This is a repository copy of *State identification sequences from the splitting tree*.

White Rose Research Online URL for this paper:
https://eprints.whiterose.ac.uk/158336/

Version: Accepted Version

Article:

Soucha, M. and Bogdanov, K. (2020) State identification sequences from the splitting tree. Information and Software Technology, 123. 106297. ISSN 0950-5849

https://doi.org/10.1016/j.infsof.2020.106297

Article available under the terms of the CC-BY-NC-ND licence (https://creativecommons.org/licenses/by-nc-nd/4.0/).

Reuse

This article is distributed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivs (CC BY-NC-ND) licence. This licence only allows you to download this work and share it with others as long as you credit the authors, but you can't change the article in any way or use it commercially. More information and the full terms of the licence here: https://creativecommons.org/licenses/

Takedown

If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk/

State Identification Sequences from the Splitting Tree

Michal Soucha*, Kirill Bogdanov*

Department of Computer Science, The University of Sheffield, UK

Abstract

Context: Software testing based on finite-state machines.

Objective: Improving the performance of existing testing methods by construction of more efficient separating sequences, so that states entered by a system under test can be identified in a much shorter span of time.

Method: This paper proposes an efficient way to construct separating sequences for subsets of states for any deterministic finite-state machine. It extends an existing algorithm that builds an adaptive distinguishing sequence (ADS) from a splitting tree to machines that do not possess an ADS. Our extension to this construction algorithm allows one not only to construct a separating sequence for any subset of states but also form sets of separating sequences, such as harmonized state identifiers (HSI) and incomplete adaptive distinguishing sequences, that are used by efficient testing and learning algorithms.

Results: The experiments confirm that the length and number of test sequences produced by testing methods that use HSIs constructed by our extension is significantly improved.

Conclusion: By constructing more efficient separating sequences the performance of existing test methods significantly improves.

Keywords: Splitting tree, separating sequence, harmonized state identifiers, adaptive distinguishing sequence, finite-state machine, software testing, regular inference

1. Introduction

Software testing takes a significant amount of time, so effective testing methods can improve the quality of software, the cost of development and the cost of execution as well. Testing methods for finite-state machines are known for their capability of both finding subtle defects and for theoretical guarantees of fault-finding. The downside of these methods is the amount of testing that has to be completed before any of the claimed guarantees can be attained. A significant contribution to the amount of testing is the number and length of what is known as separating sequences. These are the sequences of inputs derived from a finite-state model of a specification that are intended to identify states entered by an implementation during testing. For instance, clicking on a link on a web page to submit an order should place such an order and empty a basket, so that attempting to submit the same form twice would not result in a duplicate order. Knowing whether a basket is indeed empty might not be visible on a 'thank you for your order' page so for most web sites a tester has to navigate back on the main page to check the contents of the basket and possibly check the list of orders to observe that an order has indeed been recorded as placed. This is important: verification that a correct state has been entered requires interaction with a system under test with effects not immediately visible. Construction of sequences that efficiently verify states is the subject of this paper. The intention is to make it possible to construct efficient sequences to separate any chosen subset of states in a specification, because usually it is known which states may or may not be entered at the end of a sequence of commands.

The algorithm proposed is an extension to an existing algorithm that aims to build a splitting tree (ST) for finite-state machines (FSM) possessing what is known as an adaptive distinguishing sequence (ADS). An ADS is in fact a set of sequences with common prefixes. It is best represented with a tree where nodes are named with inputs and branches carrying outputs. States are identified by walking through this tree starting from the root: inputs are submitted to a system under test and depending on outputs observed, a branch is taken. An input corresponding to the entered node is sent to a system under test next and an output determines the next node. Every leaf of this tree is associated with a state in a model which is uniquely identified by the sequence of inputs/outputs from the root of the tree to the leaf. Such sequences from root to leaf are called state verifying sequences (SVS).

The existing algorithm is limited to a specific range of FSMs where an ADS can be constructed. Where it cannot be built, one has to use more than a single sequence

*Corresponding author

Email addresses: michal.soucha@gmail.com (Michal Soucha), k.bogdanov@sheffield.ac.uk (Kirill Bogdanov)

to identify states. Most existing testing methods support multiple sequences to identify states, although they are obviously more efficient where an ADS is available.

The extension to the original algorithm presented in this paper builds a small number of sequences to separate states if an ADS does not exist and produces an ADS where it does. It can be used both for identification of individual states and for separation of a group of states (where a tester knows that only some states may be entered by the implementation but does not know which exactly). The described method also permits harmonized state identifiers (HSI) to be constructed which lead to test sequences shorter than those constructed using traditional testing methods [1, 2]. This dramatically improves the efficiency of testing methods relying on HSI sequences.

Section 2 defines used terms and Section 3 summarizes the related work. Section 4 sketches the benefits of our extension. Section 5 introduces the structure of splitting tree and what a valid input means. Algorithms for the construction of separating sequences, HSIs and IADSs from the splitting tree are proposed in Section 6. Section 7 proposes our extension that is then described on a running example in Section 8, Section 9 describes experiments on randomly-generated machines and the paper is concluded in Section 10.

2. Preliminaries

This section defines the type of finite-state machines, state identification sequences and testing considered in this paper.

2.1. Finite-State Machine

A finite-state machine (FSM) is a model consisting of states and transitions between states. According to the received input, an FSM changes its current state and responds with the corresponding output. There are many different definitions of finite-state machines in the literature. This section proposes a general model called *deterministic finite-state machine* (DFSM) that has outputs both on states and on transitions, permitting the results of this work to be used for Mealy and Moore machines as well as for deterministic finite automata.

There are two functions that describe the behaviour of a model, a transition and an output function. Both functions take an input symbol and respectively produce a next state (a state where the transition leads to) and an output symbol that is observed if the transition is taken. Two special symbols are introduced to cover both Moore and Mealy machines in one definition. An input symbol $\uparrow$ called *stOut* requests the state output and the current state of the machine is assumed to remain unchanged when it is used. An output symbol $\downarrow$ called *noOut* represents 'no response'.

Definition 1. *A deterministic finite-state machine (DFSM) is a septuple $(S, X, Y, s_0, D, \delta, \lambda)$, where S is a finite nonempty set of states and s_0 is an initial state ($s_0 \in S$). Set X is an input alphabet (a finite nonempty sets of symbols, $\uparrow \notin X$), Y is an output alphabet, D is a domain of defined transitions; $D \subseteq S \times X$, $D_\uparrow = D \cup (S \times \{\uparrow\})$, δ is a transition function $\delta : D_\uparrow \to S$ such that $\forall s \in S : \delta(s, \uparrow) = s$, and λ is an output function $\lambda : D_\uparrow \to Y \cup \{\downarrow\}$.*

Note that the stOut $\uparrow$ is not in the input alphabet X so that it differs from all other input symbols. Similarly, the noOut $\downarrow$ output can be declared outside the output alphabet Y so as not to interfere with other output symbols but it is usually matched to the output of 'no output' or 'timeout' that is in Y. Therefore, it is not specified if $\downarrow$ is or is not in Y. The timeout output represents that no response is observed during the predefined time limit. Strings over $X \cup \{\uparrow\}$ are called *input sequences* and strings over $Y \cup \{\downarrow\}$ are called *output sequences*. 'Input' and 'output' are sometimes omitted so only 'sequence' is used if it is clear from the context. The empty string is denoted with ε.

Transitions are labelled with input and output symbols. The next state, or the target state, of a transition is defined by the transition function δ and the function λ assigns an output symbol to the transition. This paper works only with *completely-specified* machines, that is, DFSMs that have all transitions defined; $D = S \times X$. The transition function δ and the output function λ can be extended to work over input sequences. The extended transition function δ^* returns the target state reached by following the path labelled with the given input sequence. The output sequence formed of labels on this path is returned by the extended output function λ^*. If a transition on the path is not defined, the path and both functions are undefined. Otherwise, both functions are defined inductively as follows. If nothing is asked, then the machine stays in the same state and no response is observed so $\delta^*(s, \varepsilon) = s$ and $\lambda^*(s, \varepsilon) = \varepsilon$ for any state $s \in S$. If a sequence $x \cdot v$ is queried, then both functions follow the first symbol x and process the suffix v from the next state, that is, $\delta^*(s, x \cdot v) = \delta^*(\delta(s, x), v)$ and $\lambda^*(s, x \cdot v) = \lambda(s, x) \cdot \lambda^*(\delta(s, x), v)$ for all $(s, x) \in D_\uparrow$ and v is an input sequence consisting of defined transitions. In addition, the transition and output function with their extended versions can be applied to a set of states and a set of sequences, that is, $\gamma(S', U) = \{\gamma(s, u) \mid s \in S' \wedge u \in U\}$ for all $\gamma \in \{\delta, \lambda, \delta^*, \lambda^*\}$, $S' \subseteq S$ and $U \subseteq X_\uparrow^*$ such that $|u| = 1$ for all $u \in U$ in the case of δ and λ functions.

Any algorithm that works with DFSMs defined in Definition 1 can also handle DFA, Mealy and Moore machines. In the case of Mealy machines, the stOut input $\uparrow$ is omitted as Mealy machine has no state outputs (it is defined in terms of Definition 1 as $\delta(s, \uparrow) = \downarrow$ for all states $s \in S$). In the case of Moore machines, the stOut input is needed to obtain the output of the initial state but then it could be considered that $\uparrow$ is asked right after any input $x \in X$

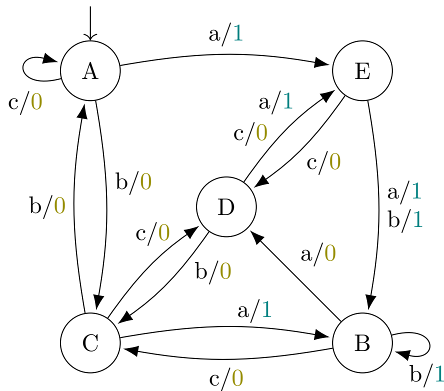


Figure 1: Mealy machine without an ADS

and so the output is obtained in response to the asked x, that is, $\lambda(s, x) = \lambda_{Moore}(\delta(s, x))$. Any deterministic finite automaton can be considered a Moore machine with just two outputs that divide the states into accepting and rejecting. An example of a Mealy machine with 5 states, A–E, 3 inputs, a–c, and 2 outputs, 0 and 1, is shown in Figure 1.

2.2. State Identification Sequences

A sequence u is *separating* for states s_i and s_j if the states provide different response to u, that is, $\lambda^*(s_i, u) \neq \lambda^*(s_j, u)$. The states s_i and s_j are then called *distinguishable*. Separating sequences get special names if they are grouped according to the following properties:

- *State verifying sequence* (SVS), or *unique input output sequence*, of state s is a sequence u such that u separates s from all other states, that is, s responds uniquely to u.
- *Adaptive distinguishing sequence* (ADS) is a set of SVSs of all states such that common prefix of any two SVSs separates the states relating to the SVSs.
- A set of separating sequences that separate all pairs of states is a *characterizing set* (CSet), sometimes denoted W.
- *Harmonized state identifiers* (HSI) are sets of separating sequences such that each set is associated with a particular state and for each pair of states s_i and s_j their separating sequence w is a prefix of a sequence in the HSIs of both s_i and s_j.
- *Incomplete adaptive distinguishing sequence* (IADS) is like an ADS of a subset of states, that is, not all pairs of states need to be separated and the used separating sequences do not have to be SVSs.

IADS is a generalization of ADS such that a state can be separated by several sequences instead of a single SVS. Note that many DFSMs have no ADS because they contain a state without an SVS. In contrast, it is always possible to form a CSet, HSIs and IADSs of any minimal DFSM. A DFSM is *minimal* if every pair of states is distinguishable and every state is reachable by a path from the initial state. A DFSM has an ADS if and only if it has HSIs such that each HSI contains just one sequence (an SVS) [3].

Algorithms dealing with finite-state machines frequently need to handle pairs of states, or *state pairs*, and store information related to them. A *state pair array* (SPA) is introduced for this purpose. The content of an SPA can be arbitrary but it relates to a particular machine M with n states. The size of SPA is always $\frac{n \cdot (n-1)}{2}$ that is about a half of the size of state pair table that has n rows and n columns.

2.3. Testing of Finite-State Machines

The specification of a system can be used for testing purposes such that a procedure called *testing method* explores the specification and constructs a *test suite*. A test suite usually consists of several sequences of inputs called *test sequences* or *tests*. The process of testing means that the constructed test sequences are successively applied to the implementation of the system. If the response to any test differs from the output given by the specification, then a discrepancy between the specification and the implementation is found; one can say that a fault in the implementation is revealed. A test suite is m*-complete* for a specification M if it can reveal a fault in the implementation that has at most m states and differs from M behaviourally. Testing methods considered in this paper work with specifications that are modelled with minimal DFSMs with n states (usually $n < m$).

FSM testing methods, such as the Vasilevskii-Chow W-method [4, 5], construct tests by exploring the state-transition diagram so that all states are visited and all transitions are attempted. In addition, target state of every transition is verified. Where the expected number of states in an implementation is potentially greater than in a specification ($m > n$), which could be related to redundancy in an implementation, the size of a test suite may increase exponentially because it is not known in advance how to reach those extra states in order to test transitions from them. For this reason, a test suite has to contain all sequences of length $m - n$ to reach these 'clone' states and then run tests to verify that they are indeed clones of the original states. The W-method is one of the early and one of the first testing methods which is proven to find all faults (that is, it is m-complete) given that an implementation has a known alphabet and a bound on the number of states. Since then a range of rather more efficient testing methods have been developed which generate much fewer and/or shorter test sequences than the W-method. This is made possible with sequences that do multiple things at the same time, such as testing transitions and verifying states reached earlier. Despite advances in test generation, the efficiency of all these methods depends on the effective identification of states.

