BACHELOR THESIS
COMPUTER SCIENCE

RADBOUD UNIVERSITY

A Complete Version of the ADS Method for Conformance Testing

Author:
G.J. van Cuyck
S4750411

First supervisor / assessor:
Prof. F.W. Vaandrager
F.Vaandrager@cs.ru.nl

Second supervisor / assessor:
J. Moerman
Joshua.moerman@cs.ru.nl

January 14, 2019

Table of Contents

Abstract

Conformance testing has many uses, for instance to check the correctness of protocol implementations [2] or to learn models from black box implementations [3]. The ADS method [1] can be used for this purpose to good effect, but there are many cases where it is not applicable. In this thesis we propose a new method based on the ADS method that can be applied to any reduced, deterministic, fully defined finite state machine. We implement this new complete ADS method and compare its results with that of a different ADS expansion, the hybrid ADS method [3]. The results of this comparison show a significant improvement in test suite length in the cases where the regular ADS method is not applicable.

Introduction

Conformance testing can be used to check if the behaviour of a finite state machine (FSM) is same as that of a black box implementation[4]. There exist several different conformance testing algorithms and many of these, such as the W [5, 6], Wp [7] and HIS [8] methods, need a set of state identifiers to work. The performance of these algorithms depends largely on the length and number of state identifiers that are used. The ADS method [1] uses adaptive distinguishing sequences to generate a small set of state identifiers. It works very well, but there are many FSM's to which it can't be applied. The hybrid ADS method solves this problem, but its results are much longer in the cases where the regular ADS method is not applicable. In this thesis we describe a new way of generating a set of state identifiers based on the ADS method, called the complete ADS method. It is applicable to all deterministic, reduced, fully defined finite state machines.

In order to test the effectiveness of the new method, we implemented it and tested it on the collection of benchmarks gathered by the Radboud university [9]. This collection contains around 335 FSM's at the time of writing, 300 of which are suitable for use with the algorithm. The ones that are not are either too massive in size, formatted improperly or only partially defined. The implementation of our algorithm and some example benchmarks can be found online [10]. We also ran the pre-existing implementation of the hybrid ADS method [11] on all of the valid benchmarks in order to compare the effectiveness of our method to an existing solution. The results of this comparison are mixed in the general case, but if we focus on just the benchmarks that are not compatible with the ADS method, then we see a significant improvement. In addition to these tests, we also give a proof that our algorithm always terminates, and that it gives correct output upon termination.

Preliminaries

In this section, we present an overview of several concepts that will be used in this thesis. These concepts are not new and further information can be found in many papers, for example Lee and Yannakakis (1996) [4]. More specifically, we adopted several definitions from Moerman (2019) [12]. A list of the type of all the variables used in the thesis is included in Appendix A for clarity.

Finite state machines

The formal description of a finite state machine (FSM) consists of six parts:

1. A set of states, denoted as S.
2. A set of input symbols, denoted as I. These are all the possible "atomic" inputs that the machine can accept.
3. A set of output symbols, denoted as O.
4. A state transition function, denoted as δ. This function defines to what state the machine moves if it receives a certain input, so its type is $\delta : S \times I \to S$.
5. An output function, denoted as γ. This function describes what the output symbol is corresponding to an input received in a certain state, so its type is $\gamma : S \times I \to O$.
6. A starting state, denoted as s_0. This is the state in which the machine begins its operations.

In this thesis we will use $M = \{S, I, O, \delta, \gamma\}$ and $M' = \{S', I, O, \delta', \gamma'\}$ to refer to instances of FSM's. In practice, FSM's are often drawn like in Figure 1. An example of applying the formal definition to the FSM in Figure 1 is S = {s_1,s_2,s_3,s_4}, I={y,x}, O={0,1} and s_0=S_1. Examples of the state transition function and output function are $\delta(S_1, y) = S_2$ and $\gamma(S_1, y) = 0$

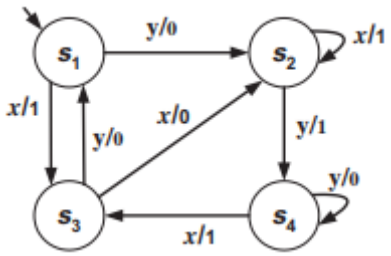


Figure 1: an example FSM, taken from [13]

The functions δ and γ can also be extended to work on sequences of inputs. We keep inputting one symbol at a time and apply the next input in the state where we ended up with the previous one. Take an input sequence aA where a represents the first input symbol and A represents the rest of the sequence. Then formally $\delta(s, aA) = \delta(\delta(s, a), A)$ and $\gamma(s, aA) = \gamma(s, a) \cdot \gamma(\delta(s, a), A)$, where $\cdot$ represents concatenation. Take for example the input sequence yxy and the FSM from figure 1. Then $\delta(s_1, yxy) = \delta(s_2, xy) = \delta(s_2, y) = s_4$ and $\gamma(s_1, yxy) = 0 \cdot \gamma(s_2, xy) = 0 \cdot 1 \cdot \gamma(s_2, y) = 011$.

Conformance testing

Conformance testing is the act of comparing a specification with an implementation of that specification. The specification is often in the form of an FSM and the implementation is often called the system under test (SUT). We assume that the behaviour of the SUT can be accurately described by some unknown FSM, this means for example that it is assumed to be deterministic. The main

purpose of conformance testing is to find differences in behaviour between the specification and the SUT. These differences correspond with bugs in the SUT, or with faults in the specification, and can be used to fix them before they are discovered by users of the SUT. There exist several techniques to prove certain properties of FSM's, but in the general case proving the correctness of software is very difficult. Conformance testing can be used to prove that the behaviour of software is semi-equivalent to that of an FSM, and that it therefore also has all the nice properties proven for the FSM.

The actual testing is done by applying a set of tests called a test suite (TS) to the SUT and then comparing the output of the SUT with the expected output described in the specification. An individual test is just a sequence of input symbols. A TS is complete (for a given M) if for every other machine M', not equivalent to M, there exists a test $t \in TS$ for which $\gamma(s_0, t) \neq \gamma'(s_0{}', t)$. This means that by applying a complete TS for M to a SUT, we can determine if the SUT is an implementation of M. It has however been proven that a complete and finite TS cannot exist if the set of input symbols is not empty. This can be illustrated by the following example from Moerman (2019) [12] seen in Figure 2. The left FSM produces only 0's as outputs. The left FSM produces n 0's and then starts producing 1's. We could create a test suite that is complete for a certain value of n, but not one that can detect the difference for any n, as n can always be made 1 higher which would require more tests in the test suite.

