

Improving active Mealy machine learning for protocol conformance testing

Fides Aarts · Harco Kuppens · Jan Tretmans ·
Frits Vaandrager · Sicco Verwer

Received: 17 December 2012 / Accepted: 7 August 2013 / Published online: 3 October 2013
© The Author(s) 2013

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 models of and revealing errors in implementations, present the new notion of a conformance oracle, and demonstrate how conformance oracles can be used to speed up conformance checking.

Keywords Active learning · Automaton learning · Mealy machines · State machine synthesis · Model-based testing · Protocol learning · Model checking

1 Introduction

Active learning is a type of machine learning in which the learner (an algorithm) is allowed to ask questions (queries) to an oracle, see, e.g. Settles (2010). In machine learning, such an

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

F. Aarts · H. Kuppens · J. Tretmans · F. Vaandrager · S. Verwer (✉)
Institute for Computing and Information Sciences, Radboud University Nijmegen, P.O. Box 9010, 6500
GL Nijmegen, The Netherlands
e-mail: siccoverwer@gmail.com

J. Tretmans
TNO—Embedded Systems Innovation, P.O. Box 513, 5600 MB Eindhoven, The Netherlands

oracle is frequently seen as a human annotator which can assign labels to training instances. A learner can use this feedback to find or improve a model for the training data. Moreover, by asking informative queries (e.g., close to the decision boundary), an active learner potentially requires fewer examples than a passive learner that learns from data samples. Since annotation can be quite costly, the goal is to ask as few questions as possible in order to reach a good quality model.

This paper is concerned with an application of active learning. However, instead of asking queries to a human annotator, we use and develop methods that ask queries to *software systems*. These methods provide a software system with input (a training instance), read the corresponding output (e.g., a label), and use this to update their current hypothesis (model) of the software's internal behavior. Although the goal in both of these settings is to learn a good model, there exists a fundamental difference in the cost associated with asking queries. In the case of software systems, this cost is (usually) low. Our methods can consequently ask millions of queries within minutes. The LearnLib toolkit (Raffelt et al. 2009), winner of the 2010 Zulu active state machine learning competition (Combe et al. 2010), can therefore learn models consisting of up to tens of thousands of states. Active state machine learning tools have been used successfully for many different applications in software engineering such as regression testing of software components (Hungar et al. 2003), fuzz testing of protocol implementations (Comparetti et al. 2009), and inference of botnet protocols (Cho et al. 2010).

Many active state machine learning algorithms are based on the well-known L^* -algorithm (Angluin 1987). In addition to asking *membership queries* (input-output pairs), L^* asks *equivalence queries*, which test whether the current hypothesis is correct. Intuitively, the membership queries are used to single out a state machine \mathcal{M} within a certain size bound by proving all others to be inconsistent, an equivalence query is then used to find a counterexample for \mathcal{M} and increase this size bound. L^* uses the information contained in these counterexamples in order to use at most a polynomial number of membership queries in the size of the learned model. While usually only a few counterexamples are needed for L^* to converge, answering equivalence queries is often the bottleneck in active state machine learning application. The reason is that testing whether a black-box software system behaves according to a given state machine description is hard (Dalal et al. 1999). Constructing informative test queries (i.e., membership queries) is known as *test selection* and is one of the main problems dealt with in model-based software testing (Lee and Yannakakis 1996). Several software testing methods and tools have been developed for this task in order to aid software development and maintenance, see, e.g., Belinfante (2010), Mostowski et al. (2009), Shafique and Labiche (2010). Although, these methods are able to approximate equivalence queries by (smartly) asking many membership queries, an exponential amount of them (or one of exponential length) are required in the worst case.

In this paper, we investigate a novel application domain for active learning of software models: establishing the correctness of protocol implementations relative to a given reference implementation. In protocol engineering, protocols are typically specified by independent, authoritative agencies, e.g., the ITU (International Telecommunication Union) or the IETF (Internet Engineering Task Force). For such a protocol specification, a reference implementation is then often developed, which helps to discover errors and ambiguities in the specification, demonstrates that the specification is actually implementable, and serves as a definite interpretation of the specification. Based on the specification and the reference implementation, independent suppliers will develop products implementing the protocol. The reference implementation is then the standard against which these products are measured and compared to assess compliance with the specification.

To the best of our knowledge, this is a novel application area of grammatical inference and machine learning. Moreover, it is a promising one since reference implementations are in existence for many real-world software systems, but models are usually lacking or incomplete (Aarts et al. 2010a, 2010b).

Our investigation is focused on a well-known benchmark case study from the verification literature: the bounded retransmission protocol (Helmink et al. 1994; D’Argenio et al. 1997) (see Sect. 5). 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 (see Appendix A). Our aim is to combine active learning methods with model-based testing (see Sect. 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 Sect. 3 and Appendix B). We show how these tools can be used for learning models of the bounded retransmission protocol and revealing implementation errors in the mutants (Sects. 6 and 7).

In addition to experimental results on learning the bounded retransmission protocol, we provide two solutions that significantly reduce the difficulty of answering equivalence queries in this setting:

1. Using abstractions over input and output values through our Tomte learning tool (Aarts et al. 2012a) and through the model-based testing tool TorXakis (Mostowski et al. 2009) (see Sect. 3). Tomte performs on-the-fly abstractions on concrete input and output values, only introducing new abstract values when required for learning.
2. Using a previously learned model of the reference implementation together with the current hypothesis in a model equivalence checker (such as the popular CADP model checker (Garavel et al. 2011)). The resulting agent, which we call a *conformance oracle* (see Sect. 4), effectively transfers knowledge from the reference learning task to the task of learning a mutant, i.e., a slight variation of the reference implementation. The speedup results from the fact that test selection is difficult while equivalence testing is easy.

Our main contributions are 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. The bounded retransmission protocol use case can serve as a benchmark for future active learning and testing methods.