3. Related Work

Separating sequences were initially introduced to form a characterizing set (CSet) that uniquely identifies any state. The easiest way to construct such a set of sequences was to group the separating sequences of all state pairs. Probably, the first algorithm that constructs separating sequences of all state pairs was based on the minimization algorithm [6, Algorithm 4.1]. More than 50 years later, the algorithm designing the shortest separating sequences (SSS) of all state pairs without the need of a minimization procedure was described in [7]. The state-of-the-art construction algorithm of the shortest separating sequences of all state pairs was then proposed in [8]. It is referred to as the ST-MSS algorithm in this paper as it builds separating sequences of minimal length (Minimal Separating Sequences) and they are stored in a splitting tree (ST).

The structure of a splitting tree was proposed in 1994 [9] but was not designed to construct the shortest separating sequences. Instead, it was used as an intermediate structure to build an adaptive distinguishing sequence (ADS). If the given machine has no ADS, the construction of the splitting tree detects it. The splitting tree and ADS will be described in detail in the following sections.

The use of harmonized state identifiers (HSI) was shown to be more beneficial than the use of CSet. Unfortunately, there was only one formally-described algorithm designing HSIs and it constructs HSIs from a given CSet [10, Appendix II]. Recently, three new algorithms were proposed. The first one builds HSI from the shortest separating sequences of all state pairs that are constructed by the SSS algorithm [11]. The second one collects the minimal separating sequences stored in the splitting tree constructed by the ST-MSS algorithm [8]. A completely different approach to the construction of HSIs is proposed in [3]. The authors of [3] noticed the correspondence between HSIs and adaptive distinguishing sequence (ADS) and used it to build HSIs from incomplete adaptive distinguishing sequences (IADS). IADSs are first constructed directly by the greedy algorithm. In contrast, the extension proposed in this paper allows one to construct both HSIs and IADSs directly from the splitting tree.

Testing of deterministic finite-state machines has a long history. The well-known testing method, the W-method [4, 5], uses a characterizing set to identify states. The Wp-method [12] improves the efficiency of the W-method but still uses CSet. The HSI-method [1] and the SPY-method [2] are representatives of testing methods that identify states using HSIs. There are other testing methods such as the H-method [13] and the SPYH-method [14] that do not use predefined sets of separating sequences but choose the sequences on the fly.

4. Motivation Example

The previous section mentioned four existing methods constructing harmonized state identifiers (HSI). These methods are compared with the approach based on our new extension ST-IADS on the Mealy machine defined in Figure 1. The machine has no ADS and so some HSIs need to contain more than one sequence. Table 1 shows the comparison of the constructed HSIs. The splitting tree in Figure 2 is created by both the ST-MSS and the new ST-IADS methods. They follow different construction procedures but for this small example the resulting splitting tree is the same. Nevertheless, HSIs constructed from the ST are different. HSIs from ST-MSS are equal to the ones produced from SSS. HSIs from ST-IADS are equal to the ones built from the CSet {aa, b, cb}. The approach based on incomplete adaptive distinguishing sequences constructs different HSIs than all other methods. The differences of the constructed HSIs are highlighted in bold in Table 1. The HSIs are compared on the average numbers of sequences and inputs per state. The numbers are below each of the three HSIs and the best values, that is, the minimal ones, are highlighted in bold. One can see that it is not good to base the construction of HSIs on the shortest separating sequences.

Harmonized State Identifiers

State	From SSS and ST-MSS	From IADSs	From CSet and ST-IADS
A	aa, cb, **b**	**b**aa, cb	aa, cb
B	a	a, **b**	a
C	aa, **b**	**b**aa	aa, **b**
D	aa, cb, **b**	**b**aa, cb	aa, cb
E	a, b	a, b	**a**a, b
Numbers of sequences and inputs per state			
sequences	2.2 (11/5)	**1.8** (9/5)	**1.8** (9/5)
inputs	3.2 (16/5)	3.4 (17/5)	**3.0** (15/5)

Table 1: Construction of HSIs – a comparison of five approaches

The ST-IADS algorithm seems to be promising in the construction of HSIs, however, it is not the only task in which it could excel. Assume that state C of the Mealy machine (Figure 1) needs to be separated from states A and E, and then state E is to be distinguished from states A and B. In the former case, one would like to obtain the state verifying sequence 'baa' of state C that uniquely identifies C amongst all states. Although state E has no state verifying sequence, sequence 'aa' separates it from A and B which is the requirement of the latter case. No HSI construction method produces both sequences for the aforementioned states, however, they can be easily constructed from ST-IADS as Section 6 describes.

5. Splitting Tree for Incomplete Adaptive Distinguishing Sequences

The splitting tree (ST) was originally described in detail in [9]; a very similar structure without sequences inside nodes can also be found in [6]. An algorithm separating

states builds an ST which can then be turned to IADS in order to easily separate states during testing. The description below introduces both an ST and IADS built from ST.

5.1. ADS and IADS

At the beginning of state identification an FSM can be in any state so the *initial uncertainty* (in terms of [9]) is the whole set of states, $\{A, B, C, D, E\}$ for the running example in Figure 1. The idea of an ADS is that one starts with such an initial uncertainty and then attempts an input associated with the root node. Different outputs from FSM in response to this input make it possible to split this set of states into subsets, associated with subsequent nodes in this ADS. This corresponds to a reduction of uncertainty. Subsequent nodes are also associated with inputs and permit further splitting of the set of states, further reducing the uncertainty. In this way, following a path through an ADS from the root node to a leaf node narrows down the set of possible starting states. For ADS, this process continues until such a set becomes a singleton by the time a leaf is reached. This can be seen in tree T_2 shown in Figure 3 where the initial uncertainty is called the *initial states* $\{A, D\}$ and shown at the top of the root node r_8. The initial input 'c' is such that both states respond with a 0 so the next node r_9 has the same initial states; the subsequent input 'b' separates the reached states: the output 0 corresponds to the initial state A and 1 - to the initial state D. These states are shown at the top of nodes r_{10} and r_{11}. During a walk through an ADS, the FSM changes states so during construction of ADS one has to keep track of the current state. States of FSM reached at each node of an ADS are called *current states* and shown at the bottom of each node. At the start, current states are the same as the initial states; after input 'c', the FSM would make a transition from A to A and D to E. Therefore, current states of node r_9 are A, E. After input 'b' and output 0, the only possible initial state is A and the current state is C; in a similar way, output 1 singles out state D and the current state becomes B. In other words, initial states for any ADS node reflect what remains to separate if we followed a path from the root of the ADS to this node and the current states reflect the FSM states that are entered when such a path is followed.

IADS have the same structure and a path through an IADS from the root node to a leaf node also narrows down the set of possible starting states, however, a leaf is not required to have a singleton set of initial states. This is the reason there can be multiple IADSs so that where one reaches a leaf of an IADS, it is possible to continue by resetting a system under test, re-running a test to enter a state of interest and then attempting a different IADS, chosen so that the set of initial states in the root of this IADS is equal to the initial states in the earlier leaf. After this, the process of reducing the initial uncertainty continues until either a leaf node is encountered in the new IADS or the next IADS is started until eventually a leaf node with a singleton set of initial states is entered.

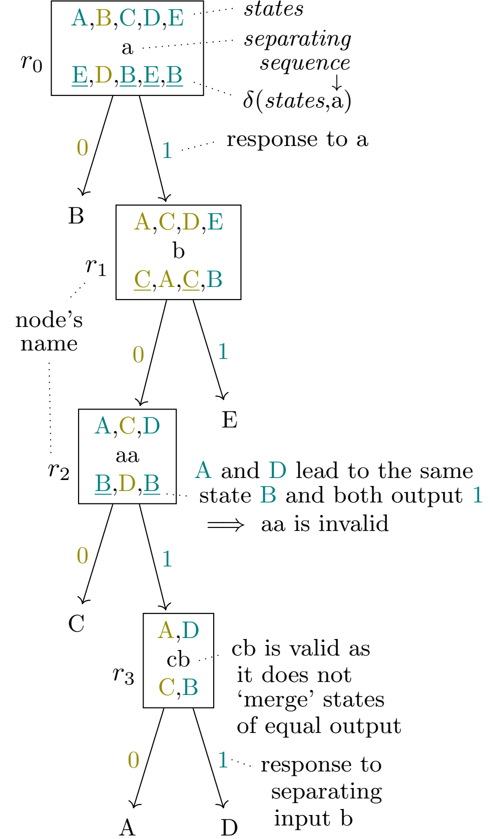


Figure 2: Splitting tree for the Mealy machine Figure 1

5.2. Splitting tree

The splitting tree of the Mealy machine (Figure 1) is shown in Figure 2 — it is the tree from which Figure 3 is constructed. A set of states called a *label* is associated with each node; these states are shown at the top of each node. The bottom row of each node shows *next states* which are entered when a sequence of inputs in a node is taken from the states labelling that node. Values in the bottom row can be helpful in the identification of current states during construction of IADS. The colour of states in next states of nodes in Figure 2 reflects the output: for node r_0, state B produces 0 and other states produce 1.

At the beginning of state identification the initial uncertainty is $\{A, B, C, D, E\}$ shown in the label for the root node r_0 of ST. The first input a makes it possible to separate states depending on the output: output 0 singles out B but if the response is 1 we could have started from any of $\{A, C, D, E\}$ as shown by the label of node r_1. In a similar way, if we somehow knew that the initial state is any of $\{A, C, D, E\}$, then input 'b' can be used to single out state E. In this way, outputs from the FSM in response to inputs in the splitting tree make it possible to separate states.

In some cases, a single input would be sufficient to separate one or more states (as in the case above with inputs a

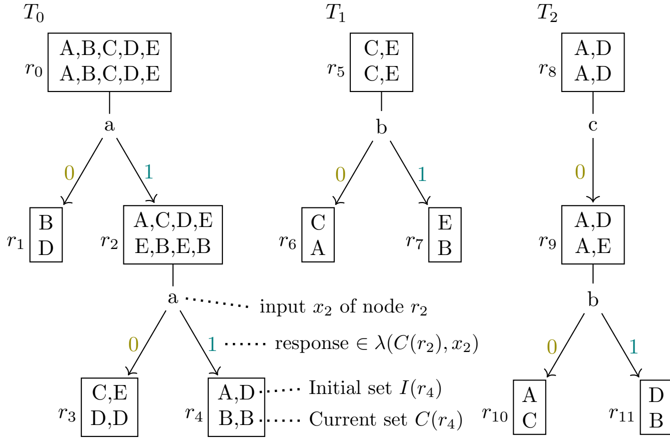


Figure 3: A fully distinguishing set of incomplete adaptive distinguishing sequences from the splitting tree in Figure 2

and b), but in other cases a single input symbol would produce the same output from all the states of interest. Such an input (called a *transferring* input) would still be useful if it leads to states that can be separated by subsequent inputs/outputs. For the considered example, this can be seen with node r_3 where input c from states A and D leads to A and E, respectively (with output 0). States A and E can then be separated with input b, therefore sequence cb can separate states A and D. This corresponds to the entire tree T_2 in Figure 3. Therefore a node in a splitting tree could have a sequence of inputs in it but transitions between nodes in ST only include the last output - it is the one that actually separates states.

Every node r in an ST answers a question 'if we started with a set of states labelling this node, how best to separate them?'. Based on the last output to a sequence in this node, there will be two or more child nodes, labelled with partitions of states from the label of r. This gives a key property of ST: a label of each node is a union of the disjoint sets of labels of its children.

When one intends to separate a set of states in an FSM, it is reasonable to expect that a larger set of states is more difficult to separate than a smaller set. For example, in an FSM producing outputs 0 and 1 a few pairs of states can be separated using a single input but separating a group of three or more states would require a longer sequence. This is why when tasked with separating a set of states S' one would pick a node in an ST with the smallest set of states containing set S'. Any node above this node will contain more states and thus solve a more difficult separating problem. If all states in an FSM can be separated, leaf nodes of a splitting tree are associated with singleton sets of states; nodes above them are labelled with a union of sets for their children etc. In this structure, anyone intending to separate states S' would be best picking a lowest common ancestor (*lca*) node of the leaf nodes corresponding to states in S'. In the described example, node r_0 separates B from $\{A, C, D, E\}$; the states entered by FSM that produced output 1 in response to input a are $\{B, E\}$. Therefore, in order to continue separating states one has to find the lowest node in ST containing this set, as it happens it is r_0 again. This time input a separates B from E. The described example shows how starting from $\{A, B, C, D, E\}$ one can construct a tree T_0 in Figure 3 resembling an ADS and splitting this set into $\{B\}$, $\{C, E\}$ and $\{A, D\}$. Since next states for different initial states in leaves r_3 and r_4 are the same, this particular tree is not able to completely separate all states. This is not surprising since the FSM in the example does not have an ADS. The contribution of this paper is an algorithm building an ST and from it a few incomplete ADS sequences (IADS) that collectively distinguish all states.

A different example is where the FSM is expected to be in one of states A, C or E and one wants to identify the state. The node of the ST that is an *lca* of the states A, C and E is r_1 and its separating sequence 'b' can be applied to the machine. Assuming that the response is 0, it means that we have not started from E. At this point, the machine is in state C or A if we started in A or C, respectively. The same procedure is repeated with the set of states $\{A, C\}$, that is, r_2 is used as a separating node of C and A (the sequence queried so far being 'baa'). According to the response, one can thus determine the unique state the machine was in at the beginning of the identification process and the state it entered at the end (the output of 1 to the last 'a' means we started from A; 0 corresponds to C). It is worth noting that although r_2 cannot separate A from D, this proved unimportant where the uncertainty is $\{A, C, E\}$ because it does not include both A and D alongside E. If we only needed to separate A and D, the *lca* would have been r_3.

