Model Reconstruction: Mining Test Cases

Edith Werner and Jens Grabowski
Software Engineering for Distributed Systems Group
Institute for Computer Science,
University of Göttingen, Göttingen, Germany
$\{$*ewerner*$|$*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. 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. 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 is influenced.

The basis of our approach is a learning algorithm, first introduced by Angluin [1], which learns a Deterministic Finite Automaton (DFA). To learn from test cases, we adapted the query mechanisms of the algorithm [2]. 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.

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. In Section VII, we conclude with a summary and an outlook.

II. Related Work

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 [3] and parameterized models [4], [5], [6], [7]. Some approaches can handle large or even infinite message alphabets [4] or potentially infinite state spaces [5]. 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 [8] learns workflow petri nets from event logs and handles incomplete data by asking an external teacher. In [9], 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, our aim is to generate 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.

III. Foundations

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

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) [10] 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 [1]. 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 for 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., $rejected \equiv \neg accepted$. In consequence, 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 [2], [11].

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, in general, 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 and 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.

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

Algorithm 1: Add a Trace to the Trace Graph

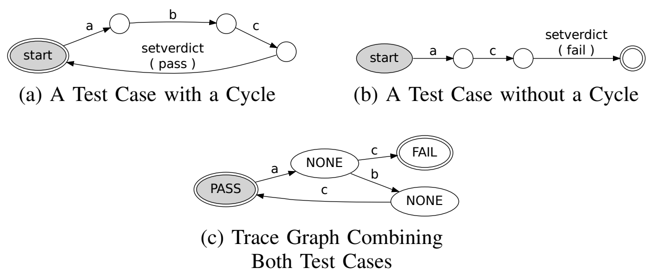
Cycles of the test case automaton need special treatment, as a cycle means that an edge loops back to an existing node. To this end, we separate the cyclic traces into three parts, a prefix leading into the cycle, the cycle itself and a postfix following the cycle. We then add the prefix and the cycle, 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.

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. Cycles and Non-Cycles

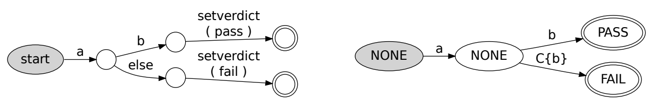
When testing a software system with repetitive behavior or a cyclic structure, the cycle has of course to be tested. However, usually it is sufficient to test the correct working of the cycle in one test case. In all other test cases the shortest possible path through the software is considered, which 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.



(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) Test Case (b) Trace Graph

Figure 2. Representing Default Branches

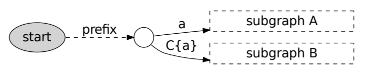


Figure 3. Generic Trace Graph with Default Branch

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.

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 which 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.

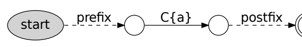
Consider a typical default situation, like a **default** statement in a **switch-case** environment. The **default** collects all cases that are not explicitly handled beforehand. As branching on alternatives splits the control flow in a program, each of the branches belongs to a different trace. Therefore, 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 $\complement\{a\}$ ("everything but a"), and the arbitrary subgraphs of *a* and $\complement\{a\}$.

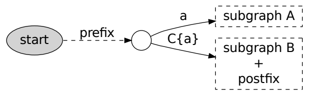
When adding a trace with a matching prefix to this trace graph, the signal s following the prefix can be matched to the trace graph according to one of the following three cases.

- *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

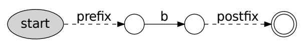


(a) Test Trace to be Added

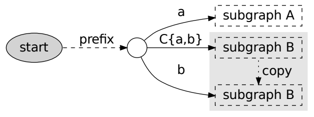


(b) New Trace Graph

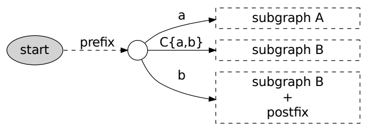
Figure 4. Add a Trace with a Matching Default



(a) Test Trace to be Added



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



(c) New Trace Graph

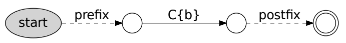
Figure 5. Add a Trace with a Subset of the Default

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.

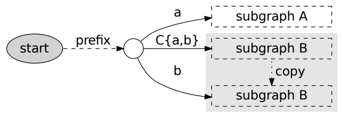
In the second case, the new trace matches a *subset* of the complementary set in the trace graph. The situation is depicted in Figure 5, the signal following the prefix in the trace (Figure 5a), *b*, is a subset of the complementary set $\complement\{a\}$. However, the postfix cannot simply be added 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.

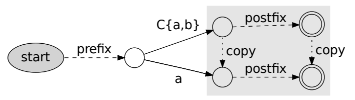
The solution is similar to the second case. The transitions



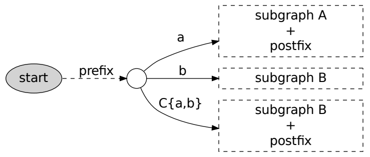
(a) Test Trace to be Added



(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

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. Experimental Results

To assess the power of our learning approach, we have developed a prototypical implementation [11]. 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* [12]. The conference protocol describes a chat-box program that can exchange messages with several other chat-boxes over a network.

Table I shows our results for a simple version of the conference protocol, where the sequence of the signals was fixed. The protocol scaled according to the number of participating chat-boxes. As the table shows, the semantic state-merging reduces the size of the trace graph by more

Number of	Size of	Size of the Trace Graph		Size of the Test Suite	
Chat-Boxes	Target Automaton	Without Merging	With Merging	Without Merging	With Merging
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

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.

Additional experiments show that while our approach quickly learns clearly structured models, models with a high degree of variation are hard to learn and require a large test suite. In fact, for a more complex version of the conference protocol with variable signal sequence, the model could only be reconstructed from a test suite satisfying path coverage [11]. However, it is possible that the size of the test suite can be further reduced using additional state-merging rules, e.g., marked stable testing states.

VII. 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.

Optimizations to deal with this problem comprise the extension of the semantic state-merging approach to exploit further 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] D. Angluin, "Learning Regular Sets from Queries and Counterexamples," *Information and Computation*, vol. 75, no. 2, pp. 87–106, 1987.

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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/

[12] 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.