Learning and Testing the Bounded Retransmission Protocol[*]

Fides Aarts F.AARTS@CS.RU.NL
Harco Kuppens H.KUPPENS@CS.RU.NL
Jan Tretmans TRETMANS@CS.RU.NL
Frits Vaandrager F.VAANDRAGER@CS.RU.NL
Sicco Verwer SICCOVERWER@GMAIL.COM
Institute for Computing and Information Sciences, Radboud University Nijmegen
P.O. Box 9010, 6500 GL Nijmegen, the Netherlands

Editors: Jeffrey Heinz, Colin de la Higuera and Tim Oates

Abstract

Using a well-known industrial case study from the verification literature, the bounded retransmission protocol, we show how active learning can be used to establish the correctness of protocol implementation I relative to a given reference implementation R. Using active learning, we learn a model M_R of reference implementation R, which serves as input for a model based testing tool that checks conformance of implementation I to M_R. In addition, we also explore an alternative approach in which we learn a model M_I of implementation I, which is compared to model M_R using an equivalence checker. Our work uses a unique combination of software tools for model construction (Uppaal), active learning (LearnLib, Tomte), model-based testing (JTorX, TorXakis) and verification (CADP, MRMC). We show how these tools can be used for learning these models, analyzing the obtained results, and improving the learning performance.

1. Introduction

The behavior of software systems can often be specified using finite state machine models. These models provide an overview of software systems, by describing the way in which they react to different inputs, and when they produce which output. Unfortunately, the construction of finite state machine descriptions is often omitted during software development (Walkinshaw et al., 2010). An alternative to constructing these models manually, is to use *software model synthesis* (or system identification/learning, or process discovery/mining) tools in order to derive them automatically from data (Cook and Wolf, 1998). This data typically consists of *execution traces*, i.e., sequences of operations, function calls, user interactions, or protocol primitives, which are produced by the system or its surrounding environment. Intuitively, software model synthesis tries to discover the logical structure (or model) underlying these sequences of events. This can be seen as a grammatical inference problem in which the events are modeled as the symbols of a language, and the goal is to find a model for this language.

[*] SV was supported by the STW project 11763 Integrating Testing And Learning of Interface Automata (ITALIA).

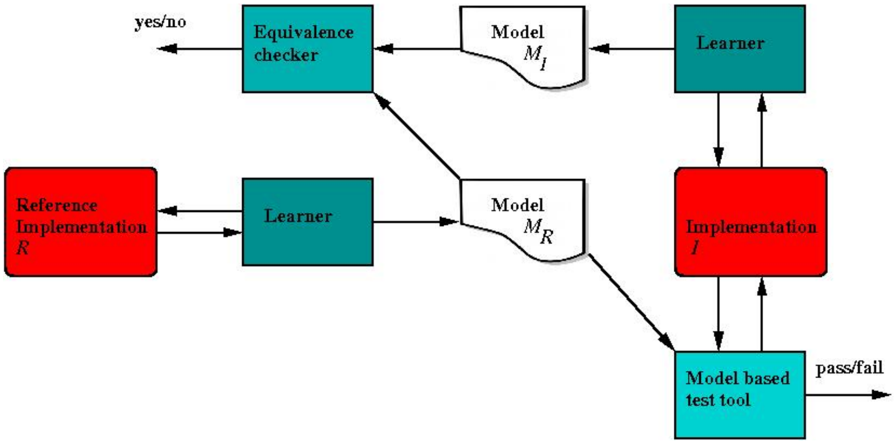

Figure 1: Use of automata learning to establish conformance of implementations. The learners interact with the implementations in order to construct models, which are then subsequently used for model-based testing and equivalence checking.

The model we use in this paper is a Mealy machine, which is a deterministic finite state automaton (DFA) variant with alternating input and output symbols, see, e.g., (Sudkamp, 2006). The Mealy machine model is popular for specifying the behavior of reactive systems and communication protocols. DFAs and Mealy machines are simple models and in some cases they will not be able to represent or identify all the complex behaviors of a software system. Some more powerful models with identification algorithms include: non-deterministic automata (Yokomori, 1993; Denis et al., 2000), probabilistic automata (Clark and Thollard, 2004; Castro and Gavaldà, 2008), Petri-nets (van der Aalst, 2011), timed automata (Verwer, 2010; Grinchtein et al., 2006), I/O automata (Aarts and Vaandrager, 2010), and Büchi automata (Higuera and Janodet, 2004). Despite their limited power, DFA and Mealy machine learning methods have recently been used to learn different types of complex systems such as web-services (Bertolino et al., 2009), X11 windowing programs (Ammons et al., 2002), network protocols (Cui et al., 2007; Antunes et al., 2011; Comparetti et al., 2009), and java programs (Walkinshaw et al., 2007; Dallmeier et al., 2006; Mariani et al., 2011).

In this paper, we make use of Mealy machine learning methods in order to establish the correctness of protocol implementations relative to a given reference implementation (which we assume to be correct). In software engineering, a reference implementation is, in general, an implementation of a specification to be used as a definite interpretation for that specification: as a standard against which all other implementations are measured. A reference implementation is usually developed concurrently with the specification and the software test suite. In addition to serving as a reference for future implementations, it helps to discover errors and ambiguities in a software specification, and demonstrates that the specification is actually implementable. To the best of our knowledge, this is a novel application area of grammatical inference. Moreover, it is a promising one since reference implementations are in existance for many real-world software systems.