In the process of state identification one jumps through nodes in an ST attempting sequences shown in the middle of each visited node and selecting subsequent nodes based on the next states of the visited nodes. This corresponds to a mostly linear walk through nodes in IADS that are also associated with inputs and branches between nodes are also based on outputs from an FSM. Here 'mostly' relates to a need to walk through additional IADSs if not all states have been separated: where all states need separating, T_0 may have to be followed by T_1 or T_2 depending on the leaf of T_0 that was entered.

5.3. Input Validity Types

In general, an input might separate some states but for other states of interest the FSM would produce the same output and enter the same state. This means that information about the starting state will be lost, therefore such inputs are called *invalid* inputs in contrast to *valid* inputs that either separate states or cause FSM to enter states that are all different. For example, sequence a of node r_0 is invalid because it merges states A and D by leading to state E with output 1. In Figure 2 this is depicted by showing the next state $\underline{E}$ both in the same colour (to indicate same output) and underlined (to indicate a *merge*).

When building an ADS, a sequence that led to different states from the starting states can be continued and eventually the starting states will be separated; invalid inputs have to be avoided because there is only one chance to run an ADS - if information is lost one can no longer separate some starting states. The machine used for the running example has no ADS, therefore, invalid separating sequences (containing invalid inputs) are assigned to nodes r_0, r_1 and r_2 of the splitting tree. For a splitting tree corresponding to an ADS no next states would be underlined - even if a node does not separate all states, for each output of an FSM the next states would be different.

A splitting tree that contains an invalid sequence cannot be the basis for an ADS but several incomplete adaptive distinguishing sequences (IADS) can be formed from it. As said earlier, such splitting trees (referred to as ST-IADS) and the algorithm to construct them are the contribution of this paper. The idea of IADS is to jump through the splitting tree as described above and pay attention not just to the outputs from FSM but also to the sets of next states. If an output is received that leads to a merge (set of next states has underlined states for this specific output), one can only continue separating states if there is something to separate, that is, the set of entered states is not a singleton. For example, T_0 in Figure 3 depicts how the first input ‘a’ causes a merge of A, D and C, E, depending on the starting states. After that, it is still possible continue separating states until the set of current states is a singleton (nodes r_3 and r_4 of T_0). At this point, one has to stop the current sequence and start a new one. During testing this is accomplished by resetting a system under test, re-running a test to enter a state of interest and then attempting a different IADS, starting from the next node in the splitting tree. Therefore, the algorithm constructing IADS ensures that if an ADS exists, it will be found and otherwise it attempts to reduce the number of different sequences that will need to be attempted in the assumption that reset and re-running a test sequence is time consuming. In the described splitting tree, an output of 1 to the second input of a has to be followed by such a reset but it also means that the initial state was not a B, C or E, leaving $\{A, D\}$ as the current uncertainty. Therefore, the root node r_8 of the next IADS T_2 has just $\{A, D\}$ as the set of initial states and sequence cb is used first since the *lca* in ST for $\{A, D\}$ is r_3.

An adaptive distinguishing sequence cannot be built from an arbitrary splitting tree. All sequences of the ST need to be composed of valid inputs. An input x is *valid* for a subset of states S' if every two distinct states s_i and s_j of S' either respond differently to x, $\lambda(s_i, x) \neq \lambda(s_j, x)$, or they lead to different states on x, $\delta(s_i, x) \neq \delta(s_j, x)$. Therefore, an input x is *invalid* for S' if there are two different states in S' such that both respond equally to x and both lead to the same state. At every step of an algorithm a partition π of states of FSM is refined; π is such that for $S_i \in \pi$ and $S_j \in \pi$, $i \neq j$ implies $S_i \cap S_j = \emptyset$ and $\bigcup_{S_i \in \pi} S_i = S$. There are three types of valid inputs x with respect to the partition π given by the labels of leaves of ST as proposed in [9]:

a) Two or more states of a block $S' \in \pi$ respond with different outputs to input x, that is, $|\lambda(S', x)| > 1$. For subsets of states of $U \subseteq S'$ producing the same output y, target states are different: for any $y \in Y$ and $U = \{s \in S' \mid \lambda(s, x) = y\}$, $|\delta(U, x)| = |U|$.

b) All states of a block $S' \in \pi$ respond with the same output, $|\lambda(S', x)| = 1$, but they lead to more than one block of π, that is, there are blocks $S_i \neq S_j$ such that $\delta(S', x) \cap S_i \neq \emptyset$ and $\delta(S', x) \cap S_j \neq \emptyset$. All target states are different: $|\delta(S', x)| = |S'|$.

c) All states of a block $S' \in \pi$ produce the same output, have different target states and lead to a block $S'' \in \pi$. In symbols, $|\lambda(S', x)| = 1$, $|\delta(S', x)| = |S'|$ and $\delta(S', x) \subseteq S'' \in \pi$.

Inputs can be also divided by their ability to separate some states regardless of their validity. An input x is called *separating* if two or more states of the subset of states S' respond differently, that is, $|\lambda(S', x)| > 1$. Otherwise, the input x is called *transferring*. This correspond to the notion of the shortest separating sequence of a pair of states. The shortest separating sequence is always formed of transferring inputs followed by a single separating input (example: cb of r_3 in Figure 2). Note that every valid input of type a) is separating and valid inputs of type b) and c) are transferring. An input of type c) that leads from block S' to itself is useless and will therefore not be used during the construction of an ST.

For invalid inputs we do not separate transferring inputs into types similar to b) and c), therefore invalid inputs are seen as one of two types,

i) invalid separating inputs where two or more states of S' respond with different outputs to input x, that is, $|\lambda(S', x)| > 1$. For some of the subsets of states of $U \subseteq S'$ producing the same output $y \in Y$, target states are merged: $U = \{s \in S' \mid \lambda(s, x) = y\}$ and $|\delta(U, x)| < |U|$.

ii) invalid transferring inputs where all states of a block $S' \in \pi$ respond with the same output, $|\lambda(S', x)| = 1$, but they either lead to more than one block of π on x (that is, states of $\delta(S', x)$ are not all in a single block of π) or they lead to a different block $\pi_2 \neq \pi_1$. Some target states are merged: $1 \leq |\delta(S', x)| < |S'|$.

In practical cases, invalid transferring inputs that merge all states ($\delta(S', x) = 1$) are completely useless for separation of states and will therefore not be selected by the ST construction algorithm below.

5.4. The structure of a splitting tree

More formally, a splitting tree (ST) is a successor tree such that:

- each node is labelled with a subset of states S',
- the root is labelled with the set of all states S,
- each internal node has a sequence w assigned that separates S',
- the label of every parent is the union of disjoint sets of states labelling its children, and
- an edge from node r to its successor r_s is labelled with the last element of the response of states of r_s to the sequence w assigned to r.

An ST is *complete* if all states are separated, that is, if there are exactly n leaves and each corresponds to one state. Note that the labels of leaves of an ST always form a partition of all states. To improve the efficiency of the implementation [9], each internal node is also associated with the set of next states $\delta^*(S', w)$ that is computed by taking a δ^* from the sequence of inputs in the node and the states of its label. Without such caching, every time one considers a node during the construction of a splitting tree, a computation of $\delta^*(S', w)$ would require a walk of up to n steps through elements of w, making time complexity worse by a factor of n.

6. State Identification Sequences from ST-IADS

A splitting tree that is complete can be a basis for the construction of separating sequences of all state pairs, characterizing sets or harmonized state identifiers as described in [8]. This section proposes an alternative design approach that does not just collect particular sequences of ST nodes as the ST-MSS algorithm does. The new approach simply follows the chosen separating sequence and extends it by another one that separates the reached states. This is repeated until no states can be separated. By doing this, new longer separating sequences are built and subsequently the number of needed sequences is reduced. This is significant for testing because HSIs are often applied at the end of every test sequence, so for instance halving their number also halves the amount of testing.

6.1. Separating Sequences from ST-IADS

Assume a system under test has entered a state s that is known to be in a subset of states S'. Algorithm 1 describes how to obtain a preset separating sequence from a given ST that separates a given state s from states in set S' (where $s \in S'$). There are much fewer FSMs that possess such sequences compared to those that have an ADS and the complexity of the construction of a preset distinguishing sequence is PSPACE [9]. For an arbitrary ST, Algorithm 1 will only separate s from some states of S'. In order to separate s from the rest of the states, one would have to re-run the algorithm on the subset of states of S' that were not separated from s. This is described later in Section 6.2.

Algorithm 1: GETSEPSEQFROMST($s \in S', S' \subseteq S$, *ST*)

```
1 w ← ε
2 while |S'| > 1 do        // there is s_i ∈ S' not
    separated from s
3   r ← getSeparatingNode(S')
4   v ← r.separatingSequence
5   w ← w · v
6   S' ← {δ*(s_i, v) | s_i ∈ S' ∧ λ*(s_i, v) = λ*(s, v)}
7   s ← δ*(s, v)
8 return w
```

Algorithm 2: GETSEPARATINGNODE(S')

```
1 select a pivot s_k from S'
2 return the node of ST.separatingNodes[(s_k, s_i)],
    s_k ≠ s_i ∈ S', with the most states
```

When separating a large set of states, a longer sequence may be needed compared to separating smaller sets. For this reason, when separating a set of states S', a good starting point is not to start with a root node of ST labelled with all the states S of the FSM but instead find a lower-level node that contains S' and as few other states as possible. As mentioned above, this corresponds to identification of a lowest common ancestor (*lca*) of the leaves in ST corresponding to states in S'.

Construction of an *lca* is done by calling function GETSEPARATINGNODE described in Algorithm 2. A state pair array *separatingNodes* that is part of ST in the implementation of this algorthm permits efficient search for *lca*. For every pair of states, *separatingNodes* stores the lowest node in ST that separates such a pair. For example, *separatingNodes*[(A,E))] $= r_1$ and *separatingNodes*[(C,D)] $= r_2$ for the ST in Figure 2. Consider a path from the root node of ST to the leaf labelled with state s. The root is labelled with all the states S, a child of the root node on this path will have a strict subset of S, the subsequent node on the path will have an even smaller set states and so long until only s is left at the leaf. Of all the nodes in ST, only nodes on this path contain s in their label which means they are the only candidates for a node that is *lca*. The length of labels of nodes on this path varies from n at the root to 1 at the leaf. Since there are at most n nodes on such a path, this gives a bound of n on the length of search for *lca* but the use of *separatingNodes* makes it possible to do it in at most $|S'| - 1$ steps.

Proposition 1. *Given a subset of states $S' \subseteq S$ and a splitting tree* ST*, Algorithm 2 finds the lowest common ancestor of the states of S' by comparison of at most* $(|S'|-1)$ *nodes of* ST*.*

Proof. Pick any $s_k \in S'$ and consider the set of nodes $L = \{separatingNodes[(s_k, s_i)] \mid s_i \in S'$ such that $s_k \neq s_i\}$.

There are at most $(|S'|-1)$ distinct nodes and they are all on a path in ST from the root node to a leaf labelled with s_k. A label of a node in ST is a union of disjoint sets of states labelling its children therefore nodes in L can be arranged in a sequence of their labels $l_1 \subseteq l_2 \subseteq \ldots \subseteq l_a$ where l_i is a label of some $r_i = separatingNodes[(s_k, s_i)]$ for some $s_i \in S'$. The use of $\subseteq$ reflects that some of these r_i could be the same. The topmost (in ST) node $r_a = separatingNodes[(s_k, s_a)]$ for some $s_a \in S'$ is therefore the node with the most states and contains states of all other nodes in L. We prove that r_a is the *lca* for S'.

Consider a node r_l that is not on a path in ST from root to the leaf for s_k. In this case, there is a node r_p that is an *lca* for r_l and node for s_k. Children of node r_p are labelled with a disjoint set of states, therefore, the label of r_l does not contain s_k which implies that r_l cannot be an *lca* for S'.

Consider a child r_c of node r_a. If its label contains both s_k and s_a then r_a cannot be the lowest node in ST separating these two states which contradicts the construction of *separatingNodes*. Therefore, r_a is an *lca* of S'. □

Algorithm 1 starts with an empty separation sequence w and an initial state s; it then appends a separating sequence v of the *lca* of S' to it. Such a sequence is expected to separate a few states based on the output but the rest of the states in S' would transfer to some other states. The algorithm therefore determines the next state $\delta^*(s, v)$ and $\delta^*(s_i, v)$ for states $s_i \in S'$ that produce the same output as s in response to v (the rest of the states in S' are separated by the output). For a subsequent iteration of the algorithm, state s is replaced with $\delta^*(s, v)$ and S' with $\delta^*(s_i, v)$ for the described s_i. The algorithm stops when the set of current states becomes a singleton - either all states have been separated or (most likely) they have been merged.

The time complexity of Algorithm 1 depends on the number n_d of states in the set S' that are separated from the given s; $n_d < |S'|$. If all different states of S' are distinguished from s, $n_d = |S'| - 1$. The algorithm separates at least one state of S' per iteration of the main loop (lines 2–7) so it does at most n_d iterations through the loop. It compares at most $|S'|$ separating nodes to find the lowest common ancestor r using getSeparatingNode. Therefore, it runs in $O(n_d * |S'|) = O(|S'|^2)$. Considering that the next states are cached in r, updating S' does not increase time complexity.

