Mining Test Cases: Optimization Possibilities

Edith Werner* and Jens Grabowski†
*Neumüller Ingenieurbüro GmbH, Nürnberg, Germany
edithbmwerner@googlemail.com
†Software Engineering for Distributed Systems Group,
Institute for Computer Science, University of Göttingen, Göttingen, Germany
grabowski@cs.uni-goettingen.de

***Abstract*—System monitors need oracles to determine whether observed traces are acceptable. One method is to compare the observed traces to a formal model of the system. Unfortunately, such models are not always available — software may be developed without generating a formal model, or the implementation deviates from the original specification. In previous work, we have proposed a learning algorithm to construct a formal model of the software from its test cases, thereby providing a means to transform test cases for offline testing into an oracle for monitoring. In this paper, we refine our learning algorithm with a set of state-merging rules that help to exploit the test cases for additional information. We discuss our approach in detail and identify optimization areas. Using the additional information mined from the test cases, models can be learned from smaller test suites.**

Keywords-Machine Learning; Reverse Engineering; Testing

I. Introduction

Today, software systems are generally designed to be modular and reusable. A common scenario of a modular, reusable system is a web service, where simple services are accessed as needed by various clients and orchestrated into larger systems that can change at any moment. While the vision of ultimate flexibility is clearly attractive, there are also drawbacks, as the further usage of a module is difficult to anticipate. In this scenario, it may be advisable to monitor a system for some time after its deployment, to detect erroneous usage or hidden errors.

Monitors are used to observe the system and to assess the correctness of the observed behavior. To this end, monitors need oracles that accept or reject the observed behavior, e.g., a system model that accepts or rejects the observed traces of the monitored system. Unfortunately, the increasing usage of dynamic software development processes leads to less generation of formal models, as the specification of a formal model needs both time and expertise. Generating a formal model in retrospect for an already running system is even harder, as the real implementation often deviates from the original specification.

We propose a method for learning a system model from the system's test cases without probing the System Under Test (SUT) itself [1]. When test cases are available, they often are more consistent to the system than any other model. Ideally, they 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. As the test cases are developed in parallel to the software, they provide a means to judge the correct behavior of the system. Also, test cases are generated at different levels of abstraction, e.g., for unit testing, integration testing, and system testing. By selecting the set of test cases to be used, the abstraction level of the generated model can be influenced.

The basis of our approach is a learning algorithm, first introduced by Angluin [2], which learns a Deterministic Finite Automaton (DFA). To learn from test cases, we adapted the query mechanisms of the algorithm [3]. Experiments with our approach show that while a model can be learned this way, the algorithm only accepts simple traces as input, thereby losing additional information from the test cases, e.g., regarding branching, default behavior, or synchronization. We believe that exploitation of this additional information would enhance the learning algorithm.

In this paper, we propose a state-merging approach, termed *semantic state-merging*, which exploits the semantic properties of test cases in order to identify implicitly defined behavior. We first define a data structure, the *trace graph*, to store the available test cases. Then, we define merging rules for cyclic test cases and for test cases with default branches for the construction of the trace graph. Based on the experiences gathered through a prototypical implementation, we identify optimization areas and possible solutions.

The remainder of this paper is structured as follows. Section II gives an overview on related work. In Section III, we introduce the foundations of our work in testing and machine learning. Section IV describes the trace graph and its construction. Based on this, Section V defines our approach to semantic state-merging on test cases. Subsequently, in Section VI, we give an overview on our experimental results. Section VII discusses the learning approach and describes solution ideas to open questions. In Section VIII, we conclude with a summary and an outlook.

II. Related Work

As our approach combines learning techniques and state merging, we need to take into account related work from

both areas. In the following, we give an overview on relevant articles regarding the adaptation of Angluin's learning algorithm and state-merging and establish the differences to our own work.

A. Related Work in Learning

During the last years, a number of approaches have adapted Angluin's learning algorithm in combination with testing. Mainly, the approaches focus on the learning side of the problem and refine the properties of the generated model. Among the most recent adaptations are approaches to learning Mealy machines [4] and parameterized models [5], [6], [7], [8]. Some approaches can handle large or even infinite message alphabets [5] or potentially infinite state spaces [6]. In all those approaches, the learning algorithm generates test cases that are subsequently executed against the SUT, so that the System Under Test itself is the oracle for the acceptability of a given behavior.

Some approaches use outside guidance to improve the learning approach. The algorithm presented in [9] learns workflow petri nets from event logs and handles incomplete data by asking an external teacher. In [10], learning is used in a modeling approach. In this approach, a domain expert provides Message Sequence Charts representing desired and unwanted behavior.

Our approach differs from the above in two aspects. First, we aim at generating a model for online monitoring. To this end, we need a model that is independent from the implementation itself. Therefore, we can neither use the implementation as an oracle nor learn from event traces generated by the implementation. Instead, we choose to learn from a test suite that was developed due to external criteria. Using a test suite also leads to the second difference of our approach. Where other approaches rely on unstructured data, a test suite provides relations between the distinct traces. We exploit those relations in order to enhance our learning procedure. Where other approaches address the learning side of the problem, our focus is actually on the structure of the teacher.

B. Related Work in Stage-Merging

The basic idea of the state-merging approach is to analyze examples of the target automaton, identify possible states, and merge similar states until the remaining states are considered to be sufficiently distinct. The notion was first introduced by Biermann [11], who used it to generate computer programs from short code samples. Meanwhile, the merging techniques have been extended to logical evaluation of the samples [12], [13] and different heuristics have been introduced [14].

Also, different input domains have been explored. One the one hand, there are approaches that synthesize models from partial models like scenario diagrams [15], [16], [17], [18]. On the other hand, there are approaches that reconstruct a behavioral model of an existing software by merging observed traces [19], [20], [21].

In all cases, the main problem in state-merging is overgeneralization, i.e., a false merging of states, thereby overly simplifying the model. In our approach, we derive the merging rules from the semantic context of our domain, i.e., the properties of the test specification language. This leads to a more conservative merging and avoids erroneous mergings, while nevertheless enlarging the sample space sufficiently.

III. Foundations

In the following, the foundations of testing and on the learning of DFA are presented.