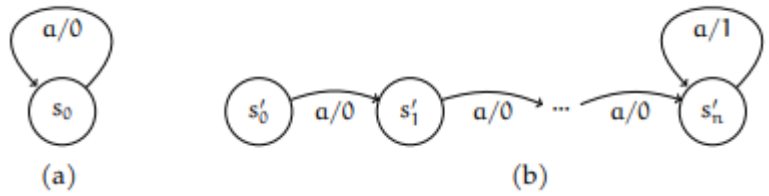


Figure 2: a) a simple FSM. b) a partial description of an FSM indistinguishable from the one on the left

This leads to a weakened definition called m-completeness. A test suite TS is m-complete for M if for any M', not equivalent to M and having at most m states, there exists a test $t \in TS$ for which $\gamma(s_0, t) \neq \gamma'(s_0{}', t)$. If a SUT passes all the tests in a test suite that is m-complete for M then that means one of two things. Either the SUT's behaviour is equal to that of M, or the SUT has more than m states. Typically, Because the developers of the SUT actively tried to make an implementation of M, most of the differences will be small mistakes where the only difference is a faulty output or a transition to the wrong state. These kinds of mistakes usually do not introduce extra states, so conformance testing is well suited to detect them. Conformance testing is however just that: testing, not proving. It is always possible that there are still undetected errors left even after passing all the tests.

Families

A family is an indexed set, where the elements are grouped in subsets and each index links to a specific subset. It can also be seen as a function from its index type to a set of its element type. A family F with states as its indices is denoted as $\{F_s\}_{s \in S}$. And F_s denotes the part of F that is mapped to s.

Operators

Below, X and Y are sets of input sequences, Z is a set of input symbols, and F is a family of input sequences

$X \cdot Y$ denotes set concatenation, which is defined as

$$X \cdot Y = \{x \cdot y \,|x \in X, y \in Y\}$$

Z^n denotes the set of sequences sampled over Z with length n. This is recursively defined as

$$Z^0 = \{\lambda\} \text{ and } Z^{n+1} = Z^n \cdot Z$$

Where λ denotes the empty sequence.

$Z^{\leq n}$ denotes the set of all sequences sampled over Z with length n or less. This is defined as

$$Z^{\leq n} = \bigcup_{i=0}^{n} Z^i$$

Z^* denotes the set of all possible sequences sampled over Z. This is defined as

$$Z^* = \bigcup_{i=0}^{\infty} Z^i$$

$\cup F$ denotes the set obtained by flattening F this set contains every element contained in one of the subsets within F. This can be defined as

$$\bigcup F = \{x|\exists\, X \in F, x \in X\}$$

$X \odot F$ is the concatenation of a set with a family. Its semantics are similar to regular set concatenation, but now for every element in X only a subset of F is used. This assumes that F can be indexed using states. The formal definition is

$$X \odot F = \{\mathrm{x} \cdot \mathrm{y} \mid \mathrm{x} \in \mathrm{X}, \mathrm{y} \in F_{\delta(s_0,\mathrm{x})}\}$$

Other definitions

State cover

A state cover is a set of input sequences that can be used to reach every state in an FSM. Formally, P is a state cover for M iff

$$\forall\, s \in S, \exists\, A \in P, \delta(s_0, A) = s$$

A state cover is used in a test suite if we want to run a test once for every state. A state cover with minimal size can be easily generated by performing breadth first search on a graph representing the transitions in an FSM, where each transition is labelled with its respective input symbol.

Transition cover

A transition cover is a set of input sequences that when executed traverses every transition in an FSM at least once. For a deterministic FSM, every state has exactly one transition for every input

symbol. So for such a machine M, a transition cover Q can be generated by executing every possible input in every possible state. Thus, if P is a state cover for M then $P \cdot I$ is a transition cover for M.

State equivalence

Two states s and s' are defined as equivalent ($\approx$) if they produce the same series of outputs for every possible series of inputs:

$$s \approx s' \leftrightarrow \forall A \in I^*, \gamma(s, A) = \gamma(s', A)$$

FSM equivalence

Two FSM's M and M' are defined as equivalent if they have the same input alphabet and their respective starting states are equal:

$$M \approx M' \leftrightarrow (I = I' \wedge s_0 \approx {s'}_0)$$

Reset operation

Most conformance testing algorithms, and all the ones discussed in this thesis, rely on the existence of a reliable reset operation (R). This reset can be seen as a special input symbol that when entered into the SUT, brings it back it to its initial state regardless of what state the SUT was in. If the SUT is a piece of software, this can be done by terminating it and starting a fresh instance. If the SUT is a machine this can often be done by cutting the power or holding the power button. This reset is required when a test suite has more than one input sequence in order to transition from one test to the next. However, not all systems have a reliable reset, which drastically reduces the available options while testing. Even if R does exist, it might be really slow compared to the other inputs. Because of this, it is usually recommended to use resets as little as possible while testing. An important note is that the reset operation is not part of I even though it is used as an input symbol. This is because it needs to be reliable and is therefore not part of the system we are testing, which is by definition not reliable.

Conformance testing algorithms that don't rely on a reset operation fall outside the scope of this thesis and are, for example, discussed in Rivest and Schapire (1993) [14]

State identifier

A state identifier is a set of input sequences that can be used to verify if the FSM is in a certain state. This is done by making sure that it gives a different set of output sequences for its target state then for any other state in the specification. Formally, a set of input sequences X is a state identifier for $s \in S$ iff

$$\forall s' \in S, s' \neq s \rightarrow \exists A \in X, \gamma(s', A) \neq \gamma(s, A)$$

Reduced FSM

An FSM is reduced if none of its states is equivalent to any of the other states, and all of its states are reachable from the starting state. If two of the states in an FSM are equivalent then it is of course impossible to generate a state identifier for them, and if states are unreachable from the starting state then they have no effect on the output of the system. Because of these 2 points, most conformance testing algorithms assume that the specification is in reduced form. This is not really a

restriction, because by using Hopcroft's algorithm [15] any deterministic FSM can be reduced in $O(|I| * n \log(n))$ time.

Fully defined FSM

An FSM is called fully defined if the transition and output functions are defined for every state input combination. In other words, the domains of both γ and δ must be $S \times I$. The FSM's we consider in this thesis are fully defined by definition.