6.2. HSI from ST-IADS

It is easy to construct harmonized state identifiers with Algorithm 1 at hand. The HSI of a state s_k is formed by successive calls of getSepSeqFromST(s_k, S', ST) for a subset S' of states that are not separated from s_k by a sequence which was already added to the HSI. Algorithm 3 builds an HSI for all states in this fashion. If the constructed HSI of a state contains just one sequence, then its construction takes $O(n^2)$ which follows from the time complexity of Algorithm 1. The complexity does not increase even if the HSI contains several sequences because the sum of the numbers n_d of distinguished states for each call of getSepSeqFromST is equal to n. Therefore, Algorithm 3 constructs HSIs of all states from the ST in $O(n^3)$.

Algorithm 3: getHSIsFromST(ST)

```
1 foreach s_k ∈ S do
2     HSI_k ← ∅
3     S' ← S
4     while |S'| > 1 do      // there is s_i ∈ S' not separated from s_k
5         w ← getSepSeqFromST(s_k, S', ST)
6         add w to HSI_k
7         S' ← {s_i ∈ S' | λ*(s_i, w) = λ*(s_k, w)}
8 return HSI as a collection of HSI_k of all states
```

6.3. IADSs from ST-IADS

The construction method of an adaptive distinguishing sequence (ADS) from a complete ST was proposed in [9] and Algorithm 4 extends it to work even if the ST contains invalid separating sequences. A set of incomplete adaptive distinguishing sequences (IADS) is thus returned in general instead of a single ADS but the algorithm returns an ADS if the complete ST has no invalid sequences. The idea is the same as in getSepSeqFromST but the separating sequences are stored in nodes of the tree representing IADS instead of appending them one after another, and all responses (different branches) are handled as there is no reference state s. An IADS is represented by a successor tree such that:

- each node r_j is labelled with the initial set $I(r_j)$, the current set $C(r_j)$ and an input x_j,
- all edges leading from an internal node r_j are labelled with distinct output symbols produced by states of $C(r_j)$ in response to x_j, and
- if $\lambda^*(s, u)$ labels the path from the root r_0 to a node r_j where $s \in I(r_0)$ and u is the input sequence formed of x_i's on the path (without x_j of r_j), then the state s is in $I(r_j)$ and $\delta^*(s, u)$ is in $C(r_j)$.

This definition implies that for every root r_0 holds $I(r_0) = C(r_0)$. The original definition of ADS in [9] requires $I(r_0) = S$ as it needs to distinguish all states and has exactly n leaves. In the case of IADS, this requirement is transferred to the following property of a set of IADSs. A set D of IADSs is *fully distinguishing* if every pair of distinct states is distinguished by some IADS from D [3], in symbols for any pair of states $s_1 \in S$ and $s_2 \in S$ there is an IADS $T_i \in D$ containing nodes $r_1 \neq r_2$ such that $s_1 \in I(r_1)$ and $s_2 \in I(r_2)$.

Algorithm 4: getIADSsFromST(ST)

```
1  push S into undistinguished
2  foreach S' ∈ undistinguished do
3      create a new IADS T_i with the root r such that
         I(r) = C(r) = S'
4      push r into unprocessedNodes
5      foreach r_j ∈ unprocessedNodes do
6          r^ST ← getSeparatingNode(C(r_j))
7          u · x_j ← r^ST.separatingSequence
8          foreach x_k ∈ u in their order do
9              r_j.input ← x_k,    y ← λ(C(r_j), x_k)
10             create a successor r_k with the edge from
                 r_j labelled with y
11             I(r_k) ← I(r_j),    C(r_k) ← δ(C(r_j), x_k)
12             r_j ← r_k
13         r_j.input ← x_j
14         foreach y ∈ λ(C(r_j), x_j) do
15             create a successor r_k with the edge from
                 r_j labelled with y
16             C(r_k) ← {δ(s_j, x_j) | s_j ∈
                 C(r_j) ∧ λ(s_j, x_j) = y}
17             I(r_k) ← {s_j ∈ I(r_j) | δ*(s_j, d_k) ∈ C(r_k)
                 where d_k is the sequence of x_i's along
                 the path from the root of T_i to r_k}
18             if |C(r_k)| > 1 then
19                 push r_k into unprocessedNodes
20             else if |I(r_k)| > 1 then
21                 push I(r_k) into undistinguished
22 return IADSs T_i's
```

Algorithm 4 uses two queues to build a fully distinguishing set of IADSs. The first queue called *undistinguished* includes subsets of states that are not distinguished from each other by any existing IADS; it initially contains the set of all states. The second queue called *unprocessedNodes* is for the leaves r_j of the current IADS T_i such that their current sets $C(r_j)$ contain several states which means that they can be separated. Each root of a new IADS T_i is initialized with a subset of states from *undistinguished* and pushed into *unprocessedNodes* to start the construction of T_i. For each unprocessed node r_j of T_i, the lowest common ancestor is found by getSeparatingNode and its separating sequence is then divided into the transferring sequence u and the separating input x_j. If u is not empty, a chain of successors representing u is appended to r_j and r_j then points to the new leaf. All these successors have the same initial state and differ in the current sets that are updated by each input of u. The separating input x_j is assigned to the leaf r_j and divides its initial and current states according to the responses. Each successor r_k corresponds to the states of $C(r_j)$ that produce the same output to x_j. Their initial and current sets are updated accordingly (lines 16 and 17 of Algorithm 4). The initial and current sets are implemented as arrays so the correspondence between initial and current states is easily accessible, and the node r^{ST} with its successors in the splitting tree provides enough information to form successors of r_j and their current sets. Finally, if the successor r_k can be further separated, then it is added to *unprocessedNodes*. If r_k cannot be separated but some initial states leading to this node were not distinguished, then these states ($I(r_k)$) are pushed into *undistinguished* and another IADS will distinguish them. This corresponds to the case where invalid sequences merge some states therefore multiple IADSs have to be constructed to separate all states. A fully distinguishing set of IADSs constructed by Algorithm 4 from the ST in Figure 2 is shown in Figure 3. Note that usually there is no need to store the chain of successors representing the transferring inputs. Therefore, a shortened version of IADSs can be introduced such that a node stores a separating sequence instead of a single input as proposed in [7]. Lines 8–12 of Algorithm 4 would be omitted and lines 13–17 would work with the entire separating sequence $u \cdot x_j$ instead of just with x_j.

At most $n-1$ separating sequences are needed to distinguish all states. A suitable separating sequence is found in the ST by getSeparatingNode in $O(n)$. Therefore, Algorithm 4 runs in $O(n^2)$ if the shortened version of IADSs is built. Otherwise, it also depends on the length of separating sequences in the ST because the chain of successors corresponding to each sequence needs be created; the time complexity is $O(n^3)$ if all separating sequences have length at most n.

All three algorithms work with any splitting tree, however, getIADSsFromST can build an ADS only from the ST that has only valid separating sequences. The ST-MSS algorithm [8] thus cannot be used to prove the existence of an ADS and generally it results in more sequences than from ST-IADS. A characterizing set can be formed from the ST as well as the collection of all separating sequences [8]. Characterizing sets formed of either sequences of IADSs constructed by Algorithm 4 or as union of HSIs constructed by Algorithm 3 are the same if they are based on the same ST.

7. ST-IADS Construction Algorithm

This section proposes the extension to the existing algorithm [9] such that the existing algorithm building a splitting tree with valid separating sequences is not disrupted by the extension. It means that the extension is not employed if the given machine has an adaptive distinguishing sequence (ADS) and so a splitting tree with only valid sequences is constructed. As the extension allows one to construct incomplete adaptive distinguishing sequences (IADS), it is referenced as the ST-IADS algorithm and its part corresponding to the existing algorithm from [9] is referenced by the ST-ADS algorithm.

The ST-IADS algorithm is described in Algorithms 5–11 such that Algorithm 5 captures the main part with

other algorithms describing routines it relies on.

7.1. General Idea

Methods building separating sequences are usually described in terms of a *partition* of states which initially consists of a single block with all states in it and every step of an algorithm splits these blocks [9]. Figure 4 shows a hypothetical construction of a splitting tree for a machine with 7 states A–G. The initial partition includes only block r_0. Figure 4(A) shows that input 'a' separates some states of the root of the splitting tree, therefore at the subsequent step the partition contains three blocks of states r_1, r_2 and r_3. In this example, state G is separated but two other blocks need to be dealt with in the steps to follow. Figure 4(B) shows how input 'b' can be used to separate states in those blocks: A, C from B and E, F from D; the next step would separate A from C and E from F. Each of these blocks of states corresponds to a node in ST. An ST is constructed incrementally and after each step of an algorithm the partition of FSM states corresponds to labels of the leaf nodes of ST. During construction not every leaf node is labelled with a singleton set but if all states are separated, all leaves have singleton labels.

Inputs of type a) that separate states based on the output are the best since they immediately provide information permitting one to separate states. If a node cannot be split in this way, another option is using an input of type b) which relies on an input that leads FSM from current states for the node to states that we know how to separate. For the running example separation of states A and D required an input c of type b) followed by input b of type a). This 'know how to separate' is based on the information in ST — input of type b) requires finding an already existing node in ST such that next states are included in the label of that node and children of that node splitting those states. In other words, we want to find a node r such that for its children r_1, r_2 the corresponding labels $l(r_1) \cap C \neq \emptyset$ and $l(r_2) \cap C \neq \emptyset$ where C is the set of next states for the considered node. If such a r (which has to be an *lca* of states in C) can be found, its children are splitting the current node and therefore can be (recursively) cloned as children of the considered node with few changes. Since this is done in order to split a specific node, the sets of states labelling those clones have to be reduced to be subsets of the label of the considered node and new sets of next states have to be computed for the clones.

Inputs of type c) switch between elements of a current partition without separating states, if these are the only valid inputs, one might have to make a number of such 'hops' until an input of type a) or type b) becomes available.

Compared to the ST-MSS algorithm [8] that processes the leaves containing several states in the order of the length of their expected separating sequence, the ST-ADS orders the leaves for processing according to the number of states in their labels. This helps with the search for a valid separating sequence for a subset of states and also guarantees that if an ADS exists it will be of a polynomial length [9]. Splitting nodes in the order of their size is close to a breadth-first construction of ST where we start with the largest node and in the subsequent step all the largest nodes of the same size are split, next step smaller nodes are split etc. In contrast, depth-first exploration may cause a long branch to be constructed and at the end of it an input of type b) may cause the whole ST to be cloned as described above for dealing with inputs of type b); while probably unlikely in practice, in the worst case this leads to an exponentially long tree which is why splitting in the order of size was originally introduced by [9].

In the described work, inputs of type a) are applied immediately in each step and where they are not available, nodes (which would be leaf nodes at that moment) are stored in a collection of leaves that need a separating sequence of several inputs. These leaves are then processed in similar manner to the SSS algorithm [11], that is, links between them are first found and subsequently if one is separated, it can be used to separate other nodes that have a link to it. This corresponds to a search for transferring inputs. In other words, when a subset r_j of states is separated with a sequence w, another subset r_i can be separated with the sequence $x \cdot w$ where x labels the link from r_i to r_j, that is, states of r_i transfer to states of r_j on x.

In the case of the ST-ADS algorithm, the links, or transitions between subsets of states, are restricted to valid inputs. Therefore, some leaves do not have to be distinguished if they do not have a valid separating sequence. The extension is thus employed such that even invalid inputs are considered for the links from leaves that have not been separated. The extension described in this work makes an effort to find the best invalid separating sequence using scoring to choose between possible inputs (the lower the score the better the input hence the score is effectively a penalty). If there is no valid separating sequence, the method explores all the shortest invalid separating sequences. During the exploration, auxiliary nodes representing the subsets of states reached by these sequences are created; they are kept after they are analysed as they may be used at any subsequent step of the algorithm.

7.2. Data structures of the ST-IADS algorithm

There are several structures that the algorithm handles.

- Every node r of the splitting tree ST contains a set of *states*, a *separatingSequence* and the associated next states $\delta^*(r.states, r.separatingSequence)$ - this is the cache of the next states to improve performance.
- *separatingNodes* is a state pair array that is a part of ST and stores for each state pair (s_i, s_j) a node r of ST that separates s_i and s_j, that is, both s_i, s_j are in *r.states* (the label of r) but they are in different *states* of the children of r.