A. Testing

A test case is itself a software program. It sends stimuli to the SUT and receives responses from the SUT. Depending on the responses, the test case may branch out, and a test case can contain cycles to test iterative behavior. To each path through the test case's control flow graph, a verdict is assigned. A common nomenclature is to use the verdict **pass** to mark an accepting test case and the verdict **fail** to mark a rejecting test case. An *accepting* test case is a test case where the reaction of the SUT conforms to the expectations of the tester. This can also be the case, when an erroneous input is correctly handled by the SUT. Accordingly, a *rejecting* test case is a test case where the reaction of the SUT violates its specification. Depending on the test specification, there may be additional verdicts, e.g., the Testing and Test Control Notation version 3 (TTCN-3) [22] extends the verdicts **pass** and **fail** with the additional verdicts **none**, **inconc**, and **error**: **none** denotes that no verdict is set; **inconc** indicates that a definite assessment of the observed reactions is not possible, e.g., due to race conditions on parallel components; and **error** marks the occurrence of an error in the test environment. During the execution of a test case, the verdict may be changed at different points. The overall assessment of a test case depends on the verdicts set along the execution trace, and is computed according to the rules of the test language. E.g., in TTCN-3, the overall verdict may only be downgraded, i.e., once an event was rated as **fail** the overall verdict may not go back to **pass**. For most SUTs, there is a collection of test cases, where each test case covers a certain behavioral aspect of the SUT. Such a collection of test cases for one SUT is called a *test suite*.

The main objective when constructing test cases for a software system is to assure that the specified properties are present in the SUT. To test against a formal specification, e.g., in the form of a DFA, test cases are derived from the model by traversing the model so that a certain coverage criterion is met, e.g., *state coverage* or *transition coverage*. State coverage means that every state of the model is visited by at least one test case. Transition coverage means that every transition of the model is visited by at least one test

case. The largest possible coverage of a system model is *path coverage*, where every possible path in the software is traversed.

B. Learning a Finite Automaton Model from Test Cases

Our learning approach is based on a method proposed by Angluin [2]. The algorithm consists of the *teacher*, which is an oracle that knows the concept to be learned, and the *learner*, who discovers the concept. The learner successively discovers the states of an unknown target automaton by asking the teacher whether a given sequence of signals is acceptable to the target automaton. To this end, the teacher supports two types of queries. A *membership query* evaluates whether a single sequence of signals is a part of the model to be learned. An *equivalence query* establishes whether the current hypothesis model is equivalent to the model to be learned.

For learning from test cases, we need to redefine the two query types in relation to test cases. The most important mechanism of the learning algorithm is the membership query, which determines the acceptability of a given behavior. In our case, the behavior of the software and thus 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. In consequence, we state that every behavior that is not explicitly allowed must be erroneous and therefore has to be rejected, i.e., $rejected \equiv \neg accepted$. Accordingly, we accept a sequence of signals if we can find a **pass** test case matching this sequence, and reject everything else.

The equivalence query establishes conformance between the hypothesis model and the target model. This is exactly what a test suite is designed for, therefore, we redefine the equivalence query as an execution of the test suite against the hypothesis model, where every test case in the test suite must reproduce its verdict. A detailed description of the learning algorithm can be found in [3], [23].

IV. Representing Test Cases

For the learning procedure, it is important that queries can be answered efficiently and correctly. Therefore, we need a representation of the test suite that is easy to search and provides a means to compactly store a large number of test cases. In the following, we define the *trace graph* as a data structure and describe its construction.

A. The Trace Graph

As described in Section III-A, a test case is itself a piece of software and can therefore be represented as an automaton containing a number of event sequences. Usually, a test case distinguishes events received from the SUT, events sent to the SUT, and internal actions like value computation or setting verdicts. Each possible path through the test case must contain the setting of a verdict.

For the learning procedure, we only regard input and output events as the transitions in our target model and ignore internal actions except for the setting of verdicts. The verdicts are used to identify accepting test cases.

In general, every test case combines a number of traces, depending on the different execution possibilities. At the same time, a test suite contains a number of test cases, where different test cases may contain identical traces as they partly overlap. To present the test cases to the learning algorithm, we combine all traces from all test cases in the test suite into a single data structure, the *trace graph*, thereby eliminating duplicates and exploiting overlaps.

To enable an efficient search on the test cases, the trace graph is based on a labeled search tree, where all traces share the same starting state. Traces with common prefixes share a path in the trace graph as long as their prefixes match. For the state-merging approach, the nodes in the trace graph are annotated with the verdicts. Cycles in the test cases are represented in the trace graph by routing the closing edges back to the starting node of the cycle. For better control, nodes where a cycle starts are also marked.

The trace graph forms the basic data structure for our semantic state-merging. The semantic state-merging methods depend on the information contained in the test cases, which in turn depends on the test language. To represent this, the trace graph can be extended to represent diverse structural information on the test cases by defining additional node labels. That way, information on the test cases will only affect the construction of the trace graph, but not the learning procedure that depends on its structure.

B. Constructing the Trace Graph

To construct the trace graph, we dissect the test cases into single traces and add them to the trace graph. Starting in the root of the trace graph, the signals in the trace to be added are matched to the node transitions in the trace graph as far as possible. We call this part of the trace the *common prefix*. The remainder of the new trace, the *postfix*, is then added to the last matched node. Algorithm 1 describes the procedure in pseudo code.

Cycles of the test case automaton need special treatment, as a cycle means that an edge loops back to an existing node. To this end, we separate the cyclic traces into three parts, a prefix leading into the cycle, the cycle itself and a postfix following the cycle. We then add the prefix and the cycle, whereby the last transition in the cycle is linked back to the beginning of the cycle. Finally, the postfix then is added to the trace graph.

C. Querying on the Trace Graph

The most important mechanism of the learning algorithm is the membership query, which determines the acceptability of a given behavior. In our case, the behavior of the software and thus of the target automaton is defined by the test cases.

Data: A sequence of signals w
Start at the root node n_0 of the trace graph;
for *all signal in* w **do**
  Get the first signal b in w;
  if *the current node has an outgoing edge marked* b **then**
    Move to the b-successor of n, which is $\delta(n, b)$;
    Remove the first signal from w;
  else
    // The signal is unknown at the current node
    Add w as a new subgraph at the current node;
    return;
  end
end

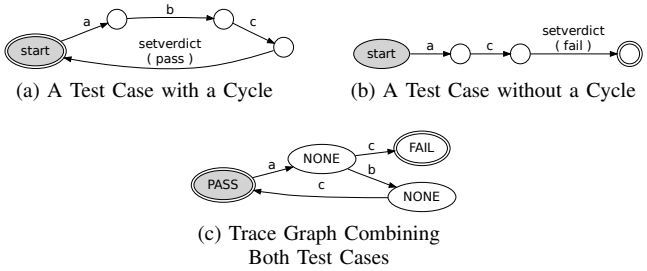
Algorithm 1: Add a Trace to the Trace Graph

Posing a membership query against the trace graph amounts to searching the queried trace in the graph and computing the verdict of the trace. The computation of the verdict has to be adapted to the semantics of the test specification language. In TTCN-3, the policy is that a verdict can only get worse. Accordingly, the overall verdict of a trace is the worst verdict that is stored in the nodes belonging to that trace. In other words, for a trace to be accepted, there must be at least one **pass** verdict and no **fail** verdicts stored in the trace graph for that trace. We rate any trace that is not found on the trace graph as not acceptable and therefore apply the verdict **fail**. The same policy applies to incomplete traces, i.e., the queried trace does not end in a final state or no verdict has been applied during the trace.