Our research takes place at the interface of model-based testing and model inference, and builds upon a rich research tradition in this area. The idea of combining testing and learning of software systems was first explored by Weyuker, who observed in 1983 “Program testing and program inference can be thought of as being inverse processes” (Weyuker 1983). Recently, there has been much interest in relating model-based testing and model inference in the setting of state machines. Berg et al. (2005), for instance, point out that some of the key algorithms that are used in the two areas are closely related. Walkinshaw et al. (2010b) show that active learning itself is an important source of structural test cases. At the ISO/IEC 2012 conference a special session was dedicated to the combination of model-based testing and model inference (Meinke and Walkinshaw 2012), a combination which is often denoted by the term *learning-based testing*. As far as we know, no previous work in this area addresses the problem of conformance with respect to a reference implementation. In addition, the specific combination of tools that we use is new, as well as the case study, and the concept of a conformance oracle.

A preliminary version of this paper appeared as Aarts et al. (2012b). Besides providing background information, technical details, and full/improved experimental data, the most significant contribution beyond Aarts et al. (2012b) is the new notion of a conformance oracle.

2 Software model synthesis

The motivation of our work stems from the fact that the behavior of software systems can often be specified using finite state machine models. These models capture the behavior of software systems, by describing the way in which they react to different inputs, and when they produce which output. Visualizing such state machines can provide insights into the behavior of a software system, which can be of vital importance during the design and specification of a system. Moreover, state machines can be used to test and analyze a software system's properties using model checking (Clarke 1997) and testing techniques (Broy et al. 2005). In practice, one often encounters software systems without formal behavioral specifications. Reasons for this situation are manifold: developing and maintaining them is often considered to be too costly (Walkinshaw et al. 2010a), legacy software is often kept running while its documentation is lost or outdated, computer viruses are specifically designed to be difficult to comprehend, and existing software models are often proprietary information of the software developer.

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). Software model synthesis is a technique for automatically constructing a software model based on observed system behavior. 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. The problem of learning state machines therefore enjoys a lot of interest from the software engineering and formal methods communities. Many different language models and ways of finding them are available in the grammatical inference (e.g., de la Higuera 2010) literature. Which one to choose depends mostly on the available data and the type of system under consideration.

In the context of this paper, we assume the existence of a reference implementation of a well-known benchmark case study from the verification literature. Given the reference implementation and some possibly faulty implementations provided by suppliers, our goal is to quickly discover the behavioral differences between the reference and the supplied programs. Using a reference implementation in order to establish the correctness of other implementations is an important problem in software engineering since reference implementations are in existence for many real-world software systems. Furthermore, since models are usually lacking or incomplete, it is often impossible to verify correctness or other properties using existing verification technologies such as model checking.

The models that we use in this paper are Mealy machines, which are a deterministic finite state automaton (DFA) variant with alternating input and output symbols, see, e.g., Sudkamp (2006). Mealy machines are 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 learning 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; Verwer et al. 2011; Grinchtein et al. 2006), I/O automata (Aarts and Vaandrager 2010), and Büchi automata (de la Higuera and Janodet 2004). Despite their limited power, DFA and Mealy machine learning methods have recently been applied successfully 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).

We expect that the approaches that are explored in this paper for Mealy machines can also be applied in the setting of more powerful modeling frameworks. In Aarts and Vaandrager (2010), for instance, we have shown that the specific restriction of Mealy machines that inputs and outputs have to alternate, which is often inconvenient in practice, is not fundamental: any tool for active learning of Mealy machines can be used for learning I/O automata that are deterministic and output determined.

3 Mealy machines, active learning, and model-based testing

We learn Mealy machines from queries using tools that build upon the well-known L^* algorithm, see Angluin (1987), Kearns and Vazirani (1994). In query learning, access to an oracle is needed that can answer specific types of questions such as: whether a specific string is part of the language (membership queries), and whether a given model is a model for the language (equivalence queries). In software model synthesis, the actual software system can be used for this purpose, see, e.g., Raffelt et al. (2009). When such an oracle is available that can be queried often and quickly, it is possible to identify very large realistic models. In this section, we first give formal descriptions of Mealy machines and the methods for testing and learning them. Afterwards, we describe how we combine state-of-the-art testing and learning tools in order to establish the conformance of an implementation relative to a given reference implementation.

3.1 Mealy machines

Mealy machines are a type of transducer very similar to deterministic finite state automata (DFAs, see, e.g., Sudkamp 2006). Instead of accepting or rejecting (classifying) an input string, Mealy machines produce (transduce) an output symbol for every input symbol transition visited (fired) by this run. Since software systems typically produce multiple outputs for multiple inputs, this makes them very suitable as software models. In particular, Mealy machines are very popular models for communication protocols since these often alternate between input and output communication.

Definition 1 A *Mealy machine (MM)* is a tuple $\mathcal{M} = \langle I, O, Q, q_0, \rightarrow \rangle$, where

- I , O , and Q are nonempty sets of *input symbols*, *output symbols*, and *states*, respectively,
- $q_0 \in Q$ is the *initial state*, and
- $\rightarrow \subseteq Q \times I \times O \times Q$ is the *transition relation*.

We write $q \xrightarrow{i/o} q'$ if $(q, i, o, q') \in \rightarrow$.

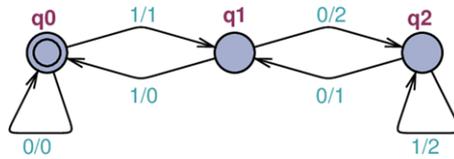


Fig. 1 A Mealy machine for computing residues modulo 3. For example, the machine maps 010 to 012 (path: $q_0 \xrightarrow{0/0} q_0, q_0 \xrightarrow{1/1} q_1, q_1 \xrightarrow{0/2} q_2$) computing $0 \bmod 3 = 0, 1 \bmod 3 = 1, 2 \bmod 3 = 2$; 1101 is mapped to 1001 computing $1 \bmod 3 = 1, 3 \bmod 3 = 0, 6 \bmod 3 = 0, 13 \bmod 3 = 1$; and 1110 to 1012 computing $1 \bmod 3 = 1, 3 \bmod 3 = 0, 7 \bmod 3 = 1, 14 \bmod 3 = 2$

Table 1 Transition relation for the Mealy machine in Fig. 1

Source state	Input		1	
	Output	Target state	Output	Target state
q_0	0	q_0	1	q_1
q_1	2	q_2	0	q_0
q_2	1	$q - 1$	2	q_2

Mealy machines are assumed to be *input enabled* (or *completely specified*): for each state q and input i , there exists an output o such that $q \xrightarrow{i/o}$. An intuitive interpretation of a Mealy machine is as follows. At any point in time, the machine is in some state $q \in Q$. It is possible to give inputs to the machine by supplying an input symbol $i \in I$. The machine then selects a transition $q \xrightarrow{i/o} q'$, produces output symbol o , and jumps to the new state q' . A Mealy machine \mathcal{M} is *deterministic* if for each state q and each input symbol i there is exactly one output symbol o and exactly one state q' such that $q \xrightarrow{i/o} q'$ ($q, q' \in Q, i \in I$). We say that a Mealy machine is *finite* if the set Q of states and the set I of inputs are finite.