The Wp method

The Wp method [7] is an improvement over the older W method. [5, 6], both of which are general ways to construct an m-complete test suite from several components. The Wp method requires a state cover P, a transition cover Q and a family of state identifiers $\mathcal{W}$, which contains a state identifier for every state in the specification. The actual test suite is then defined as:

$$\left(P \cdot I^{\leq k} \cdot \bigcup \mathcal{W}\right) \cup \left(\left(Q \cdot I^{\leq k}\right) \odot \mathcal{W}\right)$$

Here, k is the number of extra states we want to check for. As can be seen from the definition, the Wp method consists of two stages, which are then added together in a single test suite. The first part checks if every state in the specification is also present in the SUT. The second part then checks if every transition in the specification is also present in the SUT. The resulting test suite is m-complete for m = n+k, where n is the number of states present in the specification. The Wp method does not actually dictate how to calculate P, Q and W. When this is done in different ways, the resulting test suites can vastly differ in size. Especially the average number of resets required for the state identifiers has a big impact on the final output. In this thesis we propose a new way to calculate a set of state identifiers $\mathcal{W}$ using an adaptation of the ADS method.

Adaptive distinguishing sequences

An adaptive distinguishing sequence(ADS) is in fact not a sequence but a decision tree [1]. The nodes are labelled with a sequence of input symbols, and the branches are labelled with an output symbol. To use an ADS we enter the symbol at the root of the tree into the SUT, and then go down one of the branches based on the observed output. We keep doing this until we are at a leaf node, which then tells us in which state we were when we started. An example of such a tree belonging to the FSM in Figure 4 can be seen in Figure 3. Lee and Yannakakis (1994) [1] have given an algorithm of complexity $O(p*n^2)$ that constructs an ADS for a given FSM of maximum length n(n-1)/2. Here p is the number of different input symbols, n is the number of states in the FSM and the length of an ADS is the maximum number of input symbols needed to get from the root to a leaf node. Using an ADS for state identification results in a single, longer input instead of multiple short ones. This means that we need fewer resets and that the resulting test suite will be smaller. The main problem with ADS's is that they don't always exist. In fact, for the used benchmarks it is more likely that a given FSM does not have an ADS than that it does have one.

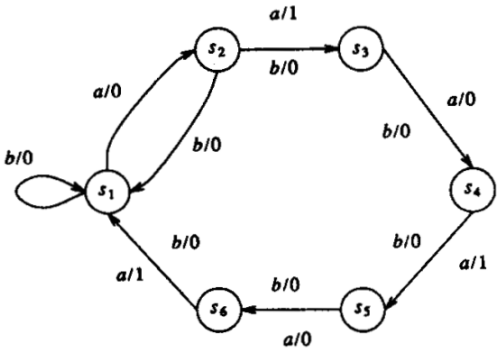


Figure 4: an example FSM [1]

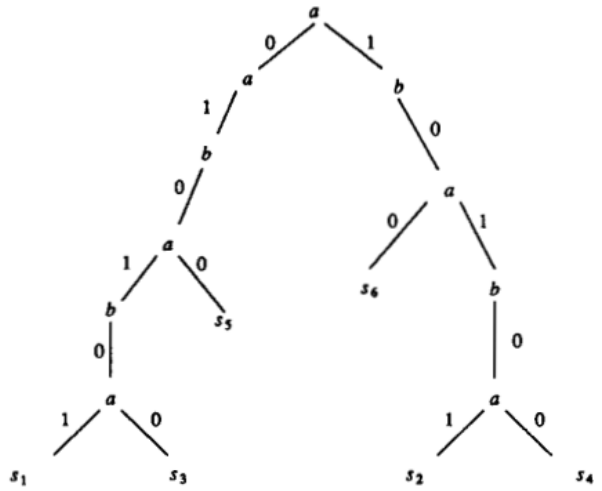


Figure 3: an ADS belonging to the FSM In Figure 4 [1]

Valid inputs

An important concept when constructing an ADS is a valid input. An input sequence A is *valid* for a set of states S iff $\forall s\ s' \in S, s \neq s' \rightarrow (\lambda(s,A) \neq \lambda(s',A) \vee \delta(s,A) \neq \delta(s',A))$. In other words, an input is valid for a set of states if no two states from this set have the same output symbol and the same output state when given that input. Take for instance the FSM in Figure 5. Applying the input a will move both s1 and s2 to s1 with output 0. We don't know in which of the two we were because the output was the same, and in both cases we are now in s1. This gives us no options for ever finding out whether we were in s1 or s2 without performing a reset. The input a is therefore called invalid. Since the whole idea of an ADS is to identify in which state we are without performing a reset, a node in an ADS can only contain input sequences that are valid for the states of that node. If we look at Figure 5 again, we see that the input b is also invalid. This leaves no possible inputs for the root of the ADS, so the FSM in Figure 5 has no ADS.

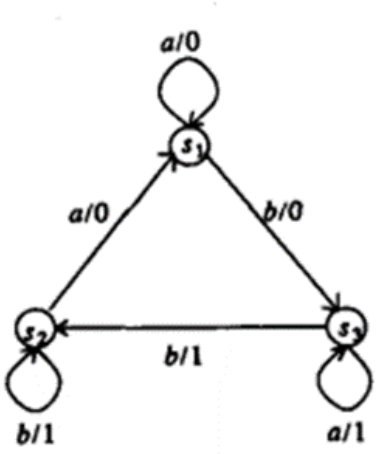


Figure 5:an example FSM without an ADS [1]

Generating an ADS

The Lee/Yannakakis algorithm [1] for generating an ADS works in two steps. First it generates a splitting tree from the specification, and then it generates an ADS from the splitting tree. The splitting tree can be seen as a simplification of the problem where the effect of applying a new input, moving to a new state, is ignored. The nodes contain a set of states and a sequence of input symbols, and the edges are unmarked. When the input sequence of a node is applied to the states of that node, some states will give different outputs than other states. The states can be grouped based on their output, where states with the same output are in the same group. Each child of a node corresponds to one of those groups. A complete splitting tree contains all the information needed to split a group of nodes down to singletons, but we can't just walk it down from the root to a leaf node because it ignores the effects of the inputs. The ADS uses the splits described in the splitting tree, but reorders them so that it does take these changes into account.

Hybrid ADS

Even when there is no ADS, the algorithm from Lee and Yannakakis can still return partial results that are useful. If there is no ADS, then it is not possible to form a complete splitting tree. The partial tree might however contain enough information to reduce the number of possible states we are in. For some states it might even contain enough information to completely separate them from the others. This partial information can be combined with the results of a different algorithm produces larger results but is applicable in more cases. These combined results can then be used for state identification more effectively than the results of the second algorithm could on their own. This idea has been fully implemented with the HSI method [8] in the hybrid-ADS method [12].