The purpose of the equivalence query is to prove that the hypothesis automaton conforms to all test cases in the test suite. This can be regarded as a structural test of the hypothesis automaton against the test suite. Based on the trace graph, the equivalence query is realized as a tree walking algorithm on the trace graph, where the number of cycle expansions is registered and the generated traces are recorded to keep track of interleaved cycles. As short counter-examples provide better results in the learning algorithm, a simple depth-first tree walking algorithm is not sufficient. To extract the shortest possible counter-example in each iteration, we need to use an iterative deepening search. The complexity of such an iterative deepening search depends on the branching factor of the tree to be searched. For the trace graph, the branching depends on the structure of the test suite.

V. Mining the Test Cases

So far, the state-merging in the trace graph only means the combination of the test case automata, where traces are only merged as far as their prefixes match. The trace graph therefore exactly represents the test cases, but nothing more. In the following, we show two techniques to derive additional traces based on our knowledge of test cases.



(a) A Test Case with a Cycle

(b) A Test Case without a Cycle

(c) Trace Graph Combining Both Test Cases

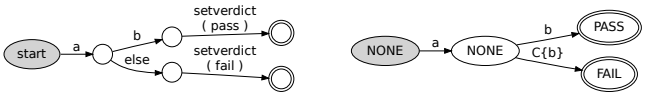
Figure 1. Precedence of Cyclic Behavior

A. Cycles and Non-Cycles

When testing a software system with repetitive behavior or a cyclic structure, the cycle has of course to be tested. However, usually it is sufficient to test the correct working of the cycle in one test case. In all other test cases the shortest possible path through the software is considered, which may mean that test cases execute only a part of a cycle or completely ignore a cycle. Depending on the test purpose, the existence of the cycle might not even be indicated in the test case. As long as the cycle itself is tested by another test case, the test coverage is not influenced. This approach results in shorter test cases, which means shorter execution time and thus faster testing. Furthermore, the readability of the test cases is increased. While the preselection of possible paths for cycles is appropriate for software testing, for machine learning it is desirable to have access to all possible paths of the software.

Consider the two test cases shown in Figures 1a and 1b. Although this is only a small example for demonstration purposes, the setting is quite typical. The test case shown in Figure 1a tests the positive case, that is, a repeated iteration of the three signals *a*, *b*, and *c*. The test case shown in Figure 1b tests for a negative case, namely what happens if the system receives the signal *c* too early. In the latter test case, the repetitive behavior is ignored, as it has been tested before and the test focus is on the error handling of the system. However, usually this behavior could also be observed at any other repetition of the cycle.

For the learning procedure, we would like to have all those possible failing traces, not only the one specified. We therefore define a precedence for cycles, which means that whenever a cycle has the same sequence of signals as a non-cyclic trace, the non-cyclic trace is integrated into the cycle. Figure 1c shows the trace graph combining the two test cases in Figures 1a and 1b. Besides the trace *a, c*, **setverdict(fail)** explicitly specified in Figure 1b, the trace graph also contains traces where the cycle is executed multiple times, *(a, b, c)*, a, c*, **setverdict(fail)**. With precedence of cycles, the test suite used as input to the learning algorithm can be more intuitive, as cycles only need to be specified once.



(a) Test Case (b) Trace Graph

Figure 2. Representing Default Branches

B. Default Behavior

Another common feature of test cases is the concentration on one test purpose. Usually, the main flow of the test purpose forms the test case, while unexpected reactions of the SUT are handled in a general, default way. Still, there may exist a test case that tests (a part of) this default behavior more explicitly.

Default branches usually occur when the focus of the test case is on a specific behavior, and all other possible inputs are ignored or classified as **fail**. Also, sometimes a test case only focuses on a part of the system, where not all possible signals are known. In such cases, the test case often contains a default branch, which classifies what is to be done on reading anything but what was specified.

For our application, this poses two challenges. The first challenge is in the learning procedure. For the different queries, we need to have as many explicitly classified traces as possible, but at the same time we do not want to blow up the size of the test suite. The second challenge is in the construction of the trace graph. When adding all different traces into one combined structure, the implicit context of what is "default" in the local test case is lost. Also, sometimes another test case uses the same default, adds more specific behavior in the range of the default, or defines a new default that slightly differs. We therefore need a method of preserving the local concept of "default" in the test cases and a method of combining different defaults in the trace graph.

Consider a typical default situation, like a **default** statement in a **switch-case** environment. The **default** collects all cases that are not explicitly handled beforehand. As branching on alternatives splits the control flow in a program, each of the branches belongs to a different trace. Therefore, when taking the traces one by one, the context of the default is not clear. To preserve this context, instead of **default** we record the absolute complementary of the set of other alternatives, which is $\complement\{a, b\}$. A *complementary set* is a set that contains everything but the specified elements. Figure 2 shows a test case with defaults (Figure 2a) and its representation as a trace graph using the complementary set notation (Figure 2b). The branch marked with $\complement\{a\}$ represents every branch not marked with a.

Figure 3 shows a trace graph with a default branch in a general way. There are some arbitrary transitions leading to the default (marked with *prefix*), the default branching itself with an edge marked a and an edge marked $\complement\{a\}$

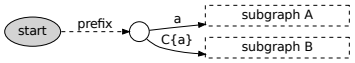
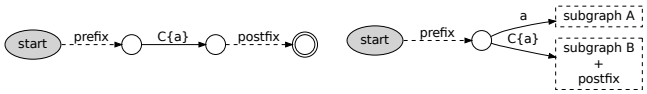


Figure 3. Generic Trace Graph with Default Branch



(a) Test Trace to be Added (b) New Trace Graph

Figure 4. Add a Trace with a Matching Default

("everything but a"), and the arbitrary subgraphs of a and $\complement\{a\}$.

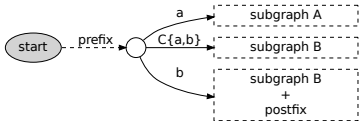
When adding a trace with a matching prefix to this trace graph, the signal s following the prefix can be matched to the trace graph according to one of the following three cases.

- *Exact Match:* s matches one of the branches of the trace graph, i.e., if s is a complementary set, it is identical to the complementary set in the trace graph.
- *Subset:* s matches one signal (or a subset of signals) of the complementary set in the trace graph.
- *Overlap:* s is a complementary set, and overlaps the complementary set in the trace graph.