Example 1 An example of a Mealy machine for computing residues modulo 3 for a binary input (most significant bit first) number is given in Fig. 1. Given an input word $i_1 \dots i_n$, the j th output of the machine corresponds to $i_1 \dots i_j$ modulo 3. The set of inputs is $I = \{0, 1\}$, the set of outputs is $O = \{0, 1, 2\}$, and the set of states is given by $Q = \{q_0, q_1, q_2\}$, where q_0 is the initial state marked with an extra circle. The transition relation is defined by Table 1.

The transition relation in Mealy machines is extended from symbols to strings (traces, sequences) as follows:

Definition 2 The *extended transition relation* $\xRightarrow{u/s}$ on a Mealy machine $\mathcal{M} = \langle I, O, Q, q_0, \rightarrow \rangle$ is the least relation that satisfies, for $q, q', q'' \in Q, u \in I^*, s \in O^*, i \in I$, and $o \in O$:

- $q \xRightarrow{\epsilon/\epsilon} q$, and
- if $q \xrightarrow{i/o} q'$ and $q' \xRightarrow{u/s} q''$ then $q \xRightarrow{i u/o s} q''$,

where ϵ denotes the empty sequence. A state $q \in Q$ is called *reachable* if $q_0 \xRightarrow{u/s} q$, for some $u \in I^*$ and $s \in O^*$.

The output strings that can be observed after supplying an input string to a Mealy machine are defined using this relation:

Definition 3 An *observation* is a pair $(u, s) \in I^* \times O^*$ such that sequences u and s have the same length. For $q \in Q$, we define $obs_{\mathcal{M}}(q)$ to be the set of observations of \mathcal{M} from state q , by

$$obs_{\mathcal{M}}(q) = \{(u, s) \in I^* \times O^* \mid \exists q' : q \xrightarrow{u/s} q'\}.$$

We write $obs_{\mathcal{M}}$ as a shorthand for $obs_{\mathcal{M}}(q_0)$.

Since Mealy machines are input enabled, $obs_{\mathcal{M}}(q)$ contains at least one pair (u, s) for each input sequence $u \in I^*$. We call \mathcal{M} *behavior deterministic* if $obs_{\mathcal{M}}$ contains exactly one pair (u, s) , for each $u \in I^*$. In this case, we write $out_{\mathcal{M}}(u)$ to denote the unique s with $(u, s) \in obs_{\mathcal{M}}$. In this paper, we consider only behavior deterministic Mealy machines. It is easy to see that a deterministic Mealy machine is also behavior deterministic. Two states $q, q' \in Q$ are *observation equivalent*, denoted $q \approx q'$, if $obs_{\mathcal{M}}(q) = obs_{\mathcal{M}}(q')$. Two Mealy machines \mathcal{M}_1 and \mathcal{M}_2 with the same sets of input symbols are *observation equivalent*, notation $\mathcal{M}_1 \approx \mathcal{M}_2$, if $obs_{\mathcal{M}_1} = obs_{\mathcal{M}_2}$.

3.2 Active learning of Mealy machines

Active learning or query learning is a learning setting in which a learner can ask questions (queries) to a teacher. In our case, the teacher consists of an *implementation*, a black-box software system that we would like to analyze, in combination with an *oracle* that produces statements about the correctness of models produced by the learner. By providing the implementation with inputs, and reading the generated outputs, the learner tries to determine (reverse engineer) its inner workings. Once it has constructed a hypothesis (a model) for these workings, it asks the oracle whether this hypothesis is correct. With some modifications (Margarita et al. 2004), we can apply the well-known L^* DFA learning algorithm (Angluin 1987) to this setting in order to learn a Mealy machine model for a black-box software system. The basic setup for active learning is illustrated in Fig. 2.

Let $\mathcal{M} = \langle I, O, Q, q_0, \rightarrow \rangle$ be a behavior deterministic Mealy machine. An *implementation* of \mathcal{M} is a device that accepts sequences $u \in I^*$ of input symbols, called *output queries* or *membership queries*. Whenever an implementation of \mathcal{M} receives a query u , it returns $out_{\mathcal{M}}(u)$ as output.

An *oracle* is a device which accepts Mealy machine models as inputs. These input models are referred to as *equivalence queries* or *hypotheses*. Upon receiving a hypothesis \mathcal{H} , the oracle may experiment with an implementation of a Mealy machine \mathcal{M} by posing output queries. After a finite number of output queries, the oracle will either produce output *pass* together with model \mathcal{H} , or else output *fail* together with a *counterexample*, which is an observation $(u, s) \in (obs_{\mathcal{M}} \setminus obs_{\mathcal{H}}) \cup (obs_{\mathcal{H}} \setminus obs_{\mathcal{M}})$. A *proper oracle* is an oracle that only generates *pass* when $\mathcal{M} \approx \mathcal{H}$. The combination of an implementation of \mathcal{M} and a proper oracle corresponds to what Angluin (Angluin 1987) calls a *minimally adequate teacher* for \mathcal{M} .