Figure 1 illustrates how we may use model synthesis for establishing conformance of protocol implementations relative to a given reference implementation. Using a state machine synthesis tool, we first actively (query) learn a state machine model M_R of the reference implementation R (using, e.g., (Raffelt et al., 2009)). Now, given another implementation I, there are basically two things we can do. The first approach is that we provide M_R as input to a model based testing tool (see, e.g., (Belinfante, 2010)). This tool will then use M_R to generate test sequences and apply them to implementation I in order to establish the conformance of I to the learned model M_R, i.e., whether they implement the same behavior. The model based testing tool will either output "pass", meaning that the tool has not been able to find any deviating behaviors, or it will output "fail" together with an execution trace that demonstrates the difference between I and M_R. The second, more ambitious approach, is to use the learning tool to learn a model M_I of the other implementation I, and then use an equivalence checker (see, e.g., (Garavel et al., 2011)) to check the behavioral equivalence of M_R and M_I. The equivalence checker will either output "yes", meaning that the two models are equivalent, or "no" together with a trace that demonstrates the difference between the two models. In the latter case, we check whether this trace also demonstrates a difference between the corresponding implementations R and I. If not, we have obtained a counterexample for one of the two models, which we may feed to the learner in order to obtain a more refined model M_R of R or M_I of I.

We investigate the feasibility of the above approach using a well-known benchmark case study from the verification literature: the bounded retransmission protocol (Helmink et al., 1994; D'Argenio et al., 1997) (see Section 2). The bounded retransmission protocol is a variation of the classical alternating bit protocol (Bartlett et al., 1969) that was developed by Philips Research to support infrared communication between a remote control and a television. We constructed an implementation of the protocol, to which we refer as the reference implementation, and 6 other faulty variations of the reference implementation. Our aim is to combine active Mealy machine learning methods with model-based testing (see Section 3) in order to quickly discover the behavioral differences between these variations and the reference. To this aim, we make use of several state-of-the-art tools from grammatical inference, software testing, and formal verification (see Section 4). We show how these tools can be used for learning models of the bounded retransmission protocol (Section 5), analyzing the obtained results, and improving the performance of the learning methods (Section 6). Our main contribution is demonstrating how active learning can be used in an industrial setting by combining it with software verification and testing tools, and showing how these tools can also be used to analyze and improve the results of learning.

2. The BRP Implementation and Its Mutants

The bounded retransmission protocol (BRP) (Helmink et al., 1994; D'Argenio et al., 1997) is a variation of the well-known alternating bit protocol (Bartlett et al., 1969) that was developed by Philips Research to support infrared communication between a remote control and a television. In this section, we briefly recall the operation of the protocol, and describe the reference implementation and the 6 mutant implementations.

The bounded retransmission protocol is a data link protocol which uses a stop-and-wait approach known as 'positive acknowledgement with retransmission': after transmission of

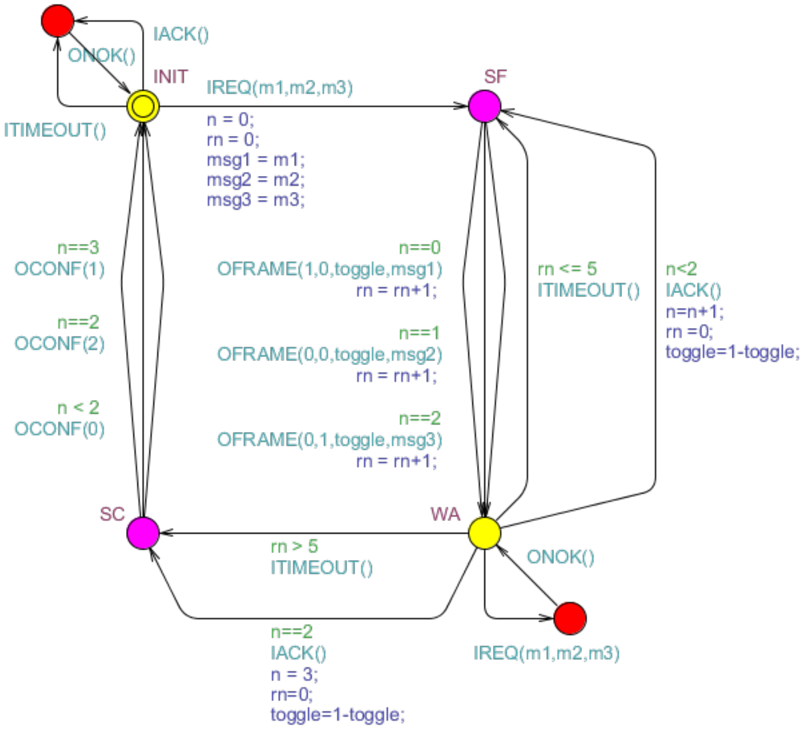

Figure 2: Reference implementation of the BRP sender. The input symbols start with I, the output symbols start with O. In addition to symbols, the transitions contain value checks (or guards, $==$, $<$, $>$) and assignments ($=$).

a frame the sender waits for an acknowledgement before sending a new frame. For each received frame the protocol generates an acknowledgement. If, after sending a frame, an acknowledgement fails to appear, the sender times out and retransmits the frame. An alternating bit is used to detect duplicate transmission of a frame.