The first and simplest case is the *exact match*, where a trace with a matching complementary set is added. As the complementary sets are identical, it suffices to add the postfix of the trace to the subgraph of the default already in the trace graph. Figure 4 illustrates this. Figure 4a shows the test trace to be added. The prefix of the trace matches the prefix of the trace graph (see Figure 3) and the complementary set $\complement\{a\}$ matches the complementary set in the trace graph. Therefore, the postfix of the trace has to be added to the subgraph of the complementary set. Assuming that there are no other defaults in the postfix, this is done according to the construction rules described in Section IV-B. Figure 4b depicts the resulting trace graph after the new trace was added.

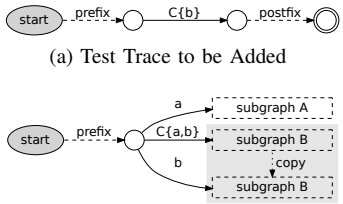
In the second case, the new trace matches a *subset* of

(a) Test Trace to be Added (b) Modify the Trace Graph: Split the Default Branch



(c) New Trace Graph

Figure 5. Add a Trace with a Subset of the Default



(a) Test Trace to be Added

(b) Modify the Trace Graph: Split the Default Branch

(c) Modify the Test Trace: Split the Default Branch

(d) New Trace Graph

Figure 6. Add a Trace with a Differing Default

the complementary set in the trace graph. The situation is depicted in Figure 5, the signal following the prefix in the trace (Figure 5a), *b*, is a subset of the complementary set $\complement\{a\}$. However, the postfix cannot simply be added to the subgraph of the complementary set, as this would allow unspecified traces. Instead, before adding the postfix, the trace graph is modified as shown in Figure 5b. The signal *b* is removed from the complementary set and represented by a distinct edge. Now, the new trace matches exactly and the adding proceeds as described for the first case. Figure 5c shows the result.

In the third and last case, the complementary sets of the new trace and the trace graph *overlap* (see Figure 6). The trace contains an edge marked with the complementary set $\complement\{b\}$ (Figure 6a), whereas the trace graph contains an edge marked with the complementary set $\complement\{b\}$ (see Figure 3). The complementary set of the test trace to be added does not fit the complementary set of the trace graph, but there is an overlap, i.e., every signal which is neither *a* nor *b* matches both sets.

The solution is similar to the second case. The transitions in the trace need to match the transitions in the trace graph, so the sets are split accordingly. For the trace graph, the edge marked b is branched out from the complementary set (Figure 6b). The remaining complementary set in the trace graph is $\complement\{a, b\}$. However, the complementary set of the test trace still does not match, so the test trace is also split (Figure 6c). The complementary sets of the trace graph and the test trace are now identical, $\complement\{a, b\}$, but the test trace has been split into two test traces. Now, the two resulting test traces can be added to the trace graph, resulting in the trace graph shown in Figure 6d.

The described techniques also generalize to sets with more than one element. In this case, the sets associated with the split branches are determined as the intersections and differences of the given sets.

VI. Implementation and Case Study

To assess the power of our learning approach, we have developed a prototypical implementation [23]. The implementation realizes an Angluin-style learner, which is adapted to learning from test cases, and the organization of the test data into a trace graph as discussed in Sections IV and V. Using the prototype, we performed a case study based on the *conference protocol* [24]. The conference protocol describes a chat-box program that can exchange messages with several other chat-boxes over a network.

In the following, we will give a short overview of the prototypical implementation. Subsequently, we describe the experiments that were performed with two versions of the conference protocol. In the last section of this chapter, we will compare the two experiments and draw some conclusions.

A. Prototypical Implementation

Our prototype is implemented in the programming language Java, the abstract structure is shown in Figure 7 as a Unified Modeling Language (UML) class diagramm.

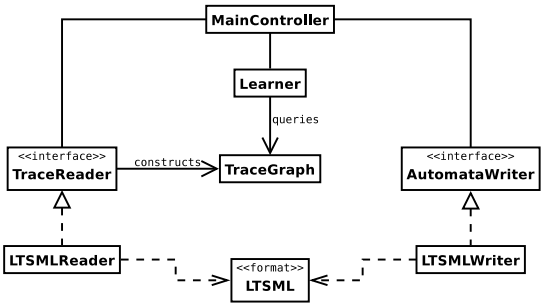


Figure 7. Abstract Structure of the Implementation

The main classes of our prototype implementation are explained further in the following. The class `Learner` implements Angluin's learning algorithm. In every iteration of the learning algorithm, a new counter example is obtained via an equivalence query and used to detect a new state. The discovered states are organized in a classification tree, which is also used to generate the new hypothesis automaton. The two queries, equivalence query and membership query, are implemented according to our adaptation to learning from test cases, and mapped onto a trace graph structure.

In the class `TraceGraph`, the basic methods of semantic state-merging are implemented. A trace graph structure is constructed by adding traces from test cases. The precedence of loops (Section V-A) is currently implemented implicitly,

as loops are simply added first to the trace tree. Default branches (Section V-B) are not yet implemented. In consequence, the currently implemented `TraceGraph` and `Learner` classes are generic and can be used for any test specification language.

For the input of the test cases and the output of the hypothesis automaton, generic interfaces were defined. In our prototype, both interfaces are implemented using the LTSML format that can be used to represent any type of automaton [25]. For the test cases, we focus on the test specification language TTCN-3, i.e., the `TraceReader` recognizes TTCN-3 keywords and generates traces according to the semantics of TTCN-3.

B. The Conference Protocol

Our case study is based on a chat-box program described in [24], the *conference protocol*. We have adapted the protocol to the constraints of our learning procedure.

The conference protocol describes a chat-box program that allows users to participate at a conference chat over a network.

- A user enters an existing conference by sending the service primitive *join*.
- Then, the user can send messages to the conference chat (*datareq*) and receive messages from the conference chat (*dataind*). Each *datareq* causes *dataind* messages to be issued to all other participating users and vice versa.
- At any time after a *join*, the user can leave the conference by sending the service primitive *leave*.

The chat-box program converts each service primitive into a protocol data unit, which is then sent to each of the other participating chat-boxes over a network. Figure 8 illustrates this scenario in a UML interaction diagram.

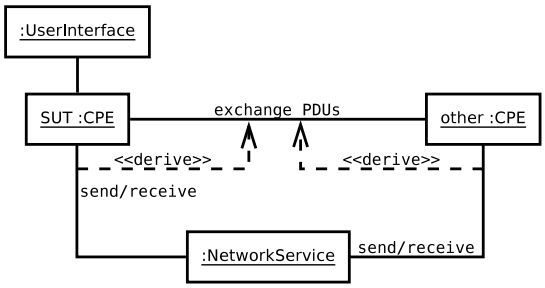


Figure 8. Two CPEs Connected over a Network Service

C. Mining for Cycles

In the first experiment, we want to assess our approach to mining the test cases for cycles. To limit the complexity of the model, we assume that the chat-boxes send messages to each other in a fixed sequence. Based on this assumption, we generate two test suites.