A *learner* is a device that, once it is triggered by a *go*-command, may pose output queries to an implementation of some (unknown) Mealy machine \mathcal{M} , and equivalence queries to an oracle for this implementation, see Fig. 2. The task of the learner is to learn a model that is observation equivalent to \mathcal{M} in a finite number of steps, from the answers generated by

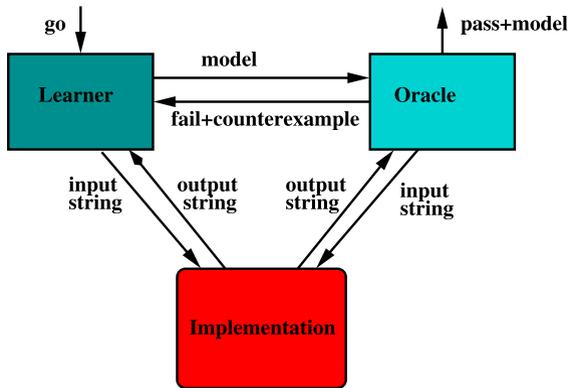


Fig. 2 Basic setting for active learning of software models. A learner starts by providing input sequences to an implementation of the software system (output queries), the implementation produces the corresponding outputs. By intelligently choosing these inputs, the learner is able to form a hypothesis model of the implementation’s internal behavior. This model is provided to an oracle (equivalence queries). The oracle then tests (by providing inputs and checking the generated outputs) whether this model correctly describes the inner workings of the implementation. If so, it returns pass. Otherwise, it returns fail and a counterexample demonstrating this incorrectness

the implementation and the oracle in response to the queries. The typical behavior of an L^* -style learner is to start by asking sequences of output queries until a “stable” hypothesis \mathcal{H} can be built from the answers. After that an equivalence query is made to find out whether \mathcal{H} is correct. If the oracle answers *pass* then the learner has succeeded. Otherwise the returned counterexample is added to the set of observations, and additional output queries are asked until the learner has singled out a new smallest stable hypothesis \mathcal{H}' that is consistent with the current set of observations. This new hypothesis \mathcal{H}' is sent to the oracle in another equivalence query, and a possible counterexample is then again used in output queries in order to update the current hypothesis. This process is iterated until the oracle answers *pass* in response to an equivalence query. A nice property of L^* and query learning is that, assuming a proper oracle, it requires only a polynomial number of queries in order to find \mathcal{H} such that $\mathcal{M} \approx \mathcal{H}$. This is surprising since learning the smallest deterministic finite state automaton (and by restriction Mealy machine) that is consistent with a given dataset is well-known to be NP-hard (Gold 1978).

LearnLib LearnLib (Raffelt et al. 2009) is a tool that supports active learning of Mealy machines or DFAs based on the L^* algorithm. It contains many optimizations that reduce the number of queries asked by L^* , and a means of selecting them in a graphical user interface. Furthermore, it implements multiple learning strategies, that differ in the method used to construct the model (depth-first or breadth-first), and includes different ways to generate membership queries. Finally, the LearnLib tool also includes various model-based testing algorithms (see Sect. 3.4) in order to implement the oracle component. 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.

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, specified using a subset of computation tree logic

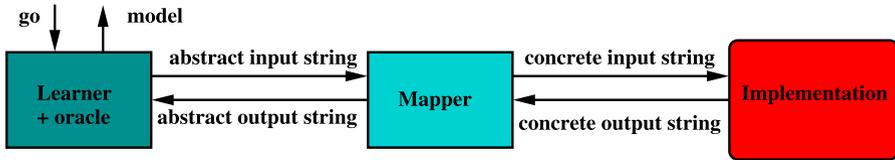


Fig. 3 Introduction of the mapper component. A mapper acts in between the implementation and the learner. It maps input sequences from the abstract (discrete) domain used by the learner to the concrete (integers/floats/messages) domain of the implementation. In reverse, it maps the concrete output to abstract output, which serves as input for the learner. An abstract input can be mapped to different concrete inputs and when two of these result in different abstract outputs, the mapper refines the abstraction on-the-fly in order to remove this non-deterministic behavior

(CTL) (Henzinger et al. 1994). In addition to the timed properties of systems, Uppaal models can contain integer variables, structured data types, and channel synchronizations between automata. It contains an extensive graphical user interface including an automaton editor, an intuitive query language, a simulator, and a verification engine.

In this article, we use the Uppaal GUI as an editor for extended finite state machines (EFSM).

3.3 Automatic abstraction refinement

Tools that are able to bridge the gap between active learning tools such as LearnLib and real software systems are required for numerous applications in different domains. Instead of learning deterministic finite state automata or Mealy machines, they aim at learning models of extended finite state machines (EFSM), including parameter values and guarded transitions based on these values. Abstraction is the key for scaling existing automata learning methods to realistic applications. The idea is to place a *mapper* in between the implementation and the learner/oracle, that abstracts (in a history dependent manner) the large set of actions of the implementation into a small set of abstract actions for the learner/oracle, see Fig. 3.

The concept of an intermediate component that takes care of abstraction is very natural and is used, implicitly or explicitly, in many case studies on automata learning. Aarts et al. (2010a) formalized the concept of such an intermediate abstraction component. By combining the abstract machine \mathcal{H} learned in this way with information about the mapper, it is possible to effectively learn an EFSM that is equivalent to the Mealy machine \mathcal{M} of the implementation. Roughly speaking, the learner is responsible for learning the global “control modes” in which the system can be, and the transitions between those modes, whereas the mapper records some relevant state variables (typically computed from the data parameters of previous input and output actions) and takes care of the data part of the implementation.

Tomte Aarts et al. (2012a) is an automatic abstraction refinement tool that 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. Tomte is named after the creature that shrank Nils Holgersson into a gnome and (after numerous adventures) changed him back to his normal size again. This is exactly what the Tomte tool is doing: It shrinks the alphabet in order to be able to learn an abstract model (through numerous experiments), and then the abstract model is enlarged again to a model of the original implementation by enriching it with information from the mapper component.

The current version of Tomte is able to automatically construct mappers for a restricted class of extended finite state machines, which is called scalarset Mealy machines, in which one can test for equality of data parameters, but no operations on data are allowed. The notion of a scalarset data type originates from model checking, where it has been used by Ip & Dill for symmetry reduction (Ip and Dill 1996). Currently, Tomte can learn models that may only remember the first and last occurrence of a parameter. A major challenge is the development of new algorithms for the automatic construction of mappers: the availability of such algorithms will allow to infer a wider class of systems and boost the applicability of automata learning technology.

Tomte uses the technique of counterexample-guided abstraction refinement: initially, the algorithm starts with a very coarse abstraction, which is subsequently refined if it turns out that the learned model is not behavior-deterministic. Nondeterminism arises naturally when we apply abstraction: it may occur that the behavior of the implementation is fully deterministic but that due to the mapper (which, for instance, abstracts from the value of certain input parameters), the implementation appears to behave nondeterministically from the perspective of the learner. Tomte uses LearnLib as basic learning tool and therefore the abstraction of the implementation may not exhibit any nondeterminism: if it does then LearnLib crashes and Tomte has to refine the abstraction.