Figure 2 illustrates the operation of our reference implementation of the sender of the BRP. Actually, the reference implementation that we used is a Java executable that was generated automatically from this diagram (represented as a Uppaal xml file, see Section 4). The sender protocol uses the following inputs and outputs:

- Via an input IREQ(m_1, m_2, m_3), the upper layer requests the sender to transmit a sequence m_1 m_2 m_3 of messages. For simplicity, our reference implementation only allows sequences of three messages, and the only messages allowed are 0 and 1. When the sender is in its initial state INIT, an input IREQ(m_1, m_2, m_3) triggers an output OFRAME(b_1, b_2, b_3, m), otherwise it triggers output ONOK.

- Via an output OFRAME(b_1, b_2, b_3, m), the sender may transmit a message to the receiver. Here m is the actual transmitted message, b_1 is a bit that is 1 iff m is the first message in the sequence, b_2 is a bit that is 1 iff m is the last message in the sequence, and b_3 is the alternating bit used to distinguish new frames from retransmissions.

- Via input IACK the receiver acknowledges receipt of a frame and via input ITIMEOUT the sender is informed that a timeout has occurred, due to the loss of either a frame or an acknowledgement message. When the sender is in state WA ("wait for acknowledgement"), an input IACK or ITIMEOUT triggers either an output OFRAME(b_1, b_2, b_3, m) or an output OCONF(i). If the sender is not in state WA, ONOK is triggered.

- Via an output OCONF(i), the sender informs the upper layer about the way in which a request was handled:

  - $i = 1$: the request has been dispatched successfully,
  - $i = 0$: the request has not been dispatched completely,
  - $i = 2$: the request may or may not have been handled completely; this situation occurs when the last frame is sent but not acknowledged.

  An output OCONF occurs when either all three messages have been transmitted successfully, or when a timeout occurs after the maximal number of retransmissions.

Note that, within the state machine of Figure 2, inputs and outputs strictly alternate. Thus it behaves like a Mealy machine. The state machine maintains variables msg1, msg2 and msg3 to record the three messages in the sequence, a Boolean variable toggle to record the alternating bit, an integer variable n to record the number of messages in the sequence that have been acknowledged, and an integer variable rn that records the number of times a message has been retransmitted. Each message is retransmitted at most 5 times.

We consider the following six mutants of the reference implementation of the sender:

1. Whereas the reference implementation only accepts a new request in the INIT state, mutant 1 also accepts new requests in state WA. Whenever mutant 1 receives a new request, the previous request is discarded and the sender starts handling the new one.

2. Whereas in the reference implementation each message is retransmitted at most 5 times, mutant 2 retransmits at most 4 times.

3. Whereas in the reference implementation the alternating bit is only toggled upon receipt of an acknowledgement, mutant 3 also toggles the alternating bit when a timeout occurs.

4. In mutant 4 the first and last control bit for the last message are swapped.

5. Mutant 5 outputs an OCONF(0) in situations where the reference implementation outputs OCONF(2).

6. If the first and the second message are equal then mutant 6 does not transmit the third message, but instead retransmits the first message.

Since input and output messages still alternate, all of these BRP mutants still behave as Mealy machines. For all BRP implementations, we consider the inputs: IREQ(m_1, m_2, m_3), IACK, and ITIMEOUT, where m_1, m_2, and m_3 can be either 0 or 1. Thus, the input alphabet consists of 10 input symbols: 8 different IREQ inputs, one IACK input, and one ITIMEOUT input. We also have the following outputs: ONOK, OFRAME(b_1, b_2, b_3, m), and OCONF(i), where $0 \leq i \leq 2$, i.e., 20 output symbols. In the next section, we discuss how to connect these implementations to an active Mealy machine learner and a model-based testing tool.

3. Active learning and model-based testing

Active learning of software systems Active learning or query learning is a learning setting in which the learner can ask questions (queries) to a teacher (an oracle). In our case, this teacher is a black-box (proprietary) software system that we would like to analyze, for instance in order to determine whether its implementation is compatible with our own. By providing this software system with inputs, and reading the generated outputs, we can try to determine (reverse engineer) its inner workings. With some modifications (Margaria et al., 2004), we can apply the well-known L^* DFA learning algorithm (Angluin, 1987) to this data in order to learn a Mealy machine model.

However, since black-box software systems are unable to answer equivalence queries, we *approximate* them using randomly generated membership queries (Angluin, 1987). Thus, for every equivalence query, we generate many strings, for each we ask a membership query, and if an output is different from the output generated by the model, we return this string as a counterexample. If no such string is found, we have some confidence that the model is correct and the more strings we test, the higher this confidence.

Model Based Testing Systematic testing of software plays an important role in the quest for improved software quality. Testing, however, turns out to be an error-prone, expensive, and time-consuming process. *Model-based testing* (MBT) is one of the promising technologies to meet the challenges imposed on software testing (Utting and Legeard, 2007). When using MBT, a *system under test* (SUT) is tested against a formal description, or a *model*, of the SUT's behavior. Such a model serves as a precise and complete specification of what the SUT should do, and, consequently, is a good basis for the algorithmic generation of test cases. MBT makes testing faster and less error-prone: millions of test events can be automatically generated from the model, and subsequently executed against the SUT, after which the results can also be automatically analyzed. Due to the inherent limitations of testing, such as the limited number of tests that can be performed in the available time, testing can never be complete: testing can only show the presence of errors, not their absence. Yet, MBT is a rigorous method for providing confidence, though no certainty, that the behavior of the SUT complies with its model.

