,0_{32}\,0_{37}$$

$$1_{3}\,0_{29}\,1_{21}\,0_{15}\,1_{25}\,0_{19}\,0_{17}\,0_{23}\,0_{9}\,1_{28}\,0_{4}\,0_{25}\,0_{19}\,0_{17}\,0_{23}\,0_{9}\,0_{16}\,0_{39}\,0_{8}\,0_{12}$$

$$0_{20}\,0_{26}\,1_{30}\,0_{22}\,1_{17}\,0_{23}\,1_{15}\,0_{14}\,0_{24}\,0_{10}\,0_{31}\,0_{32}$$


'0'
'1'
'12'
'5'
'7'
'2'
'6'
'9'
'16'
'31'
'32'
'37'
'39'
'3'
'29'
'18'
'21'
'15'
'14'
'24'
'13'
'10'
'27'
'35'
'34'
'20'
'25'
'26'
'30'
'36'
'22'
'8'
'38'
'28'
'19'
'4'
'17'
'33'
'23'
'11'