In the first test suite, we build the cases to satisfy a boundary-interior coverage, where the cycles in the data transmission phase of the protocol are executed once or twice, or skipped. The trace graph generated for this test suite contains no cycles. In the second test suite, we explicitly declare cycles, instead of unrolling them.

Table I shows our results for this experiment. The protocol was scaled according to the number of participating chat-boxes. As the table shows, the semantic state-merging of cycles reduces the size of the trace graph by more than half in this example, while the learned automaton was identical. Also, the test suite can be smaller. In addition, the compact version of the trace graph also allows an optimized equivalence query.

D. Limits in Learning Complex Communication

In order to assess the limits of our approach, we extend our version of the conference protocol. We now assume that the network service may mix up the signals, so that the data units are observed in an arbitrary sequence. In our test scenario, we want to accept all traces, where a chat-box correctly joined and left the conference. This means that all participating chat-boxes have reveiced the *join* messages before the first data packages occur, and that no data packages occur after the *leave* message has been sent.

Every service primitive is distributed to $n-1$ other chat-boxes, which means $(n-1)!$ correct paths for joining and leaving the conference and also for sending data. In addition, there are $n-1$ correct paths to receive data. We explicitly specify the data transfer as cyclic behavior. Therefore, we can compute the number of correct traces as $((n-1)!)^3 * (n-1)$, or $(|\text{join}|) \cdot (|\text{send}|) \cdot (|\text{leave}|) \cdot (|\text{receive}|)$, where $|\text{service primitive}|$ denotes the number of correct sequences for the services primitive.

The goal of this experiment is to find out how many test cases are needed to correctly learn the protocol. We tried different approaches to generate test cases for this version of the conference protocol.

A common coverage criterion in testing is the branch coverage, where every branch of the SUT is executed. In application to the conference protocol, this means that we have to cover every serialization of data units. However, it turned out that the learned automata do not correctly represent the intended protocol. A closer look at the model reveals that the learned automaton contains the traces exactly as they were specified in the test cases. Instead of generalizing from the input data, the learning algorithm learned every input trace by heart. This effect could be reproduced with different versions of the test suite. Only by using a test suite satisfying path coverage of the expected automaton, we could learn the correct automaton.

We deduce that the structure of the SUT has an influence on the complexity of the learning process and that for correct machine learning, the test suite has to be as complete as

Number of Chat-Boxes	Size of Target Automaton	Size of the Trace Graph		Size of the Test Suite	
		Without Cycles	With Cycles	Without Cycles	With Cycles
1	72 edges, 8 nodes	33 nodes	13 nodes	6 **pass** traces	2 **pass** traces
2	168 edges, 12 nodes	60 nodes	22 nodes	9 **pass** traces	3 **pass** traces
3	304 edges, 16 nodes	90 nodes	30 nodes	12 **pass** traces	4 **pass** traces
4	480 edges, 20 nodes	120 nodes	40 nodes	15 **pass** traces	5 **pass** traces
5	696 edges, 24 nodes	164 nodes	48 nodes	18 **pass** traces	6 **pass** traces

Table I
EFFECT OF SEMANTIC STATE-MERGING

possible. In fact, this precondition on the learning sample has been described before as the need for a "structurally complete" sample. As a rule of thumb, we might say that "the larger the test suite, the smaller the automaton", as a large test suite usually allows more possible paths.

Experimentation has also shown that the growing size of the test suite affects our learning procedure in two related points. The first is the generation of the trace graph. For larger test cases the preprocessing steps, such as cycle detection, are harder to handle. The second is the equivalence query, where the whole test suite has to be executed again and again. Interestingly, it is not necessary to compute all interleaving cycle executions for the equivalence query. Instead, the crucial points for the equivalence query turn out to be the intersections between different paths through the SUT. Therefore, for a better scalability of our learning procedure, we should aim at detecting intersections of the test cases in the trace graph, thereby minimizing the trace graph and reducing the necessary size of the test suite while keeping its expressiveness.

VII. DISCUSSION AND OPEN QUESTIONS

While our experiments proved the suitability of the learning approach, they also raised a number of questions regarding specific properties of the used data structures and the algorithm. Some of the observations confirmed design decisions, while at other points, decisions turned out to be less than optimal. The following sections provide an assessment of the parts of our learning approach. We reassess the generated automaton, the use of a test suite as input sample, and the learning algorithm itself with respect to their suitability for our purposes. As some of the encountered questions have also attracted the attention of other researchers, there are a number of possible solutions available that could be adapted to our learning approach. In other situations, a number of possible solutions are suggested that have not been investigated yet.

A. Suitability of the Learned Automaton

Angluin's learning algorithm generates a plain DFA, which consists of a set of states partitioned into accepting and rejecting states, an input alphabet triggering state transitions and a corresponding state transition relation. As we have argued in [3], this type of automaton is suitable for representing system models. The drawback of DFAs is that to express the same information as a more advanced model, more states are needed, making the automaton large. Contrary to expectations, it was not the size of the target automaton that proved to be a problem, but its structure and the repercussions on the size of the test suite needed to correctly identify the automaton.

1) Influence of Parameters: A DFA representing a parameterized process such as the conference protocol described in Section VI-B contains a number of paths to cover different serializations of the parameters. The experiments show that to correctly learn all the different serializations, not only all of them need to be represented in the learning sample, but also in every possible combination. This also correlates to the results by Berg et al. regarding prefix-closed models [26], which state that prefix-closed automata are harder to learn than random automata, as the learning algorithm would need to perform a membership query for every prefix.

Berg et al. address this problem by proposing an approach to learn parameterized automata [27]. Based on the original version of Angluin's learning algorithm, which uses an observation table to store the discovered information, they introduce a guard-labeling on the entries of the table, describing an input partitioning. Then, the result of an equivalence query can also be the splitting of a partition beside the discovery of a new state or the acceptance of the learned automaton. A similar approach is described by Li et al. [28], which has the advantage of taking into account both input and output signals, where Berg et al. only consider input signals.

Both proposed solutions for learning parameterized models rely on the original version of Angluin's learning algorithm, which uses an observation table to store the information learned, and the table format is essential in computing the parameterization. In contrast, our approach to learning from test cases uses a variation introduced by Kearns and Vazirani [29], which stores the gathered information in a classification tree. Therefore, an adoption of those solutions requires some adaptations.