MBT approaches differ in the kind of models that they use, e.g., state-based models, pre- and post-conditions, timed automata or functional programs, and in the algorithms they use to generate test cases. In this paper we concentrate on two state-based methods: Mealy Machines and labeled transition systems (LTS). The testing approach using Mealy Machines assumes that the SUT is modeled as a deterministic, fully-specified Mealy Machine where each transition is labeled with an (input,output) pair (Lee and Yannakakis, 1996). In the LTS approach, a model is a possibly non-deterministic, possibly infinite-state, possibly not input-enabled labeled transition system, where a label is either an input to the SUT or an output. Compliance between an SUT and an LTS model is formally defined by an *implementation relation* called **ioco** (Tretmans, 2008a).

In Mealy Machine- as well as LTS-based testing labels, i.e., inputs and outputs, needs not be atomic; they can be structured, containing data parameters. A straightforward way to deal with parameterized labels is to "flatten" them by expanding parameters towards all their (finitely many) possible values, so that a standard Mealy Machine or LTS, respectively, is again obtained. A more sophisticated method is to deal with them in a symbolic way and to lift test generation to the symbolic level (Frantzen et al., 2006).

In this paper, we use MBT at two different places. Firstly, as explained in the introduction, MBT is used to test the compliance of an implementation I with a learned model M_R of the reference implementation R. This involves a straightforward application of MBT. The only difference with the standard use of MBT is that the the model M_R is learned instead of manually developed. The second occurrence of MBT is more hidden, and concerns the use of MBT to perform the approximate equivalence queries occurring in the learning process. As explained above, an equivalence query in active learning aims to answer the question whether a hypothesized learned model is a correct model of the SUT. Since the SUT is a black-box, we can use MBT as an intelligent approach to approximate this query: test cases are generated from the hypothesized model M'_I and executed on the SUT I in order to get confidence that the hypothesized model M'_I is correct.

4. Tools

In this article, we use a unique combination of learning, testing and verification tools. In this section, we briefly introduce the tools that we have used, and the translations between model representations that we have implemented.

Uppaal The model-checker Uppaal (Behrmann et al., 2004) is one of the best known model checkers today. It is based on timed automata (Alur and Dill, 1994) and can be used to test logical properties of these systems, extended with integer variables, structured data types, and channel synchronizations between automata. In this article, we use the Uppaal GUI as an editor for extended finite state machines (EFSM). Uppaal models, represented as `.xml` files, can be translated to the corresponding implementations, encoded as Java `.jar` files, and to Labeled Transition Systems (LTSs), represented using the `.aut` format.

LearnLib The LearnLib automata learning tool (Raffelt et al., 2009) is an active learning tool for Mealy machines or DFAs based on the L^* algorithm. It contains many optimizations that reduce the amount of queries asked by L^*, implements multiple learning strategies, and includes different ways to generate membership queries. The LearnLib tool was used by the winning team in the 2010 Zulu DFA active learning competition (Combe et al., 2010). We use LearnLib as the basic active learning tool. The models learned by LearnLib are Mealy machines, represented as `.dot` files. A small script converts Mealy machines in `.dot` format to Labeled Transition Systems in `.aut` format by splitting each transition $q \xrightarrow{i/o} q'$ into a pair of two consecutive transitions $q \xrightarrow{i} q''$ and $q'' \xrightarrow{o} q'$.

CADP Construction and Analysis of Distributed Processes (CADP) (Garavel et al., 2011) is a comprehensive toolbox for verifying models of concurrent systems, i.e., models consisting of multiple concurrent processes that together describe the overall system behavior. Relying on action-based semantic models, it offers functionalities covering the entire design cycle of concurrent systems: specification, simulation, rapid prototyping, verification testing, and performance evaluation. CADP is used in this paper to check equivalence (strong bisimulation) of labeled transition systems represented as `.aut` files.

JTorX JTorX (Belinfante, 2010) is an update of the model-based testing tool TorX (Tretmans and Brinksma, 2003). TorX is a model-based testing tool that uses labeled transition systems to derive and execute tests (execution traces) based on **ioco** (Tretmans, 2008b). Using on-line testing, JTorX can easily generate and execute tests consisting of more than 1 000 000 test events. JTorX is easier to deploy and uses a more advanced version of **ioco**. We use JTorX to establish conformance of mutant implementations to a model of the reference implementation, represented as an `.aut` file.

MRMC The Markov Reward Model Checker (MRMC) (Katoen et al., 2011) is a probabilistic model checker, which can be used to check the probability that a logical statement (such as a system breakdown) occurs in a given continuous- or discrete-time Markov chain, with or without reward functions. We use MRMC to compute the probability of reaching certain states in an implementation within a certain number of steps in a setting where inputs are generated randomly. We wrote a small script that converts LTSs in `.aut` format to DTMCs in `.tra/.lab` format, which are accepted as input by MRMC.