Using Tomte, it is possible to learn fully automatically models of several realistic software components, including the biometric passport (Aarts et al. 2010b) as well as the SIP and TCP protocol (Aarts et al. 2010a).

3.4 Model-based testing

Unfortunately, in practice there is unlikely to be an oracle that can answer equivalence queries, making a direct application of L^* often impossible. The typical solution (also adopted in this paper) is to *approximate* these queries using randomly generated membership queries (Angluin 1987). Thus, for every equivalence query, we generate many input strings, for each we ask a membership query, and if an observed output string is different from the output generated by the model, we return this string as a counterexample. If no such string is found, there is some confidence that the current hypothesis is correct and the more strings we test, the higher this confidence. Since state machines cannot be learned from a polynomial amount of membership queries (Balcázar et al. 1997), this procedure requires an exponential number of queries in the worst case. Instead of generating these approximate equivalence queries uniformly at random, it therefore makes sense to use the current hypothesis and observations to only ask informative ones. The problem of finding such informative membership queries is known as *test selection*, which is one of the main problems dealt with by *model-based testing*.

In *Model-based testing* (MBT), which is a new technique that aims to make testing more efficient and more effective (Utting and Legeard 2007), the *system under test* (SUT) is tested against a model of its behavior. This model, which is usually developed manually, must specify what the SUT shall do. Test cases can then be algorithmically generated from this model using an MBT tool. When these test cases are executed on the SUT and the actual test outcomes are compared with the model, the result is an indication about compliance of the SUT with the model. Usually, MBT algorithms and tools are *sound*, i.e., a test failure assures non-compliance, but they are not *exhaustive*, i.e., absence of failing tests does not assure compliance: “Program testing can be used to show the presence of bugs, but never to show their absence!” (Dijkstra 1969).

MBT approaches differ in the kind of models that they support, e.g., state-based models, pre- and post-conditions, (timed) automata, or equational axioms, and in the algorithms that

they use for test generation. In this paper we concentrate on two state-based approaches: finite, deterministic Mealy machines (also called Finite State Machine FSM), and a class of non-deterministic automata, also referred to as labeled transition systems (LTS).

In the Mealy machine approach to MBT, the goal is to test whether a black-box SUT, which is an implementation of an unknown Mealy machine I , is observation equivalent to a given Mealy machine specification S , i.e., to test whether $I \approx S$ (Lee and Yannakakis 1996).

The LTS approach, which does not require determinism, finiteness of states and inputs, input-enabledness, nor alternation of inputs and outputs (a label on a transition is either an input or an output), is more expressive than Mealy machines. Consequently, it requires a more sophisticated notion of compliance between an SUT and a model. The implementation relation **ioco** often serves this purpose (Tretmans 2008). The tools JTorX and TorXakis, among others, generate tests based on this relation; see below. Since it is straightforward to transform a Mealy machine into an LTS, by splitting every (input,output)-pair transition into two LTS transitions with an intermediate state, LTS-based testing can be easily applied to Mealy machine models.

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 2008), a theory for defining when an implementation of a given specification is correct. Using on-line testing, TorX 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**. It contains a graphical user interface for easy configuration, a simulator for guided evaluation of a test trace, interfaces for communication with an SUT, and state-of-the-art testing algorithms.

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 Transition Systems (STSs, LTSs with data variables) with symbolic test generation to deal with structured data, i.e., symbols with data parameters (Frantzen et al. 2005), where TorX and JTorX use flattening, i.e., by replacing a parameter by all its possible, necessarily finitely many values. By working symbolically and by exploiting the structure of input actions, TorXakis is able to find certain counterexamples much faster than LearnLib and JTorX.

4 Conformance to a reference implementation

4.1 Conformance learning and conformance model-based testing

Figure 4 illustrates how we may use model synthesis for establishing conformance (i.e., behavior equivalence) 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., LearnLib). Now, given another implementation I , there are basically two things we can do. The first approach, which we call ‘conformance model-based testing’, is that we provide M_R as input to a model-based testing tool (e.g., JTorX). 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 input sequence that demonstrates the difference between

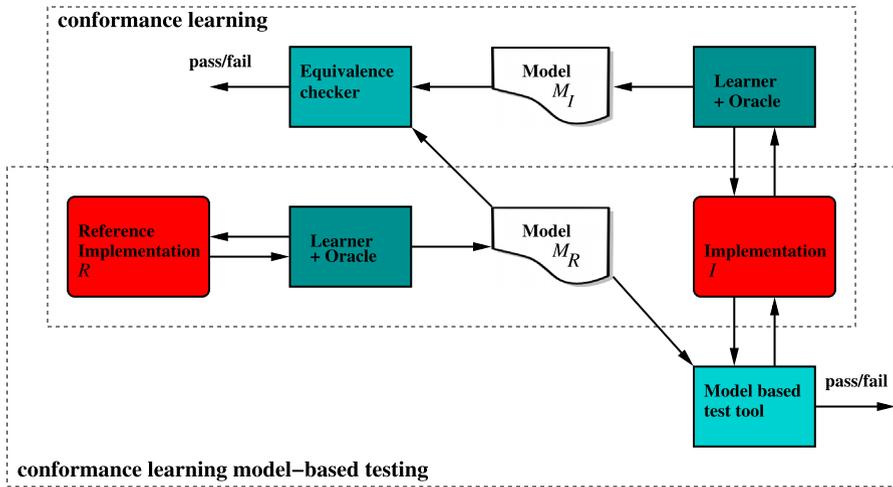


Fig. 4 Basic approaches of using automata learning to establish conformance of implementations. The learners interact with the implementations in order to construct models, which are then subsequently used for model-based testing or equivalence checking

I and M_R . The second, more ambitious approach, which we call ‘conformance learning’, is to use the learning tool to learn a model M_I of the other implementation I , and then use an equivalence checker to check observation equivalence of M_R and M_I . The equivalence checker will either output “pass”, meaning that the two models are equivalent, or “fail” together with an input sequence 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 of R or I .

In this paper, we use the CADP toolset to check observation equivalence of models.

