# Using Learning Techniques to Generate System Models for Online Testing

Edith Werner, Sergei Polonski, Jens Grabowski
Software Engineering for Distributed Systems Group,
Institute for Computer Science, University of Göttingen,
Lotzestr. 16–18, 37083 Göttingen, Germany.
{ewerner,grabowski}@cs.uni-goettingen.de,
spolonski@stud.cs.uni-goettingen.de

**Abstract:** Today's software systems are mostly modular and have to be changeable. However, the testing of such systems becomes difficult, especially when changes are applied after deployment. One way to passively test such a system is to check whether the observed traces are accepted by a system model. In this paper, we present a method to generate a model of the *System Under Test* from its test cases. We adapt Angluin's algorithm for learning finite automata to the special case of learning from traces obtained from test cases and provide the promising results of our experiment.

## 1 Introduction

Today's software systems are mostly modular and have to be changeable. The testing of such systems becomes difficult, especially when changes are applied after deployment. However, modeling organically grown systems under laboratory conditions is not always possible, so passive online testing may be the only possible way to assess the deployed system. To this end, we need an oracle that accepts or rejects the observed behavior, e.g. a system model that accepts or rejects the observed traces of the *System Under Test* (SUT).

A promising approach to the reconstruction of system models is to use learning algorithms, as has been shown for example by [CM96], [HNS03], and [SLG07]. However, all those approaches rely on the execution of active test cases against the SUT. By contrast, we suggest an approach to learn a system model from the system's test cases without probing the SUT itself. Test cases are almost always available and often more consistent to the system than any other model. Also, they usually take into account all of the system's possible reactions to a stimulus, thereby classifying the anticipated correct reactions as accepted behavior and the incorrect or unexpected reactions as rejected behavior. Simply put, we generate a system model for passive testing from the artifacts used in active testing.

The paper is structured as follows. In Section 2 we present the foundations of the learning algorithm and our adaptation to accept test case traces as input. Then, in Section 3, we discuss the first results of our approach and give an outlook on further research. In Section 4, we summarize the paper.

## 2 Learning from Test Cases

The technique of learning finite automata using queries was introduced by Dana Angluin [Ang87]. The main idea of the algorithm is to successively discover the states and transitions of an automaton, further called the *target automaton*, by querying an oracle, called *teacher*. The algorithm uses two types of queries: *membership queries* discover whether a given sequence is accepted by the target automaton, *equivalence queries* ask whether the learned automaton, called *hypothesis automaton*, already matches the target automaton. If the hypothesis automaton differs from the target automaton, the teacher answers by giving a counter-example. The already gathered information is stored in a data structure called *classification tree*, which is mainly a binary decision tree whose nodes are labeled with sequences of the target automaton.

The main flow of the learning algorithm is as follows. The hypothesis automaton is initialized with one state, the start state. A membership query on the empty sequence determines whether the start state is accepting or not. Subsequently, as long as the hypothesis automaton is not equivalent to the target automaton, the classification tree is updated with help of the counter-example, i.e. the counter-example is added to the classification tree by creating a new path in the tree. Then, the classification tree is used to generate a new hypothesis automaton by using the leaves of the classification tree as states of the automaton and creating the transitions by asking membership queries on the concatenation of access and distinguishing strings.

To learn a finite automaton from test cases, the Angluin's algorithm has to be adapted in two ways. First, test cases have to be defined in the context of the learning algorithm. Then, the two query mechanisms of the algorithm, membership queries and equivalence queries, have to be redefined.

### 2.1 Test Cases as Inputs

For the purposes of our paper, we will use a *test case* as the basic concept. A *test case* $t$ is a tuple $(w, \mathrm{expect}(w))$ where $w$ is a sequence of inputs and outputs to the software and $\mathrm{expect}(w) \in \{\text{accept}, \text{reject}\}$ determines whether the test sequence $w$ should be accepted or rejected by the system. Without loss of generality, we assume that every test case begins in the start state of the software. In the following, we will denote the input/output sequence of a given test case by $w(t)$ and similarly the expected test result as $\mathrm{expect}(w(t))$. Since Angluin's learning algorithm assumes an automaton without outputs, inputs and outputs of the software will be equally treated as inputs of the automaton.

An accepting test case $t$, where $\mathrm{expect}(w(t)) =$ *accept*, maps to a sequence that is accepted by the target automaton. Accordingly, a rejecting test case, where $\mathrm{expect}(w(t)) =$ *reject*, maps to a sequence that is not accepted by the target automaton. The collection of all test cases belonging to the software is called a *test suite* $\mathcal{TS}$, which contains both accepting and rejecting test cases.