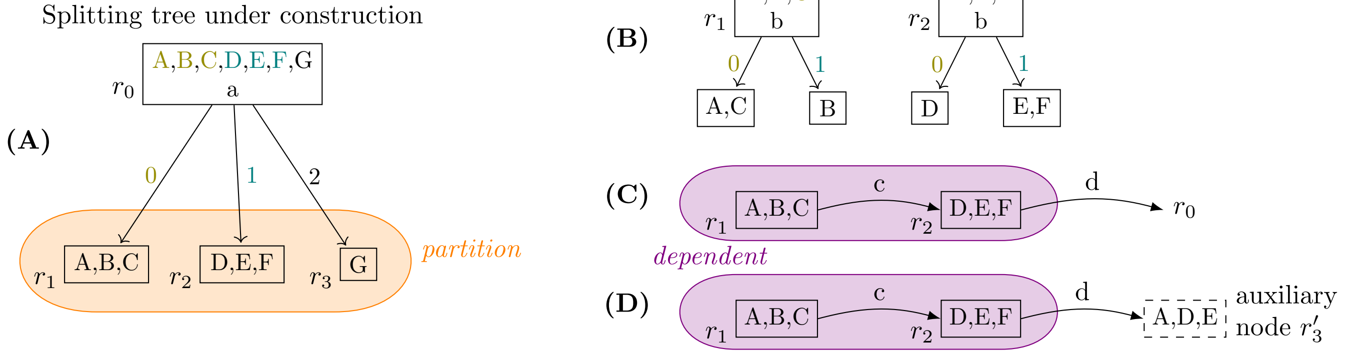


Figure 4: Splitting a node during the construction of a splitting tree

Algorithm 5: Construction of a splitting tree

input : A minimal DFSM M with n states
input : *validOnly* allows to use only valid inputs if true
output: A splitting tree ST of M or null if *validOnly* and M has no ADS

1. $r.states \leftarrow$ all n states of M `// r is initially the root of ST`
2. $ST.separatingNodes[(s_i, s_j)] \leftarrow$ null for all $(s_i, s_j)_{,i<j}$
3. **if** M has state outputs **then**
4. $\quad r.separatingSequence \leftarrow \uparrow$
5. $\quad$ sepByCreatingSuccessors(r)
6. **else** $partition \leftarrow \{r\}$ `// M is a Mealy machine`
7. **while** $|partition| \neq n$ **do**
8. $\quad r \leftarrow$ pop a node from *partition* with the most states
9. $\quad$ **if** r is not analysed **then**
10. $\quad\quad$ analyse(r)
11. $\quad$ **if** *validOnly* **and** r has no valid input **then**
12. $\quad\quad$ **return** null `// no ADS`
13. $\quad$ **if** $r.separatingSequence$ is assigned **then**
14. $\quad\quad$ sepByCreatingSuccessors(r)
15. $\quad$ **else** push r into *dependent*
16. $\quad$ **if** $|dependent| > 0$ **and** $\forall p \in partition$: $|p.states| < |r.states|$ **then**
17. $\quad\quad$ **foreach** $u \in dependent$ **do**
18. $\quad\quad\quad$ initTransOnValidInputs(u)
19. $\quad\quad$ processDependent()
20. $\quad\quad$ **if** $|dependent| > 0$ **then**
21. $\quad\quad\quad$ **if** not *validOnly* **then**
22. $\quad\quad\quad\quad$ **foreach** $r \in dependent$ **do**
23. $\quad\quad\quad\quad\quad$ initTransOnValidInputs(r)
24. $\quad\quad\quad\quad\quad$ **if** r has no valid separating sequence **then**
25. $\quad\quad\quad\quad\quad\quad$ initTransOnInvalidInputs(r)
26. $\quad\quad\quad\quad$ processDependent()
27. $\quad\quad\quad$ **else return** null `// no ADS`
28. **return** ST

- *partition* is a set of the leaves of ST; it is implemented as a priority queue such that the leaves with the most states in *states* can be fetched first. This is needed to enforce the splitting of nodes in the order from the biggest to the smallest.
- *dependent* is an array of nodes with *states* that do not have a separating input and have to be split using transferring or invalid ones.
- *dependentPriorityQueue* is a priority queue of nodes from *dependent*. At each step, nodes that are split are added to it. After all separating inputs and one-step transferring inputs have been attempted, nodes from *dependentPriorityQueue* are used in order to split nodes still in *dependent* using transferring inputs.
- *transitionsTo* is an array of lists filled with the links between nodes in *dependent*.
- For each node r in *dependent*, $best_r$ stores an *input*, a node *next* reached by the *input* and a score reflecting how good a separating sequence can be if it starts with this input. The $best_r.score$ is used when r is pushed into *dependentPriorityQueue* that favours nodes with the lowest score so that they are fetched first.

Global structures *ST.separatingNodes*, *partition*, *dependent*, *dependentPriorityQueue*, and *transitionsTo* are available in all functions of the ST-IADS (Algorithms 5–11). Besides a minimal DFSM M, the ST-IADS algorithm takes an input parameter *validOnly* that when true forces the algorithm to follow the existing ST-ADS algorithm. If M has no ADS and *validOnly* is true, the algorithm will return the *null* value representing the absence of an ADS.

7.3. ST-IADS algorithm

Algorithm 5 starts by separating all states in the root of ST using the stOut input $\uparrow$ if the given machine M produces state outputs. The function sepByCreatingSuccessors in Algorithm 6 captures how a leaf r is separated. It first appends new nodes to r and then updates

partition and $ST.separatingNodes$ if r is a part of ST. New successors of r are formed from states of r that respond to the separating sequence of r in the same way; the last output symbol of their response, that is, the response to the separating input, labels the edges from r to the successors.

The condition on line 2 of Algorithm 6 ensures that only successors of r present in ST lead to updates to *partition* and *ST.separatingNodes*. This is necessary because during a search for the best invalid separating sequence a range of auxiliary nodes of different size is created in *dependent*; out of these nodes only a subset will be used to split nodes in ST but to identify the splitting power of each auxiliary node one needs to be able to construct links between them.

Algorithm 6: SEPBYCREATINGSUCCESSORS(r)

```
1 create successors of r by grouping states of r.states
    with the same response to r.separatingSequence
2 if r is in ST then
3     add the successors to partition
4     ST.separatingNodes[(s_i, s_j)] ← r  ∀s_i, s_j in
      different successors
```

The main loop of Algorithm 5 (lines 7–27) starts after *partition* is initialized with the current leaves of ST. In each iteration through the loop, one leaf r with the most states is separated if it has a separating input of type a). Otherwise, it is stored in *dependent*. An exception is reported if r has no valid input and valid inputs are required by *validOnly*. In this case the ST-IADS algorithm returns null as a sign that there is no ADS for the given machine. To discover if r has a valid separating input, r is first analysed. The function ANALYSE(r) in Algorithm 7 checks each input x and stores it as the separating sequence of r if it is a valid separating input. Otherwise, all x and the related next states $\delta(r.states, x)$ are cached if x is valid or invalid inputs are allowed. Invalid transferring inputs that merge all states into one, that is, $|\delta(r.states, x)| = 1$, are not stored as they cannot begin a separating sequence. During the search for invalid separating sequences, a range of nodes is created and analysed, this is why it is possible that the leaf r has already been analysed as an auxiliary node so the analysis is not repeated due to the condition on line 9 of Algorithm 5.

After all leaves of ST with the same number of states are analysed and checked, those pushed into *dependent* are separated by sequences of several inputs (lines 17–27 of Algorithm 5). The condition on line 16 of Algorithm 5 is only true when all nodes of the same maximal size were processed (it literally means that we are looking at the last node of the maximal size) and so it is the time to split nodes that could not be split with inputs of type a) using transferring inputs before moving to the next step of the algorithm and splitting smaller nodes.

Assume that in the example in Figure 4 there is no

Algorithm 7: ANALYSE(r)

```
1 foreach input x ∈ X do
2     if x is a valid separating input then
3         r.separatingSequence ← x
4         return
5     else if x is a valid transferring input or
      (not validOnly and
      (|λ(r.states, x)| > 1 ∨ |δ(r.states, x)| > 1)) then
6         store x and next states δ(r.states, x)
```

separating input for a node r_1 then this node is pushed in the set *dependent*. The function INITTRANSONVALIDINPUTS (line 18 of Algorithm 5) then creates links between nodes according to the transition function. Assume that the next states of A, B and C on the input 'c' are states D, E and F, respectively, and D, E, F lead to A, D, E on 'd'. Therefore, the case shown in Figure 4(C) shows a link 'c' from r_1 to r_2. The link 'd' from r_2 points to r_0 because it is an *lca* for states A, D, E. For this reason, function PROCESSDEPENDENT forms the separating sequence 'da' for r_2 and then 'cda' for r_1 and splits the nodes accordingly.

At first, only valid separating sequences are used such that links on valid inputs are prepared by INITTRANSONVALIDINPUTS (Algorithm 8) for every node in *dependent* and then they are gradually processed by PROCESSDEPENDENT (Algorithm 9). In both cases, only transferring inputs are considered. After each complete step of the algorithm the set *dependent* will be empty - if some nodes are not separated, the FSM does not have an ADS. For this reason, either the process is repeated using invalid inputs or the algorithm exits with null if only valid inputs are allowed.

Proposition 2. *If an FSM has an ADS, the splitting tree constructed by Algorithms 5 is valid and will lead to a valid ADS.*

Proof. Sketch of the proof. The initial node contains all states of an FSM. By construction of the algorithm, nodes are processed in steps where each step splits all nodes labelled with the largest set of states (this could be a single node or multiple nodes of the same size). Each node is split by identifying a valid input of type a) to split it if there is any input of this type and otherwise it is stored in *dependent*. Once all nodes in a step that can be split with inputs of type a) are dealt with, nodes that can be separated by transferring inputs are split by INITTRANSONVALIDINPUTS. The remaining nodes are split with transferring inputs by PROCESSDEPENDENT. The stepwise process and handling of these types of inputs corresponds to cases 1)-3) of Algorithm 3.2 in [9]. As such, the proof of validity and the length of ADS from [9] can be directly applied to ST by Algorithm 5.

Construction of ADS from ST in Algorithm 4 follows the construction in Algorithm 3.3 of [9]. □

Function INITTRANSONVALIDINPUTS on line 23 considers transferring inputs leading to a node with an invalid separating sequence. If in Figure 4(C) the sequence assigned to r_0 was invalid, then an auxiliary node r'_3 with states A, D, E (a strict subset of states of r_0) would be created as shown in Figure 4(D) because a smaller set of states could have a valid separating sequence and in any case a smaller set of states is likely to have a shorter separating sequence. In this case, separating sequences for nodes r_2 and r_1 would be constructed in a similar way to Figure 4(C) using a separating sequence for the auxiliary node r'_3.

Alternatively, it can happen that there is no valid input for r'_3 and thus no valid sequences for r_1 and r_2. Both nodes then remain in *dependent* and the function INITTRANSONINVALIDINPUTS creates links between nodes even for invalid inputs. In this case, the algorithm attempts to find the biggest subset of states that can be distinguished by an invalid input and uses scoring (details in Section 7.5) for that.

Although in the algorithm function INITTRANSONVALIDINPUTS on line 23 is called for every node of *dependent*, the implementation only calls it for new auxiliary nodes that are added to *dependent* by INITTRANSONVALIDINPUTS or by INITTRANSONINVALIDINPUTS (Algorithm 10).

After the set *dependent* has been populated with nodes linked with potentially invalid transitions, the next call of PROCESSDEPENDENT is again looking at transferring inputs to splits nodes. In a similiar case to valid inputs, PROCESSDEPENDENT picks separated nodes one by one and considers nodes with transitions to separated nodes, separating them and subsequently adding them to *dependentPriorityQueue* so that nodes with transitions to newly separated nodes are processed. The loop on line 22 ensures that each node r in *dependent* is looked at by either INITTRANSONVALIDINPUTS or INITTRANSONINVALIDINPUTS and possible transferring inputs added to newly-created nodes r_1 in *dependent*. For each such r, these new nodes will have fewer states than r and again due to the loop on line 22 they will be considered by INITTRANSONVALIDINPUTS or INITTRANSONINVALIDINPUTS. Thus for a starting node r, the sequence of nodes $r \to r_1 \to r_2 \to \ldots$ has a monotonically reducing number of states hence eventually there will be an r_k that can be separated by a valid input and it will be added to *dependentPriorityQueue*. When node r_k is considered by PROCESSDEPENDENT, the nodes $r_{k-1}, r_{k-2}, \ldots, r_1, r$ will be considered in turn. This shows that the set *dependent* will also become empty at the end of each step. The algorithm will also keep a record of all those auxiliary nodes $r_1, r_2 \ldots$ that were created in the hope that these nodes would be useful later to separate states. Such nodes will not become part of ST unless they are best (according to the scoring routine) at separating nodes in ST.

Proposition 3. *If an FSM has all states pairwise separable,* GETIADSSFROMST *algorithm constructs valid IADSs from the splitting tree built by Algorithm 5.*

Proof. Sketch of the proof.

For each node of ST, Algorithm 6 constructs child nodes that partition states labelling this node. In this way, every set of states can be partitioned.

The construction of IADS is recursive in that starting with the root state r_0 of ST and the associated input sequence w_{r_0}, one needs to attempt w_{r_0} and branch on outputs y, leading FSM to the target states $N(V_0, r_0, y)$ where

$$\begin{aligned} V_0 &= \{(s, s) \mid s \in S\}, \\ C(V, r, y) &= \{s_c \mid (s_i, s_c) \in V \\ &\qquad \wedge w_r = w' \cdot x \wedge \lambda(\delta^*(s_c, w'), x) = y\}, \\ N(V, r, y) &= \delta^*(C(V, r, y), w_r). \end{aligned}$$

These target states are subsequently separated using a sequence in the node $r_1 = lca(N(V_0, r_0, y))$ where by abuse on notation we alias states and singleton ST leaf nodes labelled with those states. Such an r_1 could be r_0 or any other node in ST so in order to define the recursive process, one has to talk of both a node r in ST under consideration and a set of initial and current states of IADS node being considered. At the start, the node is root r_0 and $V_0 = \{(s, s) \mid s \in S\}$ includes pairs of initial and current states. The starting pair is (r_0, V_0). After that, the node is $r_1 = lca(N(V_0, r_0, y))$ and $V_1 = U(V_0, r_0, y)$ depending on y where

$$\begin{aligned} U(V, r, y) = \{(s_i, \delta^*(s_c, w_r)) \mid (s_i, s_c) \in V \\ \wedge w_r = w' \cdot x \wedge \lambda(\delta^*(s_c, w'), x) = y\}. \end{aligned}$$

We have thus a set of such (r_1, V_1). For a given pair (r, V), GETIADSSFROMST can be thought to construct a pair of $(ads_0, iads)$ where the first element is an IADS starting with a node labelled with the initial and current states from V and with the separating sequence of ST node r, the second element *iads* is a set of IADSs that account for the use of invalid input sequences by ads_0.

The difference of the considered ST to the one where only valid inputs are used is that for some nodes r and sets V, there is such a y that $|N(V, r, y)| < |C(V, r, y)|$.

We prove that for a pair (r_0, V_0) GETIADSSFROMST constructs a set $(ads_0, iads)$ separating all initial states in V_0.