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. It includes a wide range of verification techniques such as reachability analysis and compositional verification. CADP is used in this paper to check strong bisimulation equivalence of labeled transition systems. Two behavior deterministic Mealy machines are observation equivalent iff their associated labeled transition systems are strong bisimulation equivalent.

4.2 Conformance learning with a conformance oracle

The model-based testing (oracle) part of automata learning can be time consuming in practice. We therefore experimented with an alternative approach in which the model M_R of the reference implementation R is used as an oracle when learning a model for an implementation I . We will see that this use of what we call a *conformance oracle* may significantly speed up the learning process.

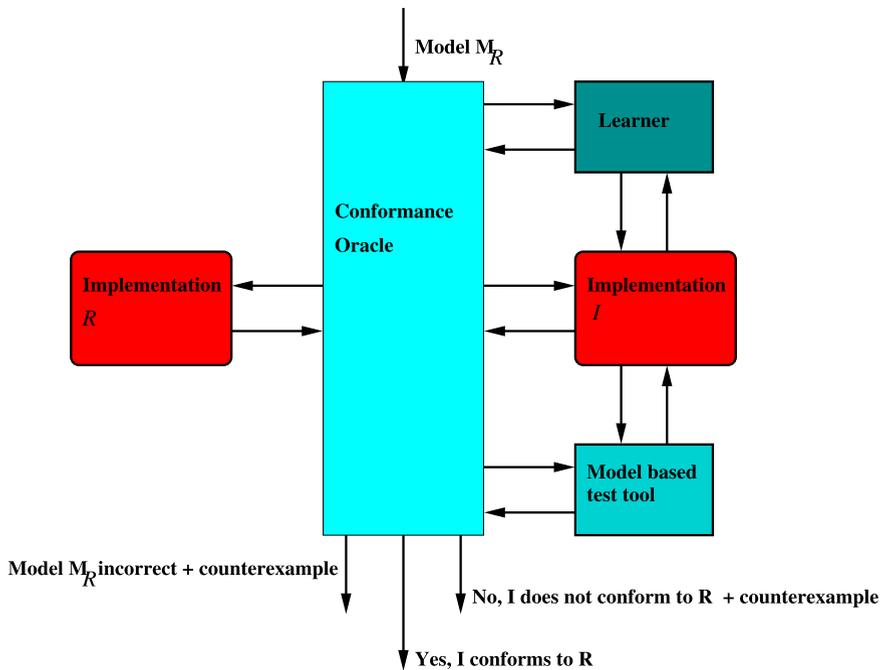


Fig. 5 Conformance oracle. The oracle takes a previously learned model M_R for the reference R as input and uses it to quickly answer equivalence queries by testing the equivalence of M_R with the hypothesized model M_I for the implementation I . When they are found to be inequivalent, the discovered counterexample is provided as input to I and R in order to test whether it proves the inequivalence of R and I , M_I and I , or M_R and R

Suppose a learner has constructed an hypothesized Mealy machine model M_I for implementation I . We want to use the availability of M_R to speed up the validation (or counterexample discovery) for M_I , and reduce the use of the model-based test oracle as much as possible. Our approach works as follows:

1. We first use an equivalence checker to test $M_R \approx M_I$. If so, then we use a model-based test tool to further increase our confidence that M_I is a good model of I . If model-based testing reveals no counterexamples we are done, otherwise the learner may use a produced counterexample to construct a new model of I , and we return to step (1).
2. If $M_R \not\approx M_I$ then the equivalence checker produces an input sequence u such that $out_{M_R}(u) \neq out_{M_I}(u)$. We apply u to both implementations R and I , and write $out_R(u)$ and $out_I(u)$, respectively, for the resulting output sequences.
3. If $out_R(u) \neq out_{M_R}(u)$ then model M_R is incorrect and we may use counterexample u to construct a new model for R .
4. Otherwise, if $out_I(u) \neq out_{M_I}(u)$ then model M_I is incorrect. In this case the learner may use counterexample u to construct a new model for I , and we return to step (1).
5. Otherwise, we have identified an observable difference between implementations R and I , i.e., I is not conforming to reference implementation R .

Figure 5 illustrates the architectural embedding of a conformance oracle.

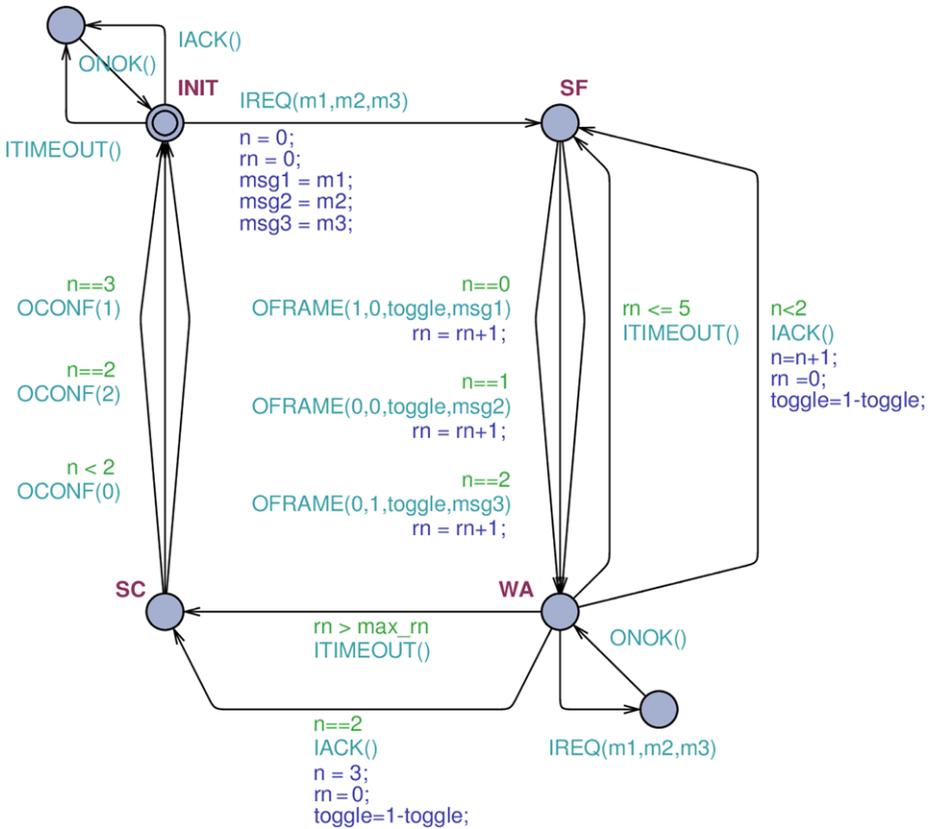


Fig. 6 Reference implementation of the BRP sender. The input symbols start with *I*, the output symbols start with *O*. In addition to symbols, the transitions contain value checks (or guards, ==, <, >) and assignments (=)

5 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 of the sender 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 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 6 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 Sect. 3). 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:
 - $i = 0$: the request has not been dispatched completely,
 - $i = 1$: the request has been dispatched successfully,
 - $i = 2$: 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 Fig. 6, 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 that have been acknowledged, and an integer variable rn to record 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 (see Appendix A):

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 the 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.