2) Handling Non-Applicable Signals: Another structural problem of the learned DFA is that the learning algorithm always generates a fully specified automaton, i.e., an automaton where in every state, a target state for every possible signal is specified. As the essence of state based systems

is that the applicable actions depend on the state of the system, most states of an automaton can only process a subset of the global signal alphabet. The handling of the non-applicable signals then depends on the semantics of the system. One commonly adopted approach is that the system should robustly ignore all unspecified signals, which implies that unspecified transitions at a given state are treated as self-loops.

However, this approach cannot be adopted by Angluin's learning algorithm, as the algorithm only discerns accepted and rejected traces and therefore cannot tell whether a signal is not specified and should be ignored via a self-loop or whether a signal is explicitly rejected and should lead to a fail state. In consequence, the learning algorithm routes all unspecified or rejected signals into a global fail state, thereby generating a fully specified automaton that rejects non-applicable signals.

Due to the properties of the learning algorithm, we can identify the global fail state in the learned automaton, as it is the first rejecting state discovered. Therefore, it would be possible to remove the global fail state and to replace transitions leading into it by self-loops to their source states. This is, however, not a safe transformation, as thereby all explicitly failing transitions would also be transformed into self-loops. In consequence, to learn a DFA that ignores some inopportune signals, those self-loop transitions need to be explicitly specified in the test suite. This obviously leads to a larger test suite, which is also less intuitive and less readable. Alternative approaches would be to distinguish explicitly and implicitly rejected transitions during learning, generating self-loops for implicitly rejected transitions, or to implement a smart transformation algorithm that checks transitions to the global fail state before removing them.

B. Suitability of a Test Suite as Sample Data

The main idea of our learning approach was to use a test suite as input data, as the verdicts **pass** and **fail** readily provided an assessment of acceptable and rejectable system traces. While this assumption holds true, mapping test cases to input traces of Angluin's learning algorithm nevertheless reduces the expressiveness of the test cases considerably. The test cases need to be linearized, a common starting state has to be established, and all circumstantial information as parameters and ports have to be integrated into the input of the target automaton. Especially the flattening of parameters and ports leads to an exponential blowup of the number of test case traces.

While Angluin's algorithm depends on traces, the semantic state-merging approach is designed to exploit the test language specific properties of test cases. The test language specific back-end then represents the sample data in a generic way to the learning algorithm. This way, the learning algorithm provides a common front-end to be combined with different test language specific back-ends. In consequence, further optimization regarding the representation of the test suite mainly concerns the state-merging part of our hybrid algorithm.

1) Mining Additional Properties: The state-merging techniques introduced in Section V define how to merge traces by generating a prefix tree, the representation and handling of cycles in the trace graph and the handling of default branches. However, cycles and defaults are only the most common properties of test languages.

Stable testing states define known and checkable states of the SUT. By marking the according states in the trace graph, a test case containing a marked testing state could be directly connected at the given state. Thereby, the need for a common starting state could be avoided.

Parallel behavior can be explicitly defined, especially in test languages that are targeted at distributed testing. By defining an according operator on the trace tree, membership queries containing different sequentializations of parallel behavior could be answered correctly without explicitly representing every such path in the test suite.

Besides the verdicts **pass** and **fail**, some test languages define additional verdicts assigned on inconclusive behavior or on an error in the test environment. In the current learning approach, everything not accepted, i.e., assigned a **pass** verdict, is rejected. However, in an open world approach, the answer "I don't know" could be given by the membership oracle. In this case, the mapping of the test verdicts has to be reconsidered. The verdict **inconc**, which is used by TTCN-3 to indicate that the result of the test case cannot be decided, maps naturally on an "I don't know" for the learner—the teacher does not know whether the behavior is acceptable or not. Then again, the verdict **error** is considered by TTCN-3 to be more severe than a verdict **fail**, but for learning purposes it could still amount to an "I don't know".

Lastly, information about *ports* in the test cases could also be used. Considering a highly connectable SUT, such a system would feature a number of different ports connecting to different other systems. To learn a protocol automaton for just a subset of those communication ports, the test case traces could be restricted to the ports in question, excluding all others.

2) Influence of Coverage: While the semantic state-merging approach is able to make up for many missing traces, the case study suggests that the test suite used in learning must at least satisfy a path coverage of the SUT, as the one experiment where only a branch coverage was used failed. However, there are other coverage criteria besides branch and path coverage, e.g., based on condition determination or on the functions of the SUT, or automatic test case generation techniques. Further research is needed to clarify the dependencies of system structure, test suite coverage, and learnability.

C. Suitability of the Learning Algorithm

When confronted with the problem of reconstructing a system model from test cases, learning algorithms seemed to be a simple solution. The starting assumption was that while the test cases could be used as they were, some adaptations would have to be made to the algorithm. Instead, research shows that the learning algorithm itself can be used without changes, while the effort of adaptation concerns the representation of the learning sample, i.e., the test cases. In fact, the approach to learning from test cases proved to be a problem of teaching more than of learning.

1) Online and Offline Learning: Online learning algorithms, like Angluin's algorithm, build a system model by querying a teacher. Their main advantage is in generating the necessary queries themselves, thereby avoiding the need for a complete sample. However, this approach implies the existence of an omniscient oracle, which is able to answer arbitrary queries. In contrast, offline learning algorithms assume the existence of a structurally complete sample, which is then merged into the target automaton.

The query mechanisms used by Angluin's algorithm naturally match the test cases' verdicts. Also, Angluin's algorithm is scalable, growing only linearly with the size of the target automaton, and always generates a minimal DFA. As a test suite always assumes completeness with regard to certain coverage criteria, it can be assumed that the completeness of the test suite is sufficient to answer the membership queries. However, our experiments show that this assumption holds only for high coverage criteria and even then depends on the structure of the system.

These results seem to suggest that an offline learning approach would work better for the learning from test cases. Lambeau et al. [30] propose an offline algorithm based on state-merging, which takes into account merge constraints as well as incompatibility constraints, requiring some states to be merged obligatorily while others need to stay separated. This approach might work well for the learning from test cases. However, state-merging algorithms always need to be closely tailored to the sample data to merge. Therefore, a state-merging approach would only work for the special semantics it is designed for.

Our approach combines the advantages of both online and offline algorithms. The online algorithm is used to drive the overall learning process, thereby establishing a learning procedure that is independent from any given test specification language. Underlying the learning process, state-merging is used to mine the test language specific information for better coverage of the automaton's traces and to generate a data structure to be used as an oracle.

Following this layered approach, existing methods could be integrated for optimization. Regarding the online learning part, these optimizations concern the type of automaton generated, incorporating i.e., parameterization [27], [28]. Optimizations on the offline learning part should extend the semantic state-merging approach. Possibly exploitable properties of test cases comprise differentiation of input and output signals and consideration of stable testing states. For example, the stable testing states could be matched to the merge constraints in the approach by Lambeau et al. [30].