Complete-ADS

In this thesis, we take a different approach to the same problem and then compare the results. While the hybrid-ADS method focuses on complementing the incomplete splitting tree with different results, we instead try to complete the splitting tree by loosening some restrictions. After we construct the splitting tree, we use it to form a set of state identifiers, one for each state of the specification. These state identifiers can then be used to form a test suite, for instance with the Wp method.

The main change in restrictions is that we allow the use of resets, and therefore we can use invalid input sequences as a last resort. The goal is to create a state identifier for every state using as few resets as possible. However, finding such an optimal solution is p-space complete [1].To get around this problem we use a greedy approach and only try to make locally optimal choices.

The main problem to solve is how to best pick an invalid input sequence if there are no valid options left. How good or bad an invalid input is, is mainly determined by the state in which we apply it. If we look at the FSM in Figure 5 again and we want to know whether we are in s3 or not, then the input a is perfect for the job, even though it is invalid. If we were in state s1 however, b is a much better choice. To describe this more formally, we introduce the concept of partial-validity. An input sequence A is partially-valid for a state $s \in S$ iff

$$\forall\, s' \in S, s \neq s' \;\rightarrow\; (\lambda(s,A) \neq \lambda(s',A) \vee\; \delta(s,A) \neq \delta(s',A)\,)$$

As long as we are in state s, an input sequence that is partially valid for s won't require a reset and is therefore just as good as an input sequence that is completely valid. So if there are valid inputs, we use the regular ADS algorithm. When those run out and the regular algorithm would declare failure, we instead do a case distinction over all the states that are left at that node. For each of those states we try and find an input sequence that is partially-valid for that state. This essentially results in a separate splitting tree for each state, where the focus of each tree is to remain partially-valid for that state. However, sometimes it is not even possible to find an input sequence that is partially valid for a state. Since we do want to complete the splitting tree, we chose an invalid input instead. If this input splits the possible states into at least two groups, we made a little progress. Since the FSM is reduced, it is always possible to find an input sequence that does this. We can solve the invalid problem later by introducing resets when we construct the state identifiers from the splitting tree.

The resulting splitting tree now has an extra dimension. Instead of a set of children and a single separator, a node can now have multiple separators where each separator has its own set of children. Each node now also contains a mapping from states to one particular separator and a set of children, that is considered the best fit for that state.

To prevent the tree from growing exponentially, unneeded nodes can be pruned early. Every part of the tree is specifically meant to separate a subset of states from all other states. This means that child nodes in such a tree part that do not contain any of these target states are unnecessary. An example of this is a node containing s1,s2,s3,s4,s5 with target set {s1}. This can happen if there was a better separator available for the other states. The only purpose of this node is to differentiate s1 from the other states. Once this is accomplished, the children that don't contain s1 do not need to be expanded because they will never be reached. This also means that we don't have to prioritize looking for a completely valid separator, as a partially-valid one for s1 will perform equally well.

After we have constructed a splitting tree as outlined in the previous paragraphs, the next step is to calculate a set of state identifiers from it. This is done one by one, once for each state, where the current one is called the target state. We start by setting up two sets of states. The current set contains the states where we could be right now, and the initial set contains the corresponding states where we were at the beginning of the calculation. They are both initialized to contain all possible states. Then we find the lowest common ancestor (lca) node in the splitting tree that contains all the nodes in the current set. We only look at the part of the tree that is relevant for the target state, using the mappings saved at each node. We apply the separator found there to the target state and to each state in the current set, and then we concatenate it to the state identifier. Here, applying means looking up the expected output symbol and state in the specification. Each state in the current set that has a different output then the target state gets removed, along with the corresponding state in the initial set. If the outputs are the same, the state in the current set gets replaced with its output state. Then we replace the target state with its output state corresponding to the separator. Next, we start over by finding the new lca of the remaining nodes in the current set. At some point the current set will be reduced to just one state. If the initial set is still bigger than 1, it means that we need to do a reset. We concatenate a reset to the state identifier and make the current set a copy of the initial set again. At some point the initial set will contain just one state, and this will be the original target state for which we have constructed a state identifier.

These two steps of creating a splitting tree and a family of state identifiers are summarized by the pseudocode of algorithm 1 and 2 respectively. To keep the code brief it omits several steps that improve running time but do not affect the quality of the output such as when to prune unnecessary nodes.

Algorithm 1

```
Input: a reduced FSM M.
Output: a splitting tree for M.
Create the root node which contains all states and add it to the worklist;
While the worklist is not empty:
    Option1: for every node in the worklist:
        If the node only contains one state:
            Remove it from the worklist;
            Continue with the next node;
        If there exists a valid split based on output symbols:
            Do the split and add the new nodes to the worklist;
            Remove the current node from the worklist;
    Option2: for every node in the worklist:
        If there exists a valid split based on output state:
            Do the split and add the new nodes to the worklist;
            Remove the current node from the worklist;
            Break from this loop and go back to Option1;
    Option3: for the first node in the worklist:
        If there exists a partially valid split for every state in the node:
            Perform all the splits and add all the nodes to the worklist;
            Update the mapping of the node to store which split belongs to which state;
            Remove the current node from the worklist;
            Go back to Option1;
    Option4: for the first node in the worklist:
        If there exists an (invalid) split:
            Perform partially valid splits where possible;
            Use invalid splits for the other states,
            these can be found using Hopcroft's algorithm[15];
            Add all the nodes to the worklist;
            Update the mapping of the current node;
            Remove the current node from the worklist;
            Go back to Option1;
    Option5: if all else fails:
        Move the current node to the back of the worklist;
        Go back to Option3;
Return the now compete splitting tree;
```

Algorithm 2
Input: A reduced finite state machine M.
A splitting tree for M as produced by algorithm 1.
Output: A family of state identifiers for M.