6 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.¹ Since all BRP implementations behave as Mealy machines, this is the type of state machine that we infer with our approach. In our experiments 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. Moreover, we have the following outputs: ONOK, OFRAME(b_1, b_2, b_3, m), where b_1, b_2, b_3 and m can be either 0 or 1, and OCONF(i), where $0 \leq i \leq 2$. In order to approximate equivalence queries, LearnLib provides several algorithms. We used the LearnLib test suite with randomly generated test traces containing 100 inputs.

The results of the inference of the reference implementation and the six mutants are shown in Table 2. For every implementation, we list the number of states in the learned model, as well as the average total number of Membership Queries (MQ). Moreover, we list the average total number of Test Traces (TT) generated for approximating equivalence queries. Note that these numbers do not include the last equivalence query, in which no counterexample has been found. Using CADP, we verified that all the learned models indeed are correct, i.e., equivalent to the Uppaal models described in Sect. 5. Each experiment was repeated 10 times with different seeds for the equivalence queries. For each measured value its average over the 10 experiments is listed in the table together with the standard deviation. If an experiment did not succeed to learn the model within two hours we aborted the experiment. In the last row we display how many of the 10 experiments did succeed. Even if an experiment did fail, we still use its numbers in calculating the average and the standard deviation, giving a lower bound for the real average and standard deviation for this experiment.

If we take a closer look at Table 2, we observe some interesting peculiarities. First, the number of test traces 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. Whenever mutant 1 receives a new request, the previous request is discarded and the sender starts handling the new request. 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. The probability that LearnLib selects (uniformly at random) six successive

¹In previous work (Aarts et al. 2012b) we used TCP/IP socket communication. TCP/IP uses optimizations, TCP delayed acknowledgment technique and Nagle's algorithm, for reducing packet overhead. These optimizations slow down the communication in a setting with very small messages. In normal communication between SUT and learner the communication is an alternating pattern of sending input followed by a returned output. However in between two queries to the SUT an extra RESET input is sent, which breaks this alternating pattern. Exactly at that point the optimizations in TCP/IP cause a delay in communication to happen. Effectively this means that *each* query to the SUT gets an extra delay. By disabling the Nagle optimization in the TCP/IP socket communication we can prevent these delays to happen. For even better performance we used direct method calls to the SUT by linking the SUT code against our learner. Removing this delay made the queries in the experiments much faster and therefore it was possible to learn the BRP models for larger retransmission counter and variable ranges in the same amount of time than in Aarts et al. (2012b). Moreover because membership queries in general are much shorter than equivalence testing queries, the performance gain per query in the first are much bigger than in the latter.

Table 2 Learning statistics for the BRP reference implementation and mutants 1–6. **MQ** refers to the average total number of Membership Queries, **TT** to the average total number of Test Traces. **vrX-Y** means $m_i \in [X \dots Y]$, where m_i is a message in $\text{IREQ}(m_1, m_2, m_3)$, e.g. vr0-1 allows values 0 and 1 in a message in IREQ

		Reffimpl	Mut1	Mut2	Mut3	Mut4	Mut5	Mut6
vr0-1	states	156	156	128	156	156	156	136
	avg. MQ	24998	22174	19618	22488	24684	24998	21662
	std.dev.	1717	627	1006	769	1828	1717	736
	avg. TT	14	5830	7	15	14	14	14
	std.dev.	13	5142	6	13	13	13	13
	Succeeded	10/10	10/10	10/10	10/10	10/10	10/10	10/10

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 Sect. 7. 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, see Figs. 7–12. 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. This will be explored further in the next section.

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, because more inputs need to be considered. Therefore, we ran 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, because no changes to the general structure of handling requests have been made). We evaluated how LearnLib performs for different maximal values for the retransmission counter **rn**. Moreover, we investigated what happens when increase the value range **vr** for each message parameter.

Table 3 and 4 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 message parameters m_1, m_2 , and m_3 . As expected, increasing the number of retransmissions **rn** and the value ranges **vr** both results in bigger models for which more membership and test traces are needed and, accordingly, more time for learning and testing. Increasing the value range leads to a fast growth in membership queries required to construct a hypothesis and the number of test traces required to find counterexamples for incorrect hypotheses, because the model to learn contains more states and the probability to select an ITIMEOUT or IACK input (which is needed in a counterexample) shrinks. For example, compare the results for rn15 and vr0-1 versus rn15 and vr0-2 in Table 3. Increasing the maximal number of retransmissions leads to a linear growth of membership queries required to construct a hypothesis, but to a fast growth of test traces required to find counterexamples for incorrect hypotheses, because the higher the retransmission counter, the more ITIMEOUT inputs are needed in a counterexample. For example, compare the results for rn15 to rn20 for vr0-1 in Table 3.