2) Other Versions of Angluin's Algorithm: Another source for optimization of the learning procedure is the version of Angluin's algorithm that is used. The prototypical implementation uses a variation introduced by Kearns and Vazirani [29], which stores the gathered information in a classification tree. This version has the advantage of asking less membership queries, but at the cost of more equivalence queries [31]. Also, the classification tree provides a structure that is easy to maintain and therefore quickly to implement.

The original version of Angluin's algorithm uses an observation table to store the gathered information. This variation asks more membership queries before constructing the first hypothesis automaton, thereby reducing the number of needed equivalence queries [31]. On the other hand, maintaining the consistency of the observation table needs more effort.

Experimentation shows that using the trace graph as an oracle, membership queries are cheap, as their complexity only depends on the length of the queried trace. Equivalence queries take time, as in the worst case, the whole test suite has to be run against the hypothesis automaton. Also, the adaptations to learning from test cases are completely independent of the underlying implementation of Angluin's algorithm. Therefore, re-implementing the core of the learning algorithm according to the original version of Angluin's algorithm might even provide a small performance advantage.

3) Breaking the Closed World Assumption: In most learning scenarios, it is comparatively easy to get a correct answer to membership queries, while the equivalence query is hard to decide. When learning from a complete test suite, it is the other way around. The equivalence query can be matched easily to a run of the test suite against the hypothesis automaton, the only limiting factor being the time needed to run a large test suite. This also relates to the results of Berg et al. [32], who investigate the similarities between conformance testing and automata inference, finding that conformance testing solves a checking problem. Therefore, we can safely assume that a test suite that is sufficient to declare a system as conforming to its specification also suffices to decide whether it is equivalent to a learned hypothesis automaton.

In contrast, when asking membership queries against a limited set of traces, as every test suite is bound to be, there will always be queried traces that are not contained in the test suite. As the experiments show, rejecting every unknown trace can lead to bad generalization in the hypothesis automaton, while trying to provide for every possible query leads to inhibitively large test suites. There are several

possible approaches to solve this dilemma.

One way is to mine the test suite for implicit traces by state-merging. First efforts in this direction have been integrated into our learning approach and have shown positive results. Besides further exploitation of the properties of the test languages, also input from existing research in state-merging techniques can be used. The state-merging approach has two main advantages. The approach is self-contained, as no external input is needed, and it is safe, as the state-merging is based on information internal to the test suite. The drawback is that the mining depends on the information available in the test suite. If a test language with restricted description possibilities is used, the possibilities of state-merging are also restricted. Besides, while state-merging is able to boost the number of covered traces, it cannot make up for missing test cases if the test suites coverage is insufficient.

Another possible approach is to include explicit "don't know" into the possible answers of a membership query. The problem of undecidable membership queries has occurred to researchers in other settings before, therefore a number of possibly adaptable methods exist. Sloan and Turán [33] define a meta-algorithm for incomplete membership oracles. For each undecidable membership query, the learning process is forked, one instance assuming acceptance, the other rejection of the queried string. If a copy is detected to be inconsistent, it is pruned. While this approach clearly leads to an exponential growth in the computation, the difficulty also is how to determine inconsistencies in the forked hypotheses. Bshouty and Owshanko [34] propose an extension of Angluin's learning algorithm including "don't know" as a possible answer to a membership query. Based on the Kearns-Vazirani version of the algorithm, they partition the possible traces of the target automaton into cover sets, resetting the algorithm when a counter-example causes a cover set to be split. Grinchtein and Leucker [35] also suggest an extension of Angluin's algorithm. Using Angluin's original version, they generate an incomplete observation table that they subsequently feed into a satisfiability solver, filling in the gaps with the most consistent solution. However, all those approaches share the disadvantage of replacing the uncertainties of the membership oracle with assumptions, thereby deviating from exact learning.

The third approach is a combination of passive and active techniques. In this approach, the algorithm learns as much as possible using the available information, fully exploiting every counter-example. When an unanswerable query is encountered, the query is either addressed at an external oracle, e.g., a domain expert, or translated into a test case that is executed against the SUT. Asking a domain expert leads to a guided learning approach. In this case, the learning is only semi-automatic. Executing a test case against an SUT could be conducted automatically. However, as the outcome of the query then would depend on the SUT, this approach compromises the independence of the learned automaton. As both approaches draw information from sources beside the test suite, inconsistencies could be introduced into the learning data.

VIII. Conclusion

We have presented a learning approach that combines state-merging and learning techniques to generate a DFA from a test suite. The state-merging is used to represent the test suite and to find additional test cases exploiting the semantic properties of the test language. The combined approach has been implemented in a prototypical tool. Experiments show that while the state-merging approach reduces the size of the test suite needed for correct identification of the model, complex models still need a large number of test cases for correct identification.

We have discussed the design decisions that form the basis of our approach to learning from test. The main issue is the size and coverage of the test suite used in the learning process. While the mapping of test cases to learning traces is intuitive and simple, the size of a test suite sufficient for learning can get inhibitively large. Optimizations to deal with this problem comprise the extension of the semantic state-merging approach to better exploit the information contained in the test cases and an extension of the learning algorithm to work with unanswerable membership queries. In addition, the relation between test suite coverage, system structure, and learnability offers interesting research topics.

Based on the experiments with our learning approach, the next step is to incorporate the identified optimizations into our prototypical implementation. In the long run, our findings on the learnability of different models could also be used to assess the adequacy of a test suite.

References

[1] E. Werner and J. Grabowski, "Model Reconstruction: Mining Test Cases," in *Third International Conference on Advances in System Testing and Validation Lifecycle (VALID 2011).*, Oct. 2011.

[2] D. Angluin, "Learning Regular Sets from Queries and Counterexamples," *Information and Computation*, vol. 75, no. 2, pp. 87–106, 1987.

[3] E. Werner, S. Polonski, and J. Grabowski, "Using Learning Techniques to Generate System Models for Online Testing," in *Proc. INFORMATIK 2008*, ser. LNI, vol. 133. Köllen Verlag, 2008, pp. 183–186.

[4] M. Shahbaz and R. Groz, "Inferring Mealy Machines," in *Proc. FM 2009*, ser. LNCS, vol. 5850. Springer, 2009, pp. 207–222.

[5] F. Aarts, B. Jonsson, and J. Uijen, "Generating Models of Infinite-State Communication Protocols Using Regular Inference with Abstraction," in *Proc. ICTSS'10*, ser. LNCS, vol. 6435. Springer, 2010, pp. 188–204.