1 $state\_identifiers =$ an empty list;
2 For every state *s* in the FSM:
3 $identifier =$ an empty list of input sequences;
4 *Buffer* $=$ an empty input sequence;
5 $target = s$
6 $current\_set =$ all states in the FSM;
7 $initial\_set =$ all states in the FSM;
8 Create a mapping between $current\_set$ and $initial\_set$;
9 Initialize the mapping to map identical states to each other;
10 While $initial\_set.size() > 1$:
11 Find the lca of the current set in the splitting tree using $target$;
12 $A =$ the separator found at this lca;
13 For every state s_c in $current\_set$:
14 If $\lambda(s_c, A) \neq \lambda(target, A)$:
15 Remove s_c from $current\_set$;
16 Remove the state mapped to s_c from $ininital\_set$;
17 Continue;
18 Else:
19 $s_c = \delta(s_c, A)$;
20 Update the mapping in such a way that the new s_c
21 still maps to the same state as before it got updated;
22 $target = \delta(target, A)$;
23 $Buffer = buffer \cdot A$;
24 If $current\_set.size() == 1$:
25 $current\_set = initial\_set$;
26 Reset the mapping between $current\_set$ and $initial\_set$;
27 $identifier.append(buffer)$;
28 $buffer =$ an empty list;
29 $state\_identifiers.append(identifier)$;
30 Return $state\_identifiers$;

m-Completeness

Using the family of state identifiers as described previously, we can create an m-complete test suite using the Wp-method. There exist several proofs of the m-completeness of the Wp-method. An example one using bi-simulations can be found in Moerman (2019) [12]. So in order to prove that the complete-ADS method can be used to create an m-complete test suite, we simply have to prove that the algorithm returns a family of state identifiers. This proof follows from the following lemmas:

Lemma 1: the loop from line 10 to line 28 in algorithm 2(the loop) has the following invariant: the partially calculated state identifier located in *identifier* and *buffer* can differentiate between s and every state not in *initial_set*, where differentiating means that it gives a different output sequence.

Lemma 2: when the loop terminates the initial set contains just s

Combining the two lemmas gives the following conclusion:

when the loop terminates the now fully calculated state identifier located in identifier can differentiate between s and every other state that is not s.

this means that *identifier* at that point behaves as a state identifier for s. Since the algorithm is run for every state in the specification, when it terminates it has generated a state identifier for every state in the specification, which together form a family of state identifiers.

The proof of lemma 1 requires the following additional lemma:

Lemma 3:

The lca of a set of states S has a separator A which satisfies $\exists\ s, s' \in S, \lambda(s, A) \neq \lambda(s', A)$.

Proof lemma 3:

The lca of a set of states S is the node of the splitting tree that contains all states in s, and that has no children that contain all states in S. Finding this node in the splitting tree is only defined for a certain target state s, this limits the search for the lca to just that part of the tree which is relevant for s. Since the lca is only required in algorithm 2, which has a variable $target$ of type state, any references to an lca implicitly assume that the value of this variable has been used as the target for finding the lca.

By definition of the splitting tree, applying a separator of a node to all states of that node gives at least two different output sequences, where every different output sequence groups a subset of the states together in a child node. None of these child nodes contain all states in S, and states in different child nodes have different output sequences when given A. this means that the states of S get mapped to at least two different child nodes, which means there are at least two different output sequences. ■

Proof lemma 1:

At the start of the first iteration of the loop the initial set contains every state in the specification. This automatically means that lemma 1 holds, because there is no state that is not in the initial set.

Assume that the invariant holds at the start of the loop and that the separator for the lca of the current set is A. Then there is at least one state s_c in the current set for which $\lambda(s_c, A) \neq \lambda(target, A)$, which follows from lemma 3. Because the target, and the elements of the current set, get updated every time something gets added to *buffer*, the target is equal to $\delta(s, buffer)$, and the element in the current set that is linked to a state s_i in the initial set is equal to $\delta(s_i, buffer)$. Since the output for s_c was not equal to the expected output $\lambda(target, A)$, it is removed from the current set, and its corresponding state s_i in the initial set is also removed. So $s_c = \delta(s_i, buffer)$, target = $\delta(s, buffer)$, and A differentiated between the two, which means that $\lambda(s, buffer \cdot A) \neq \lambda(s_i, buffer \cdot A)$. This means that lemma 1 still holds at the end of the loop after s_i has been removed from the initial set and A has been concatenated to $buffer$. ■

Proof lemma 2:

At every step of the loop, at least one state s_i gets removed from the initial set. Following from the proof of lemma 1, this is the state for which $\lambda(s, buffer \cdot A) \neq \lambda(s_i, buffer \cdot A)$. It is easy to see

that this statement can never hold for s_i = s. since the loop continues repeating until the size of the initial set is just 1, this means that in the end only s remains in the initial set. ■

Termination

algorithms 1 and 2 will always terminate for any given valid input. A proof of this is given below.

Lemma 4: Algorithm 1 will always terminate for any given valid input.

Proof:

Every time a split is found and executed for a node, that node is removed from the worklist and the new child nodes are added in. Nodes containing only one state are removed from the list without adding new nodes. Child nodes are per definition smaller than the parent: they contain a strict subset of the states of the parent node. This means that as long as we keep finding splits for every node, eventually the worklist will be empty, and the algorithm will terminate. Because the input is a reduced FSM, it is per definition always possible to find an input sequence that differentiates between two non-equal states of the FSM. It is therefore always possible to find a split for any given subset of states, and thus also for any given node. So algorithm 1 always terminates. ■

Lemma 5: Algorithm 2 will always terminate for any given valid input.

Proof:

With every iteration of the loop at line 10, all states in $current\_set$ that give an unexpected output are removed from $current\_set$. There is only one expected output, and because of lemma 3 there are at least two different observed outputs. This means that $current\_set$ gets strictly smaller with each iteration. When $current\_set$ gets smaller, $initial\_set$ also gets smaller because the same number of states is removed in line 16. If $current\_set$ reaches size one and can't get smaller anymore, it gets refilled to the same size as $initial\_set$ in line 25. The loop terminates when $initial\_set$ reaches size one, and with every iteration of the loop $initial\_set$ get strictly smaller. This means that after a finite number of iterations the initial_set reaches size one and the loop terminates. The other loop at line 2 is a simple for each loop over a constant set so that always terminates if its body terminates. This means that algorithm 2 always terminates. ■