Main part of the proof: for a pair (r, V) a recursive process separating all initial states in V starting from node r has three cases to consider:

1. a leaf node r means that the initial states in V cannot be further separated. Hence, a single IADS node labelled with initial and current states of V is constructed together with an empty set of *iads*. Where the node r is an *lca* after an invalid input was used (case 3), there may be several initial states in V and only one current state.

2. non-leaf nodes with a valid separation sequence. Let children of r in ST be $r_1 \dots r_{|Yr|}$ for different outputs $y \in Yr$ where $Yr \subseteq Y$ is the set of outputs from FSM to the last element of w_r. In this case, for every output y and $|C(V, r, y)| > 0$, the corresponding pairs $(lca(N(V, r, y)), U(V, r, y))$ can be constructed. Let us assume that GETIADSsFROMST will construct pairs $(a_1, ia_1) \dots (a_k, ia_k)$ for each of these pairs where k is the number of elements of Yr where $|C(V, r, y)| > 0$. The corresponding IADS for r has the structure $w_r/y_1 \to a_1 \dots w_r/y_k \to a_k$, hence a pair

$$(w_r/y_1 \to a_1 \dots w_r/y_k \to a_k, \cup_{j=1\dots k} ia_j)$$

will separate all initial states in V. As such, we have shown that GETIADSsFROMST will construct a valid pair $(ads_0, iads)$ for (r, V) where r is a node with a valid separating sequence.

3. non-leaf nodes with an invalid separating sequence. If r has an invalid sequence, then the set of the states of its label can be split into non-intersecting subsets $T(y, s_t) = \{s_c \in C(V, r, y) \mid \delta^*(s_c, w_r) = s_t\}$ where each subset corresponds to states that produce the same output and lead to the same state in the FSM; $|\delta^*(T(y, s_t), w_r)| = 1$. Let the number of such subsets be k, this number is lower than $|V|$ (if it is the same as $|V|$, r has a valid separating sequence). This number k can be greater than the number of outputs y satisfying $|C(V, r, y)| > 0$ because for the same output y there could be subsets $T(y, s_t) \neq T(y, s'_t)$ producing the same output and leading to different states $s_t \neq s'_t$. Some of these sets $T(y, s_t)$ will be singletons; if r had a valid separating sequence, all such $T(y, s_t)$ would be singletons. Considering node r as a pseudo-node r' labelled with such sets $T(y, s_t)$ rather than individual states makes it look like a node with a valid separating sequence, therefore the same construction as in the case of a node with a valid separating sequence leads to $(a_{r'}, ia_{r'})$. This pair makes it possible to separate subsets $T(y, s_t)$.

   The pair $(a_{r'}, ia_{r'})$ does not permit separation of individual states of the subsets $T(y, s_t)$. Therefore, additional pairs $z_j = (lca(I(y, s_t)), V(y, s_t))$ for $j = 1, \dots, k$ can be constructed where

   $I(y, s_t) = \{s_i \mid (s_i, s_c) \in V \land s_c \in T(y, s_t)\}$ and
   $V(y, s_t) = \{(s_i, s_i) \mid s_i \in I(y, s_t)\}$.

   Following the same process for these pairs, it is possible to construct pairs of (a_{z_j}, ia_{z_j}) for each of $z_1 \dots z_k$. A pair of

$$(a_{r'}, ia_{r'} \cup \{a_{z_1}, a_{z_2}, \dots, a_{z_k}\} \cup \bigcup_{j=1\dots k} ia_{z_j})$$

   separates all the initial states in V.

□

Algorithm 8: INITTRANSONVALIDINPUTS(r)

```
1  (best_r.input, best_r.next, best_r.score) ← (null, null, ∞)
2  foreach valid transferring input x of r do
3      r_x ← GETSEPARATINGNODE(δ(r.states, x))
4      if r_x ∈ dependent then
5          add (r, x) to transitionsTo[r_x]
6      else if separating sequences of r_x and best_r.next
         are not valid and |r.states| < |r_x.states| then
7          r_x ← a (stored or new) node with states
             equal to δ(r.states, x)
8          if r_x is not analysed or
             r_x.separatingSequence is not set then
9              if r_x is not analysed then
10                 ANALYSE(r_x)
11                 if r_x.separatingSequence is assigned
                     then
12                     best_r ← (x, r_x, SCORE(r, x, r_x))
13             push r_x into dependent if it is not there
14             add (r, x) to transitionsTo[r_x]
15         else if SCORE(r, x, r_x) < best_r.score then
16             best_r ← (x, r_x, SCORE(r, x, r_x))
17     else if SCORE(r, x, r_x) < best_r.score then
18         best_r ← (x, r_x, SCORE(r, x, r_x))
19 store best_r
20 if best_r.next has a valid separating sequence then
21     push r in association with best_r.score into
         dependentPriorityQueue
```

7.4. State splitting functions of ST-IADS algorithm

Algorithm 8 describes the function INITTRANSONVALIDINPUTS that initializes $best_r$ of the given node r and links from r on each valid transferring input x. In the case of the ST-ADS algorithm, this function chooses the best valid transferring input of type b) and stores the links on valid inputs of type c), see Section 5.3 for input validity types. A valid input x of type b) means that the next states $\delta(r.states, x)$ are covered by more than one block of the current partition. Therefore, there is a node r_x with the sequence that separates the next states. The best input out of those of type b) should lead to a node with the shortest separating sequence. This is exactly what is done on lines 17–18 of Algorithm 8 as the function SCORE (Algorithm 11) returns the length of separating sequence of r_x if it is valid. A valid input x of type c) transfers the states of r into another block of the current partition. It means that the node r_x representing such a block of states is a leaf in *dependent* because a valid input does not merge states, that is, $|r.states| = |\delta(r.states, x)|$, and the leaves are processed in the order of the size of *states*. Therefore, a link from r to r_x on x is stored in *transitionsTo* (lines 4–5) so that r can be separated based on r_x when a separating sequence is found for r_x. The node r_x that includes all next states is located in the ST as the low-

est common ancestor of the leaves corresponding to the next states; the function GETSEPARATINGNODE is defined in Algorithm 2. In the case of the ST-IADS algorithm, r_x can have an invalid separating sequence. Lines 6–16 of Algorithm 8 optimize the choice of invalid separating sequence starting with a valid transferring input x of type b). If r_x has an invalid separating sequence and more states than r, then there could be a better separating sequence for the next states $\delta(r.states, x)$. Therefore, an auxiliary node including just these states is created and analysed (if it was not). Note that once an r_x with a valid separating sequence is observed, the condition on line 6 is false and thus no other auxiliary nodes are created. Auxiliary nodes created for the previous inputs are already analysed and stored but do not have a separating sequence assigned unless they were processed during the search for a separating sequence of another node; the condition on line 8 allows such auxiliary nodes to be used again. If the condition on line 8 is satisfied, then the auxiliary node is added to *dependent* and a link to it from r is stored to *transitionsTo*. Analysed auxiliary nodes with a separating sequence are like the internal nodes of *ST*, therefore, the score for them is calculated and compared against the best score (lines 15–16). If one does not want to optimize the choice of invalid separating sequences, hence the number of sequences in HSIs can be larger, then lines 6–16 can be omitted. Note that the SCORE function favours valid sequences so $best_r.next$ gets the node with the shortest valid separating sequence if there is one. Finally, if a node r_x with a valid separating sequence is found, then r is sorted into *dependentPriorityQueue* according to the score calculated for the best such r_x.

Algorithm 9: PROCESSDEPENDENT()

1 **while** *dependentPriorityQueue* is not empty **do**
2   pop r from *dependentPriorityQueue* with the lowest score
3   **if** r is not separated **then**
4         $r.separatingSequence \leftarrow best_r.input \cdot best_r.next.separatingSequence$
5   SEPBYCREATINGSUCCESSORS(r)
6   **foreach** $(p, x) \in transitionsTo[r]$ **do**
7   **if** p is not separated **and** SCORE$(p, x, r) < best_p.score$ **then**
8                 $best_p \leftarrow (x, r, \text{SCORE}(p, x, r))$
9   push p with $best_p.score$ into *dependentPriorityQueue*
10   pop r from *dependent*

Nodes in *dependent* are processed using PROCESSDEPENDENT described in Algorithm 9 after their links were initialized either by INITTRANSONVALIDINPUTS or by INITTRANSONINVALIDINPUTS. Besides the links, both functions fill *dependentPriorityQueue* with nodes r for which the separating sequence can be constructed based on the chosen $best_r$'s. Algorithm 9 goes through all nodes r in *dependentPriorityQueue* that is sorted according to the scores given by $best_r$'s. If r is not yet separated, its separating sequence is set to the one of the $best_r.next$ prepended by x leading to $best_r.next$ from r. The r is then removed from *dependent*. As r is now separated, it can help other nodes in *dependent* that lead to it. Therefore, all links (p, x) in *transitionsTo* leading to r are checked (lines 6–9 of Algorithm 9). If a predecessor p is better separated based on r, it is pushed to *dependentPriorityQueue* with its new SCORE(p, x, r).

Algorithm 10: INITTRANSONINVALIDINPUTS(r)

1 **foreach** invalid separating input x of r **do**
2   **if** SCORE(r, x,null) $<$ $best_r.score$ **then**
3         $best_r \leftarrow (x, \text{null}, \text{SCORE}(r, x, \text{null}))$
4 **foreach** invalid transferring input x of r such that $|\delta(r.states, x)| > 1$ **do**
5     $r_x \leftarrow$ GETSEPARATINGNODE$(\delta(r.states, x))$
6   **if** $r_x.separatingSequence$ is not assigned or not valid **then**
7         $r_x \leftarrow$ a (stored or new) node with states equal to $\delta(r.states, x)$
8   **if** r_x is not analysed **then**
9   ANALYSE(r_x)
10   push r_x into *dependent* if it is not there
11   add (r, x) to $transitionsTo[r_x]$
12   **if** $r_x.separatingSequence$ is assigned **then**
13   **if** SCORE$(r, x, r_x) < best_r.score$ **then**
14             $best_r \leftarrow (x, r_x, \text{SCORE}(r, x, r_x))$
15 push r with $best_r.score$ into *dependentPriorityQueue*

Algorithm 10 checks all invalid inputs after all valid transferring inputs are checked by INITTRANSONVALIDINPUTS and no valid separating sequence was observed for the given node r. At first, all invalid separating inputs are checked if any of them can improve the best separating score initialized in INITTRANSONVALIDINPUTS. Then all invalid transferring inputs x that do not merge all states are processed in a similar way as the valid ones were in Algorithm 8. If the lowest common ancestor r_x of the leaves corresponding to the next states $\delta(r.states, x)$ was not processed or has an invalid separating sequence, an auxiliary node relating only to the next states is considered as r_x instead. It is analysed if it was not, and pushed into *dependent* as its separating sequence may be needed to obtain the best invalid sequence for r. The link from r to r_x is stored as well. If r_x already has a separating sequence, it is checked whether r_x is a better basis for the best separating sequence of r and so whether $best_r.score$ can be improved. Finally, r is pushed into *dependentPriorityQueue* with the best score encountered so far. Note that r can be added to

dependentPriorityQueue with a better score later in PROCESSDEPENDENT after one of its r_x's that is not separated gets a separating sequence.

Algorithm 11: SCORE(r, x, r_x)

```
1  w ← x · r_x.separatingSequence if r_x is not null else x
2  if w is a valid separating sequence of r.states then
3      return |w|
4  (a, b, c, d, e, n_r) ← (0, 0, 0, 0, |w|, |r.states|)
5  foreach response z ∈ λ*(r.states, w) do
6      S' ← {s ∈ r.states | λ*(s, w) = z} // states of successor
7      if |S'| = |δ*(S', w)| then   // w valid for S'
8          b ← b + 1 // number of valid successors
9      else
10         a ← a + |S'|       // number of states in invalid successors
11         d ← d + |S'| − |δ*(S', w)|  // undistinguished states
12     c ← c + 1             // number of successors
13 return (((a · n_r − b) · n_r − c) · n_r + d) · n_r + e
```