TorXakis TorXakis (Mostowski et al., 2009) is another extension of the TorX model-based testing tool. In addition to the testing algorithms, TorXakis uses symbolic test generation to deal with structured data, i.e., symbols with data parameters (Frantzen et al., 2005), where TorX and JTorx use flattening. By exploiting the structure of input actions, TorXakis is able to find certain counterexamples much faster than LearnLib and JTorX.

Tomte Tomte (Aarts et al., 2012) is a tool that aims to bridge the gap between active learning tools such as LearnLib and real software systems. Tomte uses and learns abstractions in order to map the extended finite state machine world of software systems into the Mealy machine world of active learning tools such as LearnLib. It then uses such a learning tool to learn an abstract Mealy machine, which is later transformed back into an EFSM. Currently, it supports learning abstractions for equality of parameter values. By exploiting the structure of states and actions, Tomte is able to construct — for one of the implementations — models much faster than LearnLib.

5. Experiments

In this section, we report on the experiments that we did using LearnLib and JTorX to establish conformance of the six mutant implementations to the reference implementation.

Learning BRP models In order to learn models of the reference implementation and its mutants, we connect the implementations, which serve as SUT, to the LearnLib tool. In order to approximate the equivalence queries, we used the LearnLib test suite with randomly generated test traces containing 100 to 150 inputs.

The results of the inference of the reference implementation and the six mutants are shown in Table 1. For every implementation, we list the number of states in the learned model, as well as the total number of membership queries (MQ) and the time needed to construct the final hypothesis. Moreover, we list the total number of test traces generated for approximating equivalence queries (EQ) and the time needed to find all counterexamples. Note that these numbers and times do not include the last equivalence query, in which no counterexample has been found. All times mentioned are rounded down, e.g., 00:00:00 refers

	RefImp	**Mut1**	**Mut2**	**Mut3**	**Mut4**	**Mut5**	**Mut6**
# states	156	156	128	156	156	156	136
# MQ	23448	21868	19247	21878	23448	23448	20448
time MQ	00:15:39	00:14:33	00:12:52	00:14:36	00:15:39	00:15:39	00:13:39
# EQ	5	2262	5	5	5	5	5
time EQ	00:00:00	00:01:47	00:00:00	00:00:00	00:00:00	00:00:00	00:00:00

Table 1: Learning statistics for the BRP reference implementation and mutants 1-6

		rn5	**rn7**	**rn10**	**rn15**	**rn20**	**rn22**
vr0-1	# states	156	212	295	436	576	Not possible to learn correct model within 5 hours
	# MQ	23448	36102	47414	74189	121069	
	time MQ	00:15:39	00:24:07	00:31:41	00:49:51	01:24:07	
	# EQ	5	77	299	11527	152185	
	time EQ	00:00:00	00:00:03	00:00:14	00:09:07	02:01:00	
		rn4	**rn5**	**rn6**			
vr0-2	# states	340	418	Not possible to learn correct model within 5 hours			
	# MQ	345170	436477				
	time MQ	03:50:12	04:52:42				
	# EQ	11	30				
	time EQ	00:00:00	00:00:01				

Table 2: Learning statistics for reference implementation

to $<$ 500 ms. Using CADP, we verified that all the learned models indeed are correct, i.e., equivalent to the Uppaal models described in Section 2.

If we take a closer look at Table 1, we observe some interesting peculiarities. First, the number of equivalence queries for mutant 1 is much higher than for the other implementations. The reason for this is that mutant 1 also accepts new requests in state WA. This makes it much harder to find a counterexample that produces an OCONF(0) or OCONF(2) output, since this requires six successive ITIMEOUT inputs without intermediate IREQ inputs. However, the probability that LearnLib selects (uniformly at random) six successive ITIMEOUT inputs in a row is low, since each time ITIMEOUT only has a 10% chance of being selected. This issue will be analyzed in more detail in Section 6. Second, the numbers for mutant 2 are slightly smaller than for the other implementations. The reason for this is that in mutant 2 the maximal number of retransmissions is smaller: 4 instead of 5. The size of the model and the times required for constructing and testing hypotheses (explored in the next section) all depend on the maximal number of retransmissions.

More learning experiments Besides the maximal value of the retransmission counter, also changes in the domain of message parameters m_1, m_2, and m_3 will influence the learning results for the different implementations. Therefore, we run some additional experiments for different parameter settings of the reference implementation and mutant 1 (the behavior of mutants 2-6 is similar to that of the reference implementation). We evaluate how LearnLib performs for different maximal values for the retransmission counter rn. Moreover, we investigate what happens when we allow three possible values for each message parameter.

		rn2	**rn3**	**rn4**	**rn5**	**rn6**	**rn7**
vr0-1	# states	72	100	128	156	184	Not possible to learn correct model within 5 hours
	# MQ	9386	13025	17954	21868	27643	
	time MQ	00:06:13	00:08:39	00:11:56	00:14:33	00:18:24	
	# EQ	12	133	1183	2262	270809	
	time EQ	00:00:00	00:00:06	00:00:56	00:01:47	03:35:28	
vr0-2	# states	184	262	340	Not possible to learn correct model within 5 hours		
	# MQ	170796	243177	325433			
	time MQ	01:53:53	02:42:19	03:37:06			
	# EQ	21	4634	23993			
	time EQ	00:00:00	00:03:36	00:18:49			