[6] T. Berg, B. Jonsson, and H. Raffelt, "Regular Inference for State Machines Using Domains with Equality Tests," in *Proc. FASE 2008*, ser. LNCS, vol. 4961. Springer, 2008, pp. 317–331.

[7] T. Bohlin, B. Jonsson, and S. Soleimanifard, "Inferring Compact Models of Communication Protocol Entities," in *proc. ISoLA 2010*, ser. LNCS, vol. 6415. Springer, 2010, pp. 658–672.

[8] M. Shahbaz, K. Li, and R. Groz, "Learning and Integration of Parameterized Components Through Testing," in *Proc. TestCom 2007*, ser. LNCS, vol. 4581. Springer, 2007, pp. 319–334.

[9] J. Esparza, M. Leucker, and M. Schlund, "Learning Workflow Petri Nets," in *Proc. PETRI NETS 2010*, ser. LNCS, vol. 6128. Springer, 2010, pp. 206–225.

[10] B. Bollig, J.-P. Katoen, C. Kern, and M. Leucker, "SMA — The Smyle Modeling Approach," *Computing and Informatics*, vol. 29, no. 1, pp. 45–72, 2010.

[11] A. W. Biermann and R. Krishnaswamy, "Constructing Programs from Example Computations," *IEEE Transactions on Software Engineering*, vol. 2, no. 3, pp. 141–153, 1976.

[12] K. J. Lang, B. A. Pearlmutter, and R. A. Price, "Results of the Abbadingo One DFA Learning Competition and a New Evidence-Driven State Merging Algorithm," in *Proc. ICGI-98*, ser. LNCS, vol. 1433. Springer, 1998, pp. 1–12.

[13] W. M. P. van der Aalst, T. Weijters, and L. Maruster, "Workflow Mining: Discovering Process Models from Event Logs," *IEEE Transactions on Knowledge and Data Engineering*, vol. 16, no. 9, pp. 1128–1142, 2004.

[14] J. E. Cook and A. L. Wolf, "Discovering models of software processes from event-based data," *TOSEM*, vol. 7, no. 3, pp. 215–249, 1998.

[15] K. Koskimies and E. Mäkinen, "Automatic synthesis of state machines from trace diagrams," *Software—Practice & Experience*, vol. 24, no. 7, pp. 643–658, 1994.

[16] I. H. Krüger and R. Mathew, "Component Synthesis from Service Specifications," in *Revised Selected Papers of the International Workshop on Scenarios: Models, Transformations and Tools*, ser. LNCS, vol. 3466. Springer, 2003, pp. 255–277.

[17] A. Bertolino, P. Inverardi, P. Pelliccione, and M. Tivoli, "Automatic Synthesis of Behavior Protocols for Composable Web-Services," in *Proc. ESEC/SIGSOFT FSE*. ACM, 2009, pp. 141–150.

[18] C. Lee, F. Chen, and G. Rosu, "Mining parametric specifications," in *Proc. ICSE 2011*. ACM, 2011, pp. 591–600.

[19] G. Ammons, R. Bodik, and J. R. Larus, "Mining Specifications," in *Proc. POPL'02*. ACM, 2002, pp. 4–16.

[20] L. M. Duarte, J. Kramer, and S. Uchitel, "Model Extraction Using Context Information," in *Proc. MoDELS 2006*, ser. LNCS, vol. 4199. Springer, 2006, pp. 380–394.

[21] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic Generation of Software Behavioral Models," in *Proc. ICSE 2008*. ACM, 2008, pp. 501–510.

[22] *ETSI Standard (ES) 201 873: The Testing and Test Control Notation version 3; Parts 1–10*, ETSI Std., Rev. 4.2.1, 2010.

[23] E. Werner, "Learning Finite State Machine Specifications from Test Cases," Ph.D. dissertation, Georg-August-Universität Göttingen, Göttingen, Jun. 2010. [Online]. Available: http://webdoc.sub.gwdg.de/diss/2010/werner/

[24] L. D. Bousquet, S. Ramangalahy, S. Simon, C. Viho, A. F. E. Belinfante, and R. G. Vries, "Formal Test Automation: The Conference protocol with TGV/Torx," in *Proc. TestCom 2000*, ser. IFIP Conference Proceedings. Kluwer Academic Publishers, 2000, pp. 221–228.

[25] D. Neumann, "Test Case Generation using Model Transformations," Master's Thesis, University of Göttingen, Institute for Computer Science, Göttingen, Germany, 2009.

[26] T. Berg, B. Jonsson, M. Leucker, and M. Saksena, "Insights to Angluin's Learning," *ENTCS*, vol. 118, pp. 3–18, 2005.

[27] T. Berg, B. Jonsson, and H. Raffelt, "Regular Inference for State Machines with Parameters," in *Proc. FASE 2006*, ser. LNCS, vol. 3922. Springer, 2006, pp. 107–121.

[28] K. Li, R. Groz, and M. Shahbaz, "Integration Testing of Distributed Components Based on Learning Parameterized I/O Models," in *Proc. FORTE 2006*, ser. LNCS, vol. 4229. Springer, 2006, pp. 436–450.

[29] M. J. Kearns and U. V. Vazirani, *An Introduction to Computational Learning Theory*. MIT Press, 1994.

[30] B. Lambeau, C. Damas, and P. Dupont, "State-Merging DFA Induction Algorithms with Mandatory Merge Constraints," in *Proc. ICGI 2008*, ser. LNCS, vol. 5278. Springer, 2008, pp. 139–153.

[31] J. L. Balcázar, J. Díaz, R. Gavaldà, and O. Watanabe, "Algorithms for Learning Finite Automata from Queries: A Unified View," in *Advances in Algorithms, Languages, and Complexity*, D.-Z. Du and K.-I. Ko, Eds. Kluwer Academic Publishers, 1997, pp. 53–72.

[32] T. Berg, O. Grinchtein, B. Jonsson, M. Leucker, H. Raffelt, and B. Steffen, "On the Correspondence Between Conformance Testing and Regular Inference," in *Proc. FASE 2005*, ser. LNCS, vol. 3442. Springer, 2005, pp. 175–189.

[33] R. H. Sloan and G. Turán, "Learning with queries but incomplete information (extended abstract)," in *Proc. COLT '94*. ACM, 1994, pp. 237–245.

[34] N. H. Bshouty and A. Owshanko, "Learning Regular Sets with an Incomplete Membership Oracle," in *Proc. COLT 2001 and EuroCOLT 2001*, ser. LNCS, vol. 2111. Springer, 2001, pp. 574–588.

[35] O. Grinchtein and M. Leucker, "Learning Finite-State Machines from Inexperienced Teachers," in *Proc. ICGI 2006*, ser. LNCS, vol. 4201. Springer, 2006, pp. 344–345.