Comparison

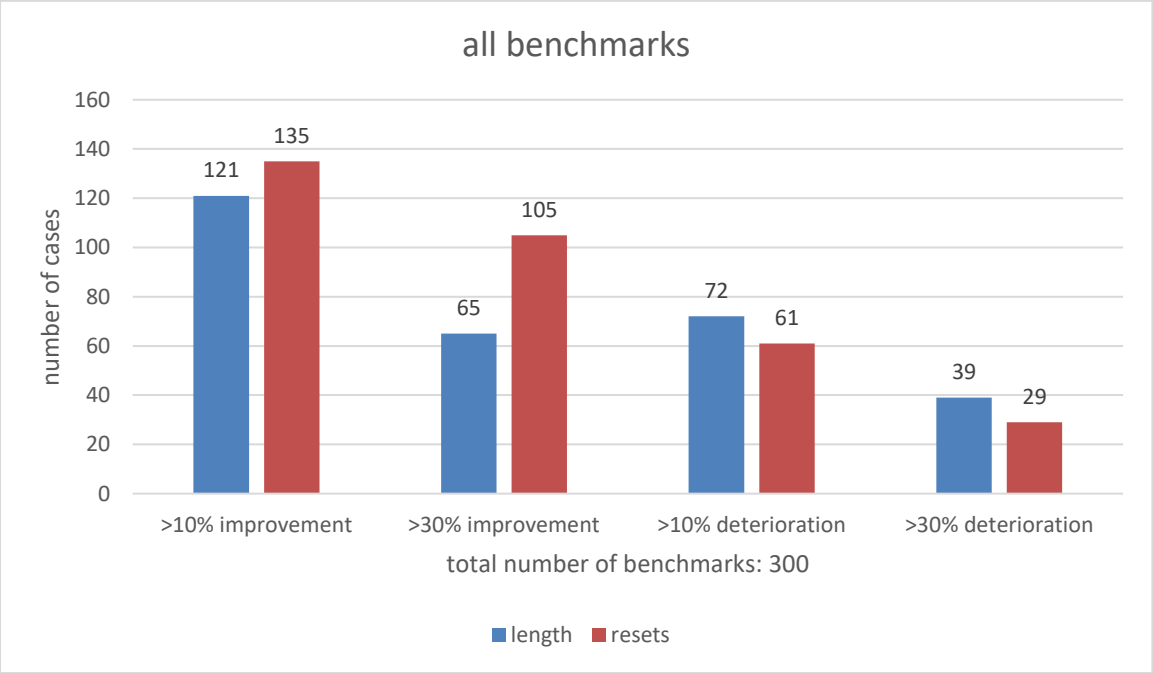
The complete ADS method can be used to generate an m-complete test suite, but it is certainly not the only way. In this section we compare our results with that of a different approach: the hybrid ADS method.

Results

The complete ADS method has been fully implemented so that its results can be compared with those of the hybrid ADS method. The hybrid ADS method was chosen to compare with because it works very well in practice and it uses similar ideas. For the comparison an m-complete test suite was generated for every compatible benchmark in the Radboud automata wiki [9], where compatible means fully defined, deterministic, reduced finite state machines with more than 1 state. Out of the 335 FSM benchmarks found at the wiki 300 passed these criteria. After generating the test suites, the results of the hybrid ADS method and the complete ADS method where compared on

terms of length and number of resets, where the length of a test suite is defined as the number of input symbols it contains plus the number of resets it contains. In order to find relations and trends in the large amount of data, the data was grouped based on performance increase or decrease of the complete versus the hybrid ads method. This grouping was done for different subsets of the data, such as all benchmarks or only the ones with more than 200 states etc. Because the raw comparison results are too long to be included in this thesis, even as an appendix, they are located at the code repository in the Raw_data folder [10].

A graph showing some of the data for the whole dataset can be seen below. The graph shows for instance that there are 121 benchmarks where the length of the test suite generated by the complete ads method is 10% shorter, or an improvement, compared to that of the hybrid ads method, as seen in the first blue bar. The graph also shows that there are 72 cases where the length of the complete test suite is in fact 10% longer, or a deterioration, as shown by the third blue bar. All values are cumulative, so a test suite that is 30% better is also 10% better etc.