Table 3: Learning statistics for mutant 1

	counterexample	**output**	**expected**
Mut1	IR(0,0,0) IR(0,0,0)	OF(1,0,0,0)	ONOK()
Mut2	IR(0,0,0) IT() IT() IT() IT() IT()	OCONF(0)	OF(1,0,0,0)
Mut3	IR(0,0,0) IT()	OF(1,0,1,0)	OF(1,0,0,0)
Mut4	IR(0,0,0) IA() IA()	OF(1,0,0,0)	OF(0,1,0,0)
Mut5	IR(0,0,0) IA() IA() IT() IT() IT() IT() IT() IT()	OCONF(0)	OCONF(2)
Mut6	IR(0,0,1) IA() IA()	OF(0,1,0,0)	OF(0,1,0,1)

Table 4: Equivalence checking of mutant models and reference implementation (IT = ITIMEOUT, IR = IREQ, IA = IACK, OF = OFRAME).

Table 2 and 3 show the results of learning models of the reference implementation and mutant 1 using different maximal numbers of retransmission and different value ranges for the messages m_1, m_2, and m_3. Increasing the number of messages leads to a sharp growth of the time required to construct a hypothesis, whereas increasing the maximal number of retransmissions leads to a fast growth of the time required to find counterexamples for incorrect hypotheses. For mutant 1, the time needed for testing increases so fast that if the maximal number of retransmissions is 7 and there are 2 messages, no correct model can be learned within 5 hours. Also in the case where the maximal number of retransmissions is 5 and there are 3 messages, LearnLib is not able to construct a correct model for mutant 1 within 5 hours. This is not surprising, because in both cases the probability to select a counterexample is even lower than for mutant 1 in Table 1.

Conformance checking We compare the two conformance testing methods described in the introduction. We only consider the models with at most 5 retransmissions and 2 different messages.

The first method used the CADP equivalence checker to compare the models M_I that we learned for the mutant implementations I with the model M_R learned for the reference implementation R.[1] For each of the mutants, CADP quickly found a counterexample trace illustrating the difference between the models of the mutant and the model of the reference implementation. The counterexamples found by CADP are depicted in Table 4.

[1] Essentially the same counterexamples were also found using the JTorX ioco checker.

	Mut1	**Mut2**	**Mut3**	**Mut4**	**Mut5**	**Mut6**
test steps	11	397	8	28	1937	127
time	00:00:02	00:00:17	00:00:01	00:00:02	00:01:21	00:00:07

Table 5: Conformance testing with learned reference model and mutant implementations

The second method used the model M_R of the reference implementation R as input for the JTorX model based testing tool and the mutant implementations I as SUTs. Test steps were executed until a counterexample was found. Again, JTorX found a counterexample for each of the mutant implementations. The number of test steps (every step is one input or output) and the running times of the experiments are shown in Table 5. The resulting counterexamples are rather long sequences and are therefore not shown in the table.

When we compare the computation times from Table 1 and Table 5, we see that model based testing based on a model of the reference implementation is by far the fastest method for finding bugs in implementations. Learning models of proposed implementations takes more time but also provides more information in the form of a learned model. Such learned models could, for instance, be used for model checking analysis.

6. Further analysis and improvements

MRMC In our experiments, the most effective technique available in LearnLib for approximating equivalence queries turned out to be random testing. In order to analyze the effectiveness of this method, we may compute the probabilities of reaching certain states within a certain number of transitions, by translating the Mealy machine of the teacher (the system under test) into a discrete time Markov chain (DTMC). This DTMC has the same states as the Mealy machine, and the probability of going from state q to state q' is equal to the number of transitions from q to q' in the Mealy machine divided by the total number of inputs. Through analysis of this DTMC, the MRMC model checker can compute the probability of finding certain counterexamples within a given time.

Using MRMC we computed that for the reference implementation with up to 7 retransmissions the probability of reaching, within a single test run of 125 steps, a state with an outgoing $\mathsf{OCONF}(0)$ transition is 0.0247121. This means the probability of reaching a state with an outgoing $\mathsf{OCONF}(0)$ transition within 75 test runs is 0.847. This result explains why LearnLib needed 77 test runs to learn a correct model of this system (see Table 2). Using MRMC, we also computed that for the version of mutant 1 with up to 7 retransmissions the probability of reaching, within 125 steps, a state with an outgoing $\mathsf{OCONF}(0)$ transition is only 0.0000010. This result explains why LearnLib was not able to learn a correct model of this system within 5 hours.

TorXakis As explained above, the probability of finding a counterexample for mutant 1 is related to the probability of reaching a state with an outgoing $\mathsf{OCONF}(0)$ or $\mathsf{OCONF}(2)$ transition. This probability is very low, because it requires a test sequence with $(rn + 1)$ consecutive $\mathsf{ITIMEOUT}$ inputs. Since LearnLib treats any possible instantiation of $\mathsf{IREQ}(m_1, m_2, m_3)$ as a separate input, an increase of vr therefore leads to a dramatic

		rn5	rn7
vr0-1	(#test symbols, st. dev)	(1785, 1921)	(19101, 22558)
vr0-2	(#test symbols, st. dev)	(2991, 2667)	(18895, 15158)
vr0-9	(#test symbols, st. dev)	(3028, 3199)	

Table 6: Equivalence query statistics for TorXakis.

	rn5	**rn6**	**rn7**	**rn8**	**rn9**	**rn10**	**rn12**	**rn15**
# states	38	44	50	56	62	68	80	Not possible to learn correct model within 5 hours
# MQ	750	999	1086	1389	1532	1893	2478	
time MQ	00:00:28	00:00:38	00:00:43	00:00:55	00:01:01	00:01:16	00:01:39	
# EQ	3	3	261	261	261	2332	5482	
time EQ	00:00:18	00:00:17	00:00:29	00:00:29	00:00:28	00:04:07	00:09:43	

Table 7: Mutant 1: Learning statistics using Tomte on **vr0-∞**

reduction of the probability of selecting ITIMEOUT. We investigate how using TorXakis for answering equivalence queries influences this drop in learning performance.