7.5. Scoring of inputs

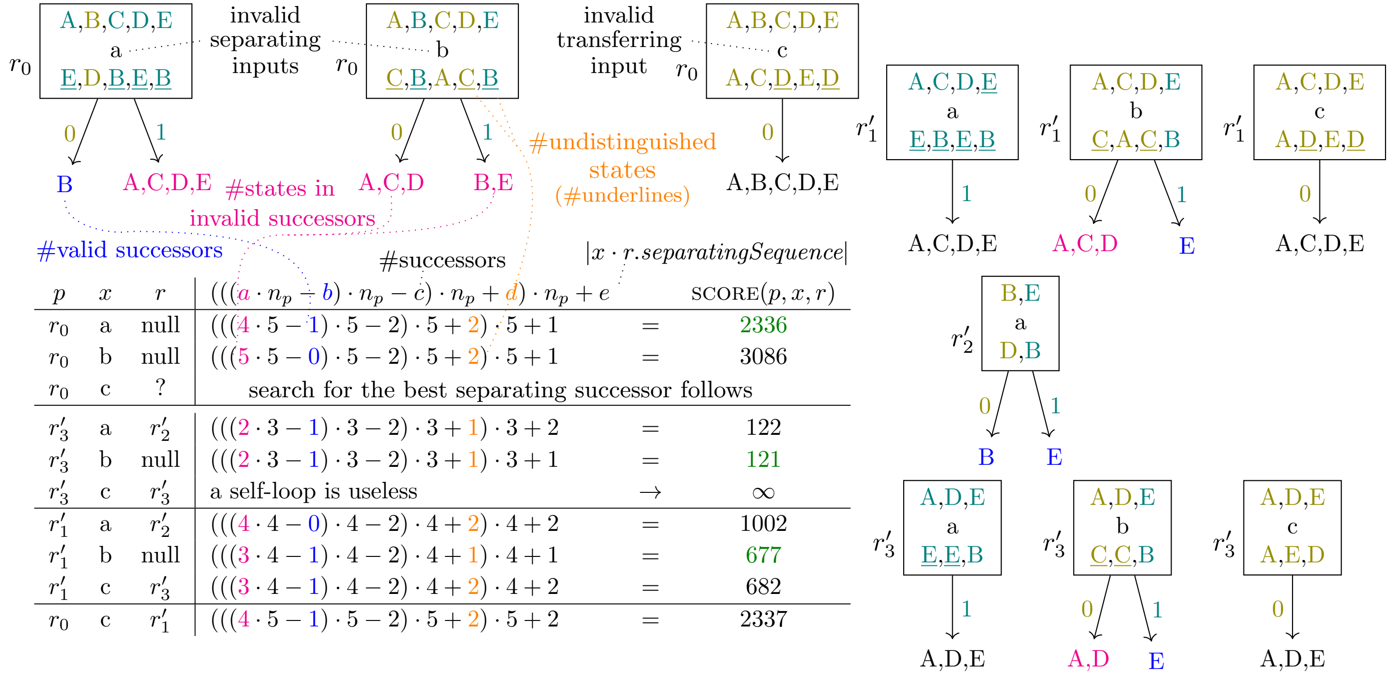
The last part of the ST-IADS algorithm is the scoring function in Algorithm 11. The function SCORE(r, x, r_x) analyses how the states of r would be separated by the sequence $w = x \cdot r_x.\mathit{separatingSequence}$, or only by x if the given r_x is null. If w is a valid sequence for *r.states*, then the length of w is returned. Otherwise, a higher score is returned so that valid sequences are favoured. As the ultimate aim is to have the smallest number of separating sequences for a given subset of states, this scoring function prioritises invalid sequences that are valid for the maximum total number of states in the successors for which the sequence does not merge any two states. A successor r_i of r is valid if *r.separatingSequence* is valid for $r_i.\mathit{states}$, that is, $|r_i.\mathit{states}| = |\delta^*(r_i.\mathit{states}, r.\mathit{separatingSequence})|$. None that $|\lambda^*(r_i.\mathit{states}, r.\mathit{separatingSequence})| = 1$ as r_i is a successor of r. If some states of r_i are merged by *r.separatingSequence*, r_i is an invalid successor of r. The number of undistinguished states of an invalid successor r_i is the difference between the number of states in r_i and the number of their next states on *r.separatingSequence*, that is, $|r_i.\mathit{states}| - |\delta^*(r_i.\mathit{states}, r.\mathit{separatingSequence})|$. There are five parameters *a*-*e* to compare invalid separating sequences. The parameters represent:

- a - the total number of states in invalid successors,
- b - the number of valid successors,
- c - the number of all successors of r,
- d - the total number of undistinguished states,
- e - the length of w.

As the score of the best sequence is the lowest, the parameters b and c that are signs of a good separating sequence decrease the score and the parameters a, d and e are rather bad signs so that they increase the score. The priority of parameters how they influence the score is given by their alphabetical order. The parameter a estimates for how many states another separating sequence will be needed, for example in the construction of HSIs. Therefore, the higher a the less likely the sequence is chosen to be the separating sequence of r. Parameters b and c provide a ratio of the number of 'good' successors to their total number and c also represents how well the states of r are divided by w; the higher c the more state pairs are likely to be separated by w. The parameter d estimates how many states will remain undistinguished when asked to separate a state from the others in *r.states*. The parameters are connected together by the formula on line 13 of Algorithm 11 to get one number evaluating the invalid sequence w. The number n_r of states in r is employed in the formula to order the parameters in the resulting score by their priority. Note that $d < a \leq n_r$ and $b < c < n_r$ as there is always an invalid successor that contains at least two states. Therefore, only e could interfere with d if $e \geq n_r$, but it is acceptable as d provides just an estimate that does not have to be precise and this interference thus penalizes sequences that are too long. Note that the scoring function could be implemented differently which influences the choice of invalid separating sequences and not the correctness of the algorithm.

7.6. Time Complexity

The time complexity of the existing ST-ADS algorithm is $O(n^2p)$ where $n = |S|$ and $p = |X|$ ([9, Theorem 3.2]). There are at most $n-1$ refinements of the partition and so the resulting ST has at most $2n-1$ nodes (n leaves and at most $n-1$ inner nodes). Each node r is analysed in $O(np)$ as all states of r and at most all inputs are checked. The function INITTRANSONVALIDINPUTS (Algorithm 8) prepares links in $O(np)$ and it is called at most n times as *dependent* contains less than n nodes in total during the entire construction procedure. During the preparation of links all valid transferring inputs are checked and for each of them the lowest common ancestor of the next states is found by GETSEPARATINGNODE in $O(n)$. Each node of *dependent* is processed by Algorithm 9 once and it allows one to check all links stored in *transitionsTo*. The total number of links is in $O(np)$ which is also the complexity of PROCESSDEPENDENT in Algorithm 9. Therefore, the ST-ADS algorithm runs in $O(n^2p)$. Moreover, all separating sequences are of the length at most $n-1$ so the space complexity of the splitting tree is $O(n^2)$ if there is an ADS [9]. Note that the proposed pseudocode is based on the implementation of the ST-ADS algorithm proposed in [7] rather than the original one from [9] as [7] simplifies dealing with valid transferring inputs.



p	x	r	$(((a \cdot n_p - b) \cdot n_p - c) \cdot n_p + d) \cdot n_p + e$		SCORE(p, x, r)
r_0	a	null	$(((4 \cdot 5 - 1) \cdot 5 - 2) \cdot 5 + 2) \cdot 5 + 1$	=	2336
r_0	b	null	$(((5 \cdot 5 - 0) \cdot 5 - 2) \cdot 5 + 2) \cdot 5 + 1$	=	3086
r_0	c	?	search for the best separating successor follows		
r'_3	a	r'_2	$(((2 \cdot 3 - 1) \cdot 3 - 2) \cdot 3 + 1) \cdot 3 + 2$	=	122
r'_3	b	null	$(((2 \cdot 3 - 1) \cdot 3 - 2) \cdot 3 + 1) \cdot 3 + 1$	=	121
r'_3	c	r'_3	a self-loop is useless	$\rightarrow$	∞
r'_1	a	r'_2	$(((4 \cdot 4 - 0) \cdot 4 - 2) \cdot 4 + 2) \cdot 4 + 2$	=	1002
r'_1	b	null	$(((3 \cdot 4 - 1) \cdot 4 - 2) \cdot 4 + 1) \cdot 4 + 1$	=	677
r'_1	c	r'_3	$(((3 \cdot 4 - 1) \cdot 4 - 2) \cdot 4 + 2) \cdot 4 + 2$	=	682
r_0	c	r'_1	$(((4 \cdot 5 - 1) \cdot 5 - 2) \cdot 5 + 2) \cdot 5 + 2$	=	2337

Figure 5: Construction of splitting tree – analysis of the root node

The proposed extension potentially increases the time and space complexity by a significant amount but only if no ADS exists. In the worst case $O(2^n)$ (auxiliary) nodes representing all subsets of states could be analysed. This can be easily avoided by omitting lines 6–16 of Algorithm 8 and lines 6–11 of Algorithm 10 that try to find a better invalid separating sequences by introducing auxiliary nodes. Without these lines, the ST-IADS algorithm still works but may result in more sequences in HSIs built from the resulting ST. Nevertheless, the number of created auxiliary nodes is restricted by the transition and output functions of the given machine so that it hardly reaches huge numbers. A possible improvement that could result in lower number of separating sequences in HSIs is the use of a scoring function on valid sequences. This would mean adjusting the existing ST-ADS algorithm, not just extending it as was done by the proposed ST-IADS algorithm.

8. Running Example

The ST-IADS algorithm is described in this section how it builds the splitting tree showed in Figure 2 for the Mealy machine M defined in Figure 1. Examples describing just the ST-ADS algorithm can be found in [9, 7].

The algorithm starts with the root r_0 of ST that contains all states A–E. As M does not have state outputs, r_0 is the only node in *partition* (line 6 of Algorithm 5). The root is then popped from *partition* and analysed. The analysis of all three input symbols is captured in Figure 5 where the scoring function is also explained. The inputs 'a' and 'b' are separating as the states respond to them with 2 different outputs; the input 'c' is transferring. Notice that states and next states in the root r_0 have the colour of the corresponding output. As all inputs merge some states, there is no valid input for r_0 and null would be returned as a sign that M has no ADS if *validOnly* was true. The root is thus added to *dependent* that is immediately processed by PROCESSDEPENDENT because *partition* is empty and there is no valid transferring input that could be checked by INITTRANSONVALIDINPUTS. However, *dependentPriorityQueue* is empty so that PROCESSDEPENDENT exits and r_0 is still in *dependent*. Hence, INITTRANSONINVALIDINPUTS is called to prepare links from r_0.

INITTRANSONINVALIDINPUTS first calculates score for the invalid separating input 'a' such that r_0 with 5 states would have on 'a' 4 states in the invalid successor, 1 valid successor out of 2 successors, 2 undistinguished states and the separating input has the length of 1. SCORE(r_0, a, null) is thus 2336. States A, D and B, E merge in their corresponding successors on the input 'b', therefore, there is no valid successor of r_0 on 'b' and so the score 3086 is worse than on 'a'. The invalid transferring input 'c' does not merge all states and so it is checked if it can begin a better invalid separating sequence than 'a'. The lowest common ancestor r_x of the leaves relating to the next states $\delta(r_0.states, c)$ is the root itself and as it has no separating sequence assigned yet, an auxiliary node r'_1 is created; $r'_1.states = \{$A, C, D, E$\}$. The analysis of r'_1 for all inputs is shown on the right of Figure 5; 'a' and 'c' are invalid transferring for r'_1 and 'b' is an invalid separating input. The node r'_1 is added to *dependent* and a link (r_0, c) is stored into *transitionsTo*$[r'_1]$. As r'_1 does not have a separating sequence yet, it cannot improve the best score for r_0 and so r_0 with the score of 2336 relating to the invalid sep-

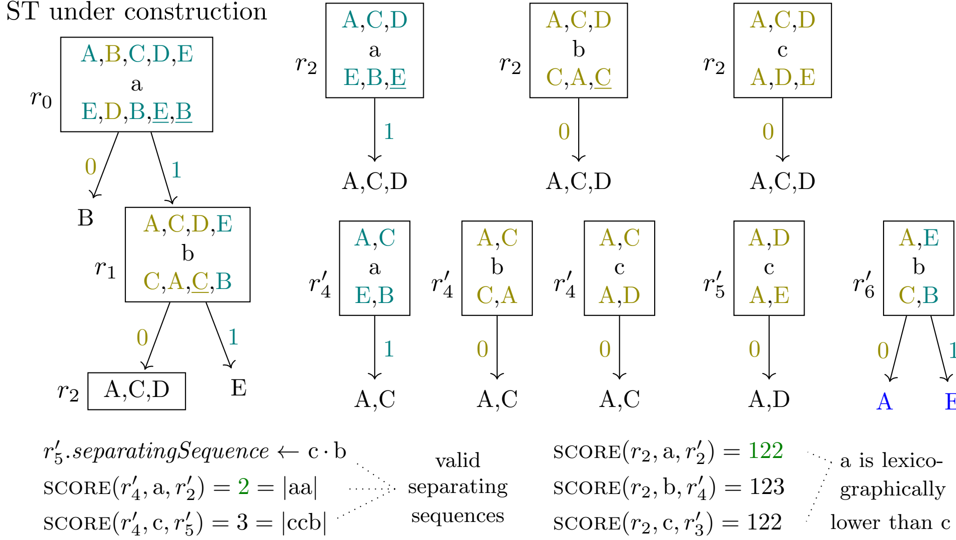


Figure 6: Analysis and separating node r_2 with states A,C,D

arating input 'a' is pushed into *dependentPriorityQueue*.

The next cycle of 'foreach' loop on line 22 of Algorithm 5 chooses to initialize links from the node r'_1. The node has no valid inputs so again INITTRANSONINVALIDINPUTS is called. After the score of 677 is calculated for the invalid separating input 'b', both transferring inputs are processed. In both cases, auxiliary nodes are created; $r'_2.states = \delta(r'_1.states, a) = \{B,E\}$ and $r'_3.states = \delta(r'_1.states, c) = \{A,D,E\}$. A valid separating input 'a' is found for the auxiliary node r'_2 during the analysis of inputs, hence, the analysis is not finished. SCORE(r'_1, a, r'_2) can be thus calculated. Nevertheless, the value 1002 does not improve the best score for r'_1 relating to the input 'b'. Fortunately, the node r'_2 is then reused to calculate the score for the auxiliary node r'_3 on the invalid transferring input 'a'. The score of 122 is worse than the best one set by the invalid separating input 'b'; the scores differ only by 1 because of the length of separating sequence 'aa'. The states of node r'_3 are transferred to themselves on the input 'c' which means that 'c' cannot start the shortest separating sequence of a good score; such nodes can be thrown away already during the analysis of inputs. After r'_3 is processed by INITTRANSONINVALIDINPUTS, *dependentPriorityQueue* is filled up with r'_3 (score 121), r'_1 (677) and r_0 (2336). PROCESSDEPENDENT pops r'_3 first and sets 'b' as its separating sequence. Then, it tries to improve the best of r'_1 but SCORE(r'_1, c, r'_3) = 682 is not better. The same happens to r'_1 that is popped from *dependentPriorityQueue* next. It gets 'b' as the separating sequence and SCORE(r_0, c, r'_1) = 2337 does not improve the best so that r_0 is separated with 'a'. Analyses of inputs for r'_2 and r'_3 and the calculation of scores are shown in Figure 5. Notice that the separating sequence 'cb' has the score worse than the selected separating input 'a' just because of its length.