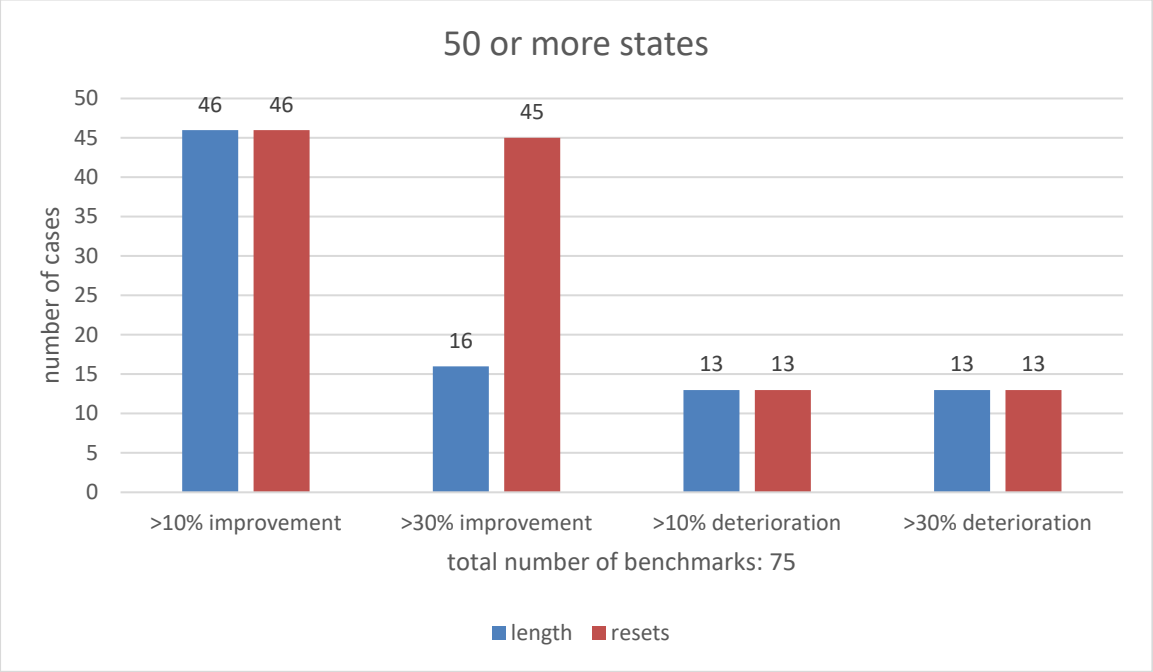


Graph 1

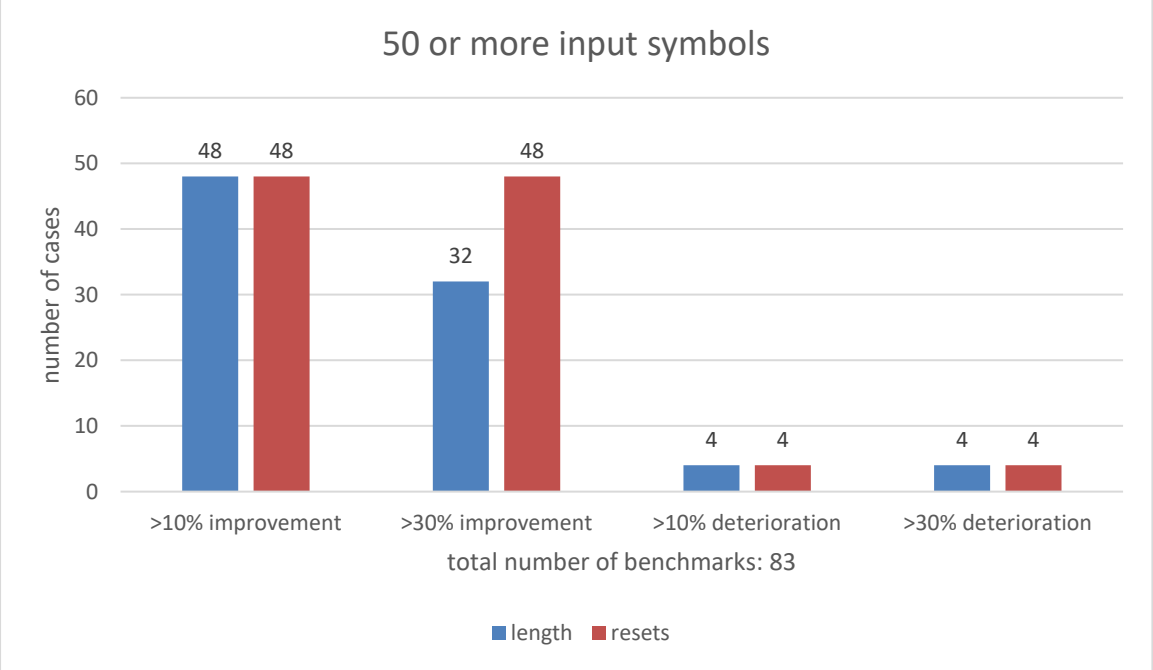
Graph 1 shows that there are more cases where the complete ADS method gives an improvement than where it deteriorates performance. It also shows that the gain in resets is generally higher than the gain in length, which makes sense because the complete ADS method tries to minimize the number of resets, but does not optimize for length at all. There is also a length improvement visible because a lower number of resets in the state identifiers generally leads to a shorter test suite.

But even though the general trend seems to be one of improvement, there are still a lot of cases where performance deteriorates significantly. Looking at the raw data there are even cases where this is almost 200% which means that the resulting test suite is three times as long as the hybrid ADS one.

The benchmarks that show improvement seem to have some common characteristics however, as can be seen in Graph 2 and Graph 3. When Just looking at the bigger benchmarks in terms of either number of states or input symbols the ratio between improvement and deterioration shifts enormously.