Table 6 summarizes the results obtained with TorXakis when testing mutant 1 against the hypothesized LearnLib model for $rn7, vr0-1$ and $rn5, vr0-2$. These hypothesized models are incorrect but LearnLib did not manage to detect a counterexample. In addition, the results for the scenarios $rn5, vr0-1$, $rn5, vr0-9$, and $rn7, vr0-2$ are presented. The numbers in Table 6 are the average lengths of the test runs, measured in test symbols (both input and output), until a discrepancy between the model and mutant 1 was detected. The average is taken over 10 different random test runs. It can be noted that there is a large variation in the lengths depending on the random selections.

TorXakis is able to detect counterexamples for the incorrect hypothesized models for $rn7, vr0-1$, and for $rn5, vr0-2$. TorXakis achieves this by treating the message parameters symbolically, so that ITIMEOUT, ACK, and IREQ inputs have an equal probability of being selected, i.e., $\frac{1}{3}$. Combined with the fact that TorXakis generates one very long test case, it is able to find a counterexample for the scenario $rn7, vr0-1$ within reasonable time.

Tomte Through the use of counterexample abstraction refinement, Tomte is able to learn models for a restricted class of extended finite state machines in which one can test for equality of data parameters, but no operations on data are allowed. The current version of Tomte requires that only the first and the last occurrence of parameters of actions is remembered. As a result, Tomte can only learn models for instances of mutant 1, where each IREQ input overwrites previous occurrences of the message parameters. For these instances, however, Tomte outperforms LearnLib with several orders of magnitude. Table 7 gives an overview of the statistics for learning mutant 1 with Tomte. Since in Tomte the entire range of message values for mutant 1 is abstracted into a single equivalence class, Tomte needs far fewer queries than LearnLib (cf. Table 3). Work is underway to extend Tomte so that it can also learn the BRP reference implementation and the other mutants.

7. Conclusion

We show how to apply active state machine learning methods to a real-world test-case: conformance testing implementations of the bounded retransmission protocol (BRP). To the best of our knowledge, this use of active learning methods is entirely novel. We demonstrate how to make this application work by combining active learning algorithms with tools from verification (an equivalence checker, CADP) and testing (a model-based test tool, JTorX).

A nice property of the BRP is that it contains two parameter values (the number of retransmissions **rn** and the range of message values **vr**), which can be increased to obtain increasingly complex protocols. This makes it an ideal test-case for state machine learning methods because it allows us to discover the limits of their learning capabilities. We investigated these limits on testing the conformance of six mutant implementations with respect to a given reference implementation. The state-of-the-art LearnLib active learning tool quickly runs into trouble when learning one of these mutant implementations. This case was analyzed separately using a probabilistic model checker (MRMC), and based on this analysis we suggested two ways of improving the performance of the active learning method: using a state-of-the-art model-based test tool (TorXakis) for evaluation of equivalence queries, and using a new learning method based on abstraction refinement (Tomte).

References

F. Aarts and F. Vaandrager. Learning I/O automata. In *CONCUR*, vol. 6269 of *LNCS*, pages 71–85. Springer, 2010.

F. Aarts, F. Heidarian, H. Kuppens, P. Olsen, and F. Vaandrager. Automata learning through counterexample-guided abstraction refinement. In *(submitted), see also: http://www.italia.cs.ru.nl/tools/*, 2012.

R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126: 183–235, 1994.

G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *POPL*, pages 4–16. ACM, 2002.

D. Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.

J. Antunes, N. Neves, and P. Verissimo. Reverse engineering of protocols from network traces. *Reverse Engineering, Working Conference on*, 0:169–178, 2011.

K. Bartlett, R. Scantlebury, and P. Wilkinson. A note on reliable full–duplex transmission over half–duplex links. *Commun. ACM*, 12:260–261, 1969.

G. Behrmann, A. David, and K. Larsen. A tutorial on Uppaal. In *Formal Methods for the Design of Real-Time Systems*, vol. 3185 of *LNCS*, pages 33–35. Springer, 2004.

A. Belinfante. Jtorx: A tool for on-line model-driven test derivation and execution. In *TACAS*, vol. 6015 of *LNCS*, pages 266–270. Springer, 2010.

A. Bertolino, P. Inverardi, P. Pelliccione, and M. Tivoli. Automatic synthesis of behavior protocols for composable web-services. In *FSE*, pages 141–150. ACM, 2009.

J. Castro and R. Gavaldà. Towards feasible PAC-learning of probabilistic deterministic finite automata. In *ICGI*, pages 163–174, 2008.

A. Clark and F. Thollard. PAC-learnability of probabilistic deterministic finite state automata. *Journal of Machine Learning Research*, pages 473–497, 2004.