Separating the root r_0 with 'a' results in two new leaves and the update of *partition* and *ST.separatingNodes*. One leaf represents state B and so it will not be further processed. The second leaf r_1 represents the other states. *ST.separatingNodes* is thus updated to point to r_0 for all state pairs associated with state B. Both leaves form the current partition but r_1 is popped from *partition* immediately when the second cycle of the main loop starts (line 8 of Algorithm 5). Fortunately, r_1 is the same as the auxiliary node r'_1 so that it is directly separated with 'b' that was analysed as the separating sequence of r'_1. Note that r'_1 with its successors just replaces r_1 in the *ST* in the implementation.

The current partition is updated to contain two singletons representing states B and E, and the leaf r_2 that includes states A, C, D. The splitting tree in the current form is shown on the left of Figure 6. The node r_2 is popped from *partition* and analysed as it is not stored amongst the auxiliary nodes. It has no valid separating sequence which leads to r_2 being added to *dependent* and the algorithm tries to find its separating sequence right after that as there is no other leaf with the same number of states in *partition*. This time there is a valid transferring input and a separating sequence is assigned to the node to which the input leads. Thus, $best_r$ for r_2 is initialized in INITTRANSONVALIDINPUTS with the input 'c' leading to r'_3 and the score of 122. Note that r_1 as the lowest common ancestor of the leaves containing states A, C, E is considered first instead of r'_3 but as r_1 also contains state D and its sequence 'b' is invalid, r'_3 is chosen. As the separating sequence 'b' of r'_3 is invalid, r_2 is not pushed into *dependentPriorityQueue* and so the first call of PROCESSDEPENDENT does not change anything. All the shortest invalid separating sequences for r_2 need to be compared to choose the one that separates it. The auxiliary nodes created during the search for the best invalid sequence are shown in Figure 6. In the case of r'_5 relating to states A and D, the inputs 'a' and 'b' are not visualized as they merge

the states so that they cannot begin a separating sequence. The node r'_6 has a valid separating input 'b' so that only this input is used and shown. After all the needed auxiliary nodes are created, analysed and connected by links, the nodes of *dependent* are processed by Algorithm 9 as sketched at the bottom of Figure 6. The implementation checks all inputs in one pass so that alphabetically lower inputs with the minimal score are favoured. Six auxiliary nodes were created and two of them were later reused. As $n = 5$ and the total number of explored nodes is $9+4 = 13$, the space complexity seems to be closer to $\Theta(n^2)$ than in $\Theta(2^n)$.

9. Experiments

Our new extension was used to construct harmonized state identifiers from the splitting tree and then these HSIs was used in the HSI- and SPY- methods to show the improvement. Altogether 8 testing methods were compared on randomly-generated machines. The results of experiments are described in this section. Our implementation of each method used for experimental evaluation is described in [15] and available in FSMlib v3.1[1].

The FSMlib contains a generator of random DFSM models. The DFSM generator first assigns the target state to each transition randomly and then changes some of the transitions such that each state is reachable from the initial state. The outputs are also assigned randomly but such that each output symbol is captured at least once in the machine. If the generated machine is not minimal, it is thrown away and another machine is generated. This process is repeated until the given number of minimal completely-specified machines with the given numbers of states, inputs and outputs is obtained. The experiments consist of 1700 DFSMs, 1700 Mealy machines and 1700 Moore machines with 5 inputs and 5 outputs. There are 17 groups of 100 machines with different number of states for all three machine types. The number of states of these 17 'state groups' are: multiples of 10 ranging from 10 to 100 (10 groups) and 150, 200, 300, 400, 600, 800 and 1000.

Each of the 8 testing methods constructs 3 so-called m-complete test suites ($m = l + n$) for each of 5100 machines depending on the given number l of extra states that is 0, 1 or 2. All machines and the results are available in the repository FSMmodels v1.1[2]. As mentioned in Section 2.3, the idea of the m-complete test suite is to make it possible to find any fault in an implementation with up to m-states and where the expected number of states in an implementation is potentially greater than that in a specification $l > 0$, the size of a test suite may increases exponentially.

The exploration efficiency is a new objective developed by the authors of this paper. It is calculated as the number of edges in the testing tree of T divided by the total length of tests in T. As it is based on the testing tree, it permits one to evaluate how much of the implementation will be explored by tests, even in the implementation with much more states than the specification. Moreover, it captures how many prefixes of tests are overlapping with other tests, for example, fixed access sequences are covered by several tests. The exploration efficiency is thus higher (and better) if a testing method constructs longer sequences that do not significantly overlap.

Figure 7 shows the results for Moore machines and 0 extra states. It compares the testing methods on 4 objectives: the total number of inputs in the constructed test suite T, the number of tests in T, the exploration efficiency and the time spent by the construction of T. Each of 4 graphs show the first and third quartiles calculated for each state group of 100 machines, and boxplots with minimum and maximum values as whiskers for the machines with 1000 states.

The performance of the HSI- and SPY- methods is well improved using the HSIs constructed from ST-IADS instead of the HSIs formed of the shortest separating sequences of all state pairs. This can be seen in Figure 7 as the testing methods using ST-based HSIs (labelled 'HSI/ST' and 'SPY/ST') produce the smallest test suites (2-3 times fewer sequences) and outperform even the most advanced methods, the H- and SPYH- methods. The construction of the splitting tree increases time but it is still less than in the case of the W- and Wp- methods that try to minimize the characterizing set before its use. Moreover, the growth of the construction time with respect to the number of states is does not correspond to the worst-case time complexity derived at the end of Section 7: it takes 0.2 sec for 600 states and 0.6 sec for 1000 states.

Figure 7 captures just one setting out of 9 possible (3 machine types and 3 different numbers of extra states). The results of other settings capture the same trends that the HSI/ST and SPY/ST are better or comparable with the most advanced methods, the H-, SPY- and SPYH-methods. Therefore, the improvement by the proposed ST-IADS algorithm is confirmed on randomly-generated machines.

10. Conclusion

This paper describes the new ST-IADS algorithm that allows one to construct a splitting tree for any deterministic finite-state machine. With this tree, separating sequences for any subset of states, incomplete adaptive distinguishing sequences and harmonized state identifiers (HSI) can be easily derived from them. If the machine has an adaptive distinguishing sequence (ADS), the algorithm will return it. In the absence of an ADS, a number of incomplete adaptive distinguishing sequences (IADSs) are produced so that state identification would require resetting a system under test as few times as possible. When used for the purpose of HSI construction, the sequences from a splitting tree lead to the smallest number of sequences

[1] https://github.com/Soucha/FSMlib/releases/tag/v3.1
[2] https://github.com/Soucha/FSMmodels/releases/tag/v1.1

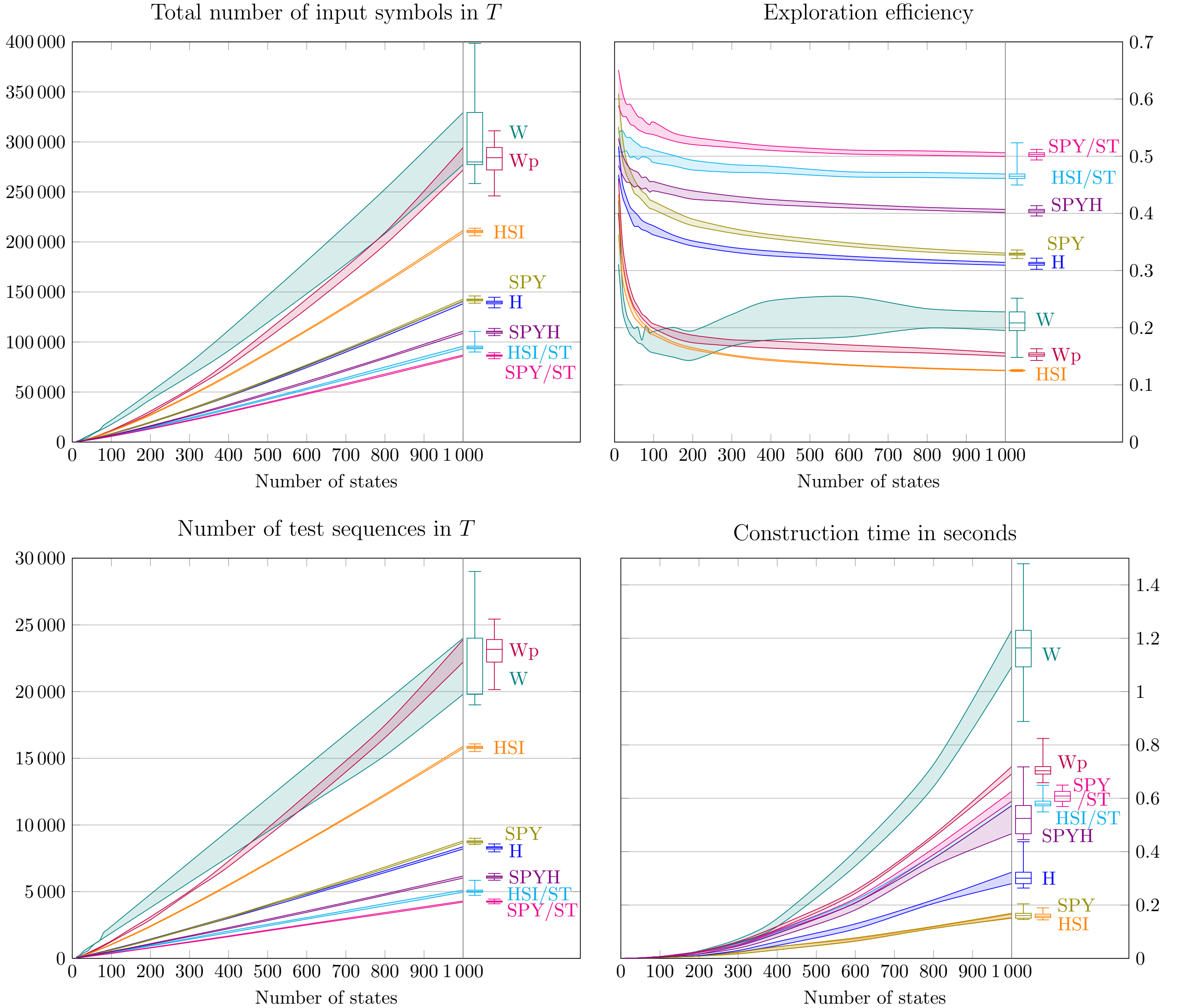


Figure 7: Construction of test suites T for Moore machines with 5 inputs, 5 outputs and no extra states: first and third quartiles calculated for 100 machines per each state group; boxplots with whiskers from minimum to maximum for machines with 1000 states are shown on the right of each graph

in HSIs as opposed to HSIs of numerous short separating sequences; this choice permits testing with as few resets as possible and therefore test suites are significantly smaller: in the experiments conducted by the authors on 5100 randomly-generated machines the improvement was 2-3 times for testing methods using the splitting tree.

The computational effort to construct a splitting tree with these properties is significant in the theoretical worst case. In the experiments nothing even remotely close to a worst case was encountered: an automaton with 1000 states could be handled in around half a second.

Future work involves optimisation of test sequence generation for automata without reset, where reset has to be approximated with exponentially long test sequences that are guaranteed to re-enter a state of interst. In this context, although the reduction of the number of sequences in a test suite using IADSs would make a major improvement, one might be get even better results by integrating state verification and testing such as by extending the SPYH-Method [14] developed by the authors.

References

[1] A. Petrenko, Checking experiments with protocol machines, in: Proceedings of the IFIP TC6/WG6. 1 Fourth International Workshop on Protocol Test Systems IV, North-Holland Publishing Co., Amsterdam, The Netherlands, 1991, pp. 83–94.

[2] A. Simão, A. Petrenko, N. Yevtushenko, On reducing test length for FSMs with extra states, Software Testing, Verification and Reliability 22 (6) (2012) 435–454.

[3] R. M. Hierons, U. C. Türker, Incomplete distinguishing sequences for finite state machines, The Computer Journal 58 (11) (2015) 3089–3113.

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

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

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

[7] M. Soucha, Finite-state machine state identification sequences, Bachelor's thesis (2014).

[8] R. Smetsers, J. Moerman, D. N. Jansen, Minimal separating sequences for all pairs of states, in: International Conference on Language and Automata Theory and Applications, Springer, 2016, pp. 181–193.

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

[10] G. Luo, A. Petrenko, G. v. Bochmann, Selecting test sequences for partially-specified nondeterministic finite state machines, in: Protocol Test Systems, Springer, 1995, pp. 95–110.

[11] M. Soucha, Checking experiment design methods, Master's thesis, Czech Technical Univerzity in Prague (2015).

[12] S. Fujiwara, F. Khendek, M. Amalou, A. Ghedamsi, et al., Test selection based on finite state models, Software Engineering, IEEE Transactions on 17 (6) (1991) 591–603.

[13] R. Dorofeeva, K. El-Fakih, N. Yevtushenko, An improved conformance testing method, in: F. Wang (Ed.), Formal Techniques for Networked and Distributed Systems-FORTE 2005, Springer, Taipei, Taiwan, 2005, pp. 204–218.

[14] M. Soucha, K. Bogdanov, SPYH-method: An improvement in testing of finite-state machines, in: 2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), IEEE, Vasteras, Sweden, 2018, pp. 194–203.

[15] M. Soucha, Testing and active learning of resettable finite-state machines, Ph.D. thesis, University of Sheffield (2019).