### 2.2 Adapting the Query Mechanisms

The most important mechanism of the learning algorithm is the membership query, which determines the acceptability of a certain behavior. In our case, the behavior of the software and thus also of the target automaton is defined by the test cases. Since the test cases are our only source of knowledge, we assume that the test cases cover the complete behavior of the system and thus redefine the membership query as follows: A sequence $w$ is *accepted* by the automaton if it matches an accepting test case in the test suite. Assuming a closed world, we simply state that every behavior that is not explicitly allowed must be erroneous and therefore has to be rejected, i.e. $\textit{rejected} \equiv \neg\textit{accepted}$. Likewise, the hypothesis automaton is *equivalent* to the target automaton, if for every test case in the test suite, the processing of its test sequence on the automaton is accepted or rejected as specified by the expected test result. The first test sequence that violates its expected test result is returned as a counter-example. The remaining parts of the learning algorithm can be adopted without changes.

## 3 Experimental Results

The results of our first experiments are promising: it is possible to reproduce an automaton model from its test cases. For the experiments, we used the state machine of the Initiator entity of Inres protocol [Hog91]. Since from the plain construction of an automaton we cannot assess the validity of our approach, we actually started by generating a *reference automaton*. From the reference automaton, we generated accepting and rejecting test cases that we used as inputs to the learning algorithm. In the last step, we reconstructed the automaton from the test cases by way of the adapted learning algorithm as described in Section 2. For the Inres protocol, we were able to reconstruct a reference automaton with six states from a test suite consisting of twelve accepting and three rejecting test cases.

One of the main observations of our experiments is that the closed world assumption is questionable for several reasons. When constructing the positive test cases, we observed that it is not enough to cover the transitions of the reference automaton. In case the software can behave cyclically, then transition covering test cases will not necessarily unroll those loops. Then, when a membership query is asked for a path that loops twice, the path will be rejected on the grounds that there is no test case representing this behavior. A possible solution to this problem would be a preprocessing of the test suite by a loop detection algorithm, annotating the possible loops and thus enabling on-the-fly loop unrolling during the actual learning.

Also, it is arguable whether a test suite can in fact cover the complete behavior. In Section 2.2, we assumed a closed world and stated that every behavior that is not explicitly allowed must be erroneous and therefore has to be rejected. However, since the learning algorithm is accumulative, i.e. the algorithm can only learn new transitions and states but never forget them, in the case that an accepting test case was missing the algorithm would have to be started from scratch. Assuming an open world, a possibility would be to reject

only sequences that are matched to a negative test case. The drawback of this option is that then not all membership queries can be answered, since some sequences cannot be matched to test cases. Several possible solutions to this question remain to be explored.

Further, we plan to investigate test case generation methods that are based on *Unique Input/Output Sequence* (UIO) or *Distinguishing Sequence* (DS) with respect to our learning algorithm. It is most likely that methodically generated test suites will be more analyzable and therefore easier to expand.

## 4 Summary

We presented an approach to generate models for online testing by using a learning algorithm that we adapted to learn a behavioral model from test cases. The adaptation was done in two steps: First, test cases were defined in terms compatible to the algorithm. Second, the key features of the algorithm, membership and equivalence queries were redefined suitably. The adapted algorithm was implemented in a prototypical tool [Pol08]. To evaluate the concept, we then applied the algorithm to an examplary system. The first experimental results were promising; we were able to reproduce our reference automaton from the system's test suite. Finally, we gave an overview on possible optimizations of the algorithm. In the long term, the learning approach could also be used to assess the quality of test suites, e.g. by comparing the automaton learned to an existing model and thereby determine the coverage of the underlying test suite.

## References


[Ang87] D. Angluin. Learning Regular Sets from Queries and Counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[CM96] D. Carmel and S. Markovich. Learning Models of Intelligent Agents. In Howard Shrobe and Ted Senator, editors, *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference, Vol. 2*, pages 62–67, Menlo Park, California, 1996. AAAI Press.

[HNS03] H. Hungar, O. Niese, and B. Steffen. Domain-Specific Optimization in Automata Learning. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 315–327. Springer, 2003.

[Hog91] D. Hogrefe. OSI Formal Specification Case Study: The Inres Protocol and Service. Technical Report IAM-91-012, University of Berne, Institute for Informatics and Applied Mathematics, May 1991.

[Pol08] Sergei Polonski. Learning of protocol-based automata. Master's thesis, Institute for Computer Science, University of Göttingen, Germany, GAUG-ZFI-MSC-2008-09, May 2008.

[SLG07] M. Shahbaz, K. Li, and R. Groz. Learning and Integration of Parameterized Components Through Testing. In Alexandre Petrenko, Margus Veanes, Jan Tretmans, and Wolfgang Grieskamp, editors, *TestCom/FATES*, volume 4581 of *Lecture Notes in Computer Science*, pages 319–334. Springer, 2007.