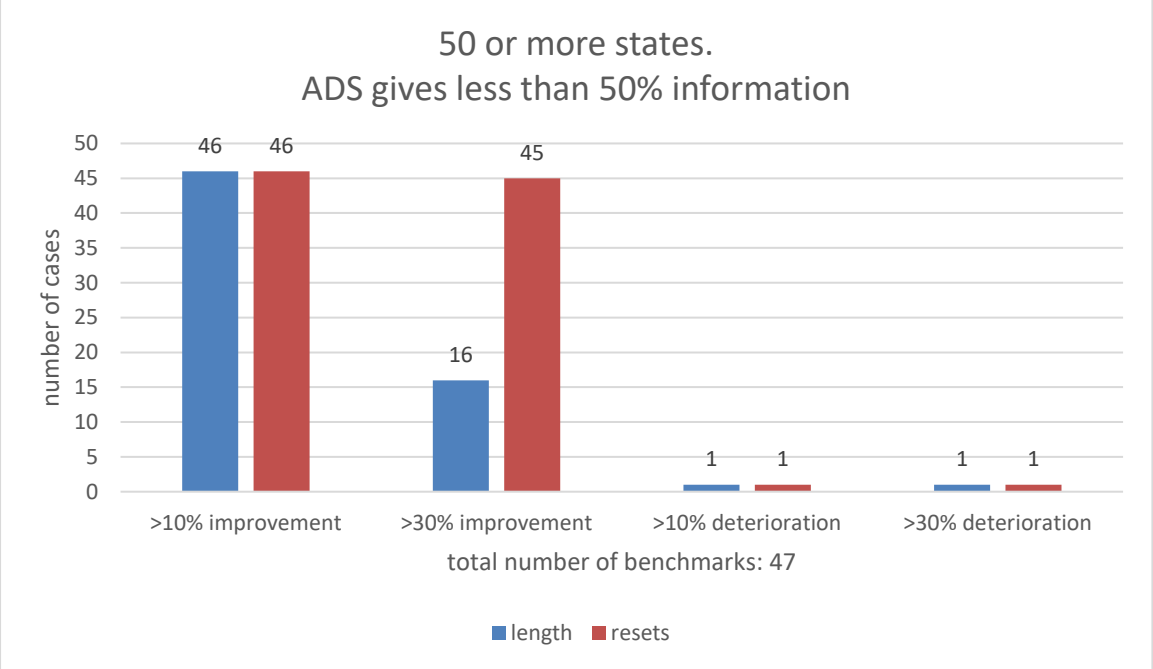
Graph 2



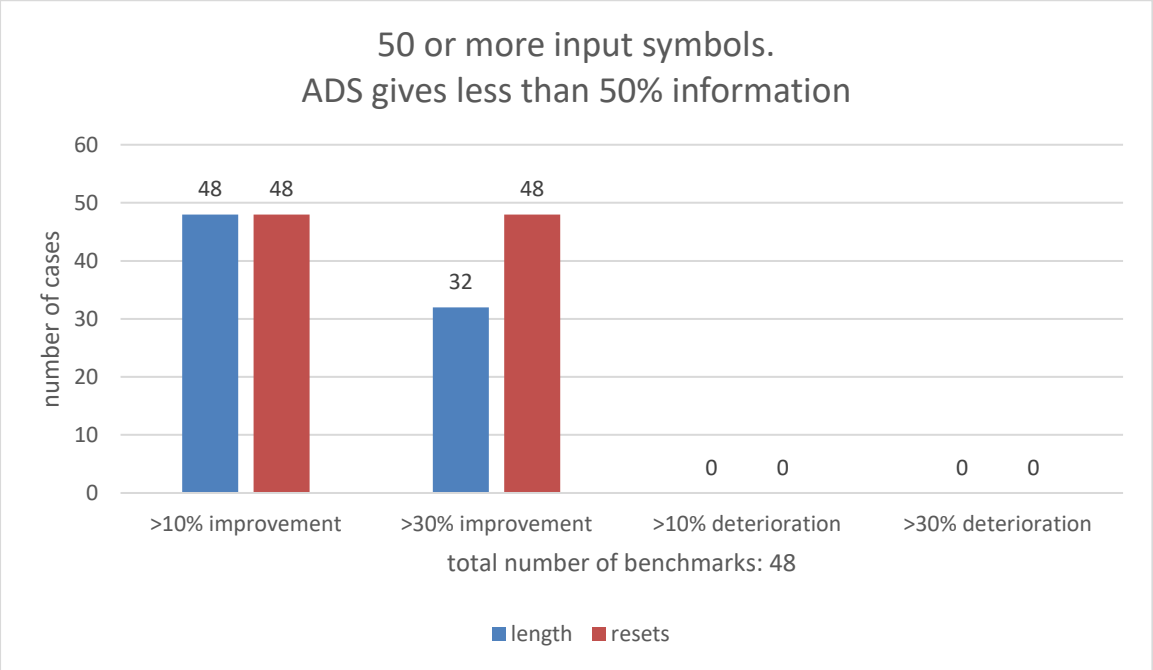
Graph 3

But this still ignores one important aspect: the applicability of the ADS method. The whole idea behind the complete ADS method is that it can be applied where the regular ADS method cannot. Partial information gained by the ADS method can be used to identify a state as a member of a

subset of the complete set of states. Graph 4 and Graph 5 show data about the cases where this subset is bigger than half the size of the complete set of states. This means that the information gain of the ADS method was less than 50%. This seems like a rather steep requirement, but by comparing Graph 1,2,3 and 4 with each other we see that this holds true for about 60% of all decently sized benchmarks. After filtering this way, Graph 4 and 5 show that the complete ADS method gives an improvement in almost all of the remaining cases.



Graph 4



Graph 5

Conclusion

In the general case the complete ADS method gives an improvement over the hybrid ADS method more often than not. The complete ADS method does better on FSM's that have a large number of states, and even better on FSM's that have a large number of input symbols. If we only look at the cases where the ADS method does not give much information, then there is a significant performance gain in almost 100% of the remaining cases. Since the number of resets decreased a lot more than the length of the test suite, this performance gain will be even more pronounced in situations where resets are more expensive than a single input symbol.

Future work

Calculating the information gain of the ADS method is much easier than generating an m-complete test suite, so it seems possible to formulate a general rule that determines which method will give better results based on the ADS information gain, the number of states and the number of input symbols of an FSM. This rule could then be used to choose the best method for each specific case. Specifying and testing such a rule will however not be a part of this thesis due to time constraints.

Bibliography

1. Lee, D., and Yannakakis, M.: 'Testing finite-state machines: state identification and verification', IEEE Transactions on Computers, 1994, 43, (3), pp. 306-320. DOI:10.1109/12.272431
2. Endo, A.T., and Simao, A.: 'Model-Based Testing of Service-Oriented Applications via State Models'. Proc. Proceedings of the 2011 IEEE International Conference on Services Computing2011 pp. 432-439. DOI:10.1109/scc.2011.77
3. Smeenk, W., Moerman, J., Vaandrager, F., and Jansen, D.N.: 'Applying Automata Learning to Embedded Control Software', 2015, 9407, pp. 67-83. DOI:10.1007/978-3-319-25423-4_5
4. Lee, D., and Yannakakis, M.: 'Principles and methods of testing finite state machines-a survey', Proceedings of the IEEE, 1996, 84, (8), pp. 1090-1123. DOI:10.1109/5.533956
5. Chow, T.S.: 'Testing Software Design Modeled by Finite-State Machines', IEEE Transactions on Software Engineering, 1978, SE-4, (3), pp. 178-187. DOI:10.1109/TSE.1978.231496
6. Vasilevskii, M.P.: 'Failure diagnosis of automata', Cybernetics, 1973, 9, (4), pp. 653-665. DOI:10.1007/BF01068590
7. Fujiwara, S., v. Bochmann, G., Khendek, F., Amalou, M., and Ghedamsi, A.: 'Test selection based on finite state models', IEEE Transactions on Software Engineering, 1991, 17, (6), pp. 591-603. DOI:10.1109/32.87284
8. Luo, G., Petrenko, A., and v. Bochmann, G.: 'Selecting Test Sequences for Partially-Specified Nondeterministic Finite State Machines', in Mizuno, T., Higashino, T., and Shiratori, N. (Eds.): 'Protocol Test Systems: 7th workshop 7th IFIP WG 6.1 international workshop on protocol text systems' (Springer US, 1995), pp. 95-110. DOI:10.1007/978-0-387-34883-4_6
9. http://automata.cs.ru.nl/Overview#Mealymachinebenchmarks, accessed October 22 2018.
10. https://gitlab.science.ru.nl/gvcuyck/complete-ads, accessed December 25 2018.
11. https://gitlab.science.ru.nl/moerman/hybrid-ads, accessed October 27 2018.
12. Moerman, J.: 'Nominal techniques and black box testing for automata learning' (phd thesis, Radboud University, To appear: 2019, chapter 2)
13. Dorofeeva, R., El-Fakih, K., Maag, S., Cavalli, A.R., and Yevtushenko, N.: 'FSM-based conformance testing methods: A survey annotated with experimental evaluation', Information and Software Technology, 2010, 52, (12), pp. 1286-1297. DOI:10.1016/j.infsof.2010.07.001
14. Rivest, R.L., and Schapire, R.E.: 'Inference of Finite Automata Using Homing Sequences', Information and Computation, 1993, 103, (2), pp. 299-347. DOI:10.1006/inco.1993.1021
15. Hopcroft, J.: 'An n log n algorithm fro minimizing states in a finite automaton', in Kohavi, Z., and Paz, A. (Eds.): 'Theory of Machines and Computations' (Academic Press, 1971), pp. 189-196. DOI:10.1016/B978-0-12-417750-5.50022-1

Appendix A: variable types

The following list contains the type of every variable used in the thesis

Variable:	type
a,b:	input symbol
Z,A,t,$buffer$:	sequence of input symbols
X,Y,$identifier$:	set of input sequences
λ:	the empty sequence.
I:	set of input symbols
O:	set of output symbols
$state\_identifiers$:	set of state identifiers
M,M':	FSM
TS:	test suite
P:	a state cover, which is a set of input sequences.
Q:	a transition cover, which is a set of input sequences.
s,s',s_0,target:	state
s_c	state from the current set
s_i	state from the initial set
S,$current\_set, initial\_set$:	set of states
Z_s:	state identifier for s
x,y:	untyped elements of generic sets.
F:	family of input sequences
δ:	$S \times I \rightarrow S$
γ:	$S \times I \rightarrow O$
n:	natural number