D. Combe, C. de la Higuera, and J.-C. Janodet. Zulu: An interactive learning competition. In *FSMNLP*, vol. 6062 of *LNCS*, pages 139–146. Springer, 2010.

P. Comparetti, G. Wondracek, C. Krügel, and E. Kirda. Prospex: Protocol specification extraction. In *IEEE Symposium on Security and Privacy*, pages 110–125. IEEE, 2009.

J. E. Cook and A. L. Wolf. Discovering models of software processes from event-based data. *ACM Trans. Softw. Eng. Methodol.*, 7:215–249, July 1998.

W. Cui, J. Kannan, and H. J. Wang. Discoverer: automatic protocol reverse engineering from network traces. In *USENIX Security Symposium*, page nr. 14, 2007.

V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining object behavior with ADABU. In *Int. Workshop on Dynamic Systems Analysis*, pages 17–24. ACM, 2006.

P. D'Argenio, J.-P. Katoen, T. Ruys, and J. Tretmans. The bounded retransmission protocol must be on time! In *TACAS*, vol. 1217 of *LNCS*, pages 416–431. Springer, 1997.

F. Denis, A. Lemay, and A. Terlutte. Learning regular languages using non deterministic finite automata. In *ICGI*, pages 39–50, 2000.

L. Frantzen, J. Tretmans, and T. Willemse. Test generation based on symbolic specifications. In *FATES*, vol. 3395 of *LNCS*, pages 1–15. Springer, 2005.

L. Frantzen, J. Tretmans, and T. Willemse. A Symbolic Framework for Model-Based Testing. In *FATES and RV*, vol. 4262 of *LNCS*, pages 40–54. Springer, 2006.

H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2010: A toolbox for the construction and analysis of distributed processes. In *TACAS*, vol. 6605 of *LNCS*, pages 372–387. Springer, 2011.

O. Grinchtein, B. Jonsson, and P. Petterson. Inference of event-recording automata using timed decision trees. In *CONCUR*, vol. 4137 of *LNCS*, pages 435–449. Springer, 2006.

L. Helmink, M. Sellink, and F. Vaandrager. Proof-checking a data link protocol. In *International Workshop TYPES*, vol. 806 of *LNCS*, pages 127–165. Springer, 1994.

C. d. l. Higuera and J.-C. Janodet. Inference of omega-languages from prefixes. *Theoretical Computer Science*, 313(2):295–312, 2004.

J.-P. Katoen, I. S. Zapreev, E. M. Hahn, H. Hermanns, and D. N. Jansen. The ins and outs of the probabilistic model checker mrmc. *Performance Evaluation*, 68(2):90 – 104, 2011.

D. Lee and M. Yannakakis. Principles and Methods for Testing Finite State Machines – A Survey. *Proceedings of the IEEE*, 84(8):1090–1123., August 1996.

T. Margaria, O. Niese, H. Raffelt, and B. Steffen. Efficient test-based model generation for legacy reactive systems. In *High-Level Design Validation and Test Workshop*, pages 95–100, IEEE, 2004.

L. Mariani, F. Pastore, and M. Pezze. Dynamic analysis for diagnosing integration faults. *IEEE Transactions on Software Engineering*, 37:486–508, 2011.

W. Mostowski, E. Poll, J. Schmaltz, J. Tretmans, and R. Wichers Schreur. Model-based testing of electronic passports. In *Formal Methods for Industrial Critical Systems*, vol. 5825 of *LNCS*, pages 207–209. Springer, 2009.

H. Raffelt, B. Steffen, T. Berg, and T. Margaria. Learnlib: a framework for extrapolating behavioral models. *Int. Journal on Software Tools for Technology Transfer*, 11:393–407, 2009.

T. A. Sudkamp. *Languages and Machines: an introduction to the theory of computer science.* Addison-Wesley, third edition, 2006.

J. Tretmans. Formal methods and testing. In *Formal Methods and Testing*, chapter Model based testing with labelled transition systems, pages 1–38. Springer, 2008a.

J. Tretmans. Model based testing with labelled transition systems. In *Formal Methods and Testing*, vol. 4949 of *LNCS*, pages 1–38. Springer, 2008b.

J. Tretmans and E. Brinksma. Torx: Automated model-based testing. In *First European Conference on Model-Driven Software Engineering*, pages 31–43, 2003.

M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach.* Morgan-Kaufmann, 2007.

W. M. P. van der Aalst. *Process Mining - Discovery, Conformance and Enhancement of Business Processes.* Springer, 2011.

S. Verwer. *Efficient Identification of Timed Automata: Theory and Practice.* PhD thesis, Delft University of Technology, 2010.

N. Walkinshaw, K. Bogdanov, M. Holcombe, and S. Salahuddin. Reverse engineering state machines by interactive grammar inference. In *Working Conference on Reverse Engineering*, pages 209–218. IEEE, 2007.

N. Walkinshaw, K. Bogdanov, C. Damas, B. Lambeau, and P. Dupont. A framework for the competitive evaluation of model inference techniques. In *First International Workshop on Model Inference In Testing*, pages 1–9. ACM, 2010.

T. Yokomori. Learning non-deterministic finite automata from queries and counterexamples. In *Machine Intelligence*, pages 196–189. University Press, 1993.