For mutant 1, the time needed for testing increases so fast that if the maximal number of retransmissions is 8 and there are 2 values, not every seed results in a correct model within 2 hours. In contrast, for 2 values the reference implementation only starts failing to learn

Table 3 Learning statistics for reference implementation. **MQ** refers to the average total number of Membership Queries, **TT** to the average total number of Test Traces. **vrX-Y** means $m_i \in [X \dots Y]$, where m_i is a message in $IREQ(m_1, m_2, m_3)$, e.g. vr0-3 allows values 0, 1, 2, and 3 in a message in $IREQ$. Similar, **rn** refers to the value of the retransmission counter in the model. Changing the number of retransmissions is done by increasing the value in the guard statement ($rn > 5$ and $rn \leq 5$) for the ITIMEOUT input from state WA to SF and from WA to SC

		———— increasing retransmission counter ———>					
		rn15	rn16	rn17	rn18	rn19	rn20
vr0-1	states	436	464	492	520	548	576
	avg. MQ	83346	91949	99465	107206	108582	104113
	std.dev.	4122	5807	4298	13624	6402	33896
	avg. TT	20699	82854	103041	137750	295292	2665735
	std.dev.	21865	129195	130379	180024	284401	3044574
	Success	10/10	10/10	10/10	10/10	10/10	9/10
		rn09	rn10	rn12	rn13	rn14	rn15
vr0-2	states	730	808	964	1042	1120	1198
	avg. MQ	755837	853000	1037250	1124192	1211588	564335
	std.dev.	13562	18753	31762	37755	29241	603545
	avg. TT	7507	20944	348407	1349612	2395184	7903428
	std.dev.	5219	14340	460829	1882704	2608484	2833324
	Succeeded	10/10	10/10	10/10	10/10	10/10	4/10
		rn06	rn07	rn08	rn09	rn10	rn11
vr0-3	states	1052	1220	1388	1556	1724	1892
	avg. MQ	4971425	5789485	6641683	7445564	5975070	2433575
	std.dev.	34020	84075	84468	105242	3413383	3341094
	avg. TT	2587	14928	111122	502686	3795812	8129711
	std.dev.	1682	9797	116094	545382	3528212	3376705
	Succeeded	10/10	10/10	10/10	10/10	7/10	2/10
		rn03	rn04	rn05	rn06	rn07	rn08
vr0-4	states	994	1304	1614	1924	2234	2544
	avg. MQ	16537318	21810721	27159739	32278521	37677908	27794859
	std.dev.	56460	106048	102494	171051	277996	18521581
	avg. TT	230	1392	13057	148571	1375285	4732741
	std.dev.	246	1089	13744	162899	1056963	3445873
	Succeeded	10/10	10/10	10/10	10/10	10/10	6/10
		rn02	rn03	rn04	rn05	rn06	
vr0-5	states	1120	1636	2152	2668	3184	
	avg. MQ	53959581	79140421	104476825	129818719	130303196	
	std.dev.	0	106997	214990	232653	43041807	
	avg. TT	106	1353	16010	98888	2591811	
	std.dev.	77	1287	11680	116544	1743665	
	Succeeded	10/10	10/10	10/10	10/10	6/10	
		rn02	rn03	rn04			
vr0-6	states	1712	2510	3308			
	avg. MQ	205602133	302303504	353391698			
	std.dev.	177195	259788	51494449			
	avg. TT	294	6044	105437			
	std.dev.	378	5491	65300			
	Succeeded	10/10	10/10	5/10			

———— increasing value range ———>

Table 4 Learning statistics for mutant 1. **MQ** refers to the average total number of Membership Queries, **TT** to the average total number of Test Traces. **vrX-Y** means $m_i \in [X \dots Y]$, where m_i is a message in $\text{IREQ}(m_1, m_2, m_3)$, e.g. vr0-3 allows values 0, 1, 2, and 3 in a message in IREQ . Similar, **rn** refers to the value of the retransmission counter in the model. Changing the number of retransmissions is done by increasing the value in the guard statement ($\text{rn} > 5$ and $\text{rn} \leq 5$) for the ITIMEOUT input from state WA to SF and from WA to SC

		—— increasing retransmission counter ——>					
		rn03	rn04	rn05	rn06	rn07	rn08
vr0-1	states	100	128	156	184	212	240
	avg. MQ	13214	18199	22174	27999	32256	27365
	std.dev.	402	515	627	738	850	16821
	avg. TT	154	1250	5830	61495	667761	7903190
	std.dev.	129	1615	5142	78465	516786	3371144
	Succeeded	10/10	10/10	10/10	10/10	10/10	6/10
			rn02	rn03	rn04	rn05	
vr0-2	states	184	262	340	418		
	avg. MQ	171318	243928	326408	334443		
	std.dev.	1603	2281	2960	131246		
	avg. TT	347	9655	278077	4388168		
	std.dev.	331	8693	159918	2974071		
	Succeeded	10/10	10/10	10/10	8/10		
			rn02	rn03	rn04		
vr0-3	states	380	548	716			
	avg. MQ	1730589	2495661	1272022			
	std.dev.	0	0	1017988			
	avg. TT	3312	224096	6522672			
	std.dev.	3031	246072	2488837			
	Succeeded	10/10	10/10	2/10			

—— increasing value range ——
↓

the correct model within hours for all seeds for the maximal number of retransmissions of 20. As mentioned before, the reason for this is that mutant 1 also accepts new requests in state WA and discards previous requests. Also in the case where the maximal number of retransmissions is 5 and there are 3 values, LearnLib is not able to construct a correct model for all seeds for mutant 1 within 2 hours. This is not surprising, because in both cases the probability to select a counterexample is even lower than for mutant 1 in Table 2. Once LearnLib fails to learn a correct model, we assume that it will also fail for larger value ranges of the parameters. The same holds for the retransmission counter. Once some of the experiments fail, the number of successful runs with a higher retransmission counter will decrease.

Conformance checking We compare the two methods, described in Sect. 4.1, for establishing the conformance of the mutant implementations to the reference implementation of BRP. We only consider the versions of the models with at most 5 retransmissions and 2 different parameter values.

Table 5 Counterexamples found by equivalence checking of mutant models and reference implementation (IT = ITIMEOUT, IR = IREQ, IA = IACK, OF = OFRAME)

	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()	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()	OCONF(0)	OCONF(2)
Mut6	IR(0,0,1) IA() IA()	OF(0,1,0,0)	OF(0,1,0,1)

Table 6 Conformance testing the mutants with JTorX for retransmission counter 5 and value range 0-1: using the reference model JTorX will generate a test sequence and applies it to the mutant implementation in order to establish the conformance between the reference model and the mutant. For all mutants it detected the non-conformance and returned a counterexample. The table shows the average number of IO symbols, i.e. the average number of inputs and outputs, needed to find the counterexample for each mutant

		mut1	mut2	mut3	mut4	mut5	mut6
vr0-1	avg. IO symbols	5	894	19	37	6657	198
rn05	std.dev.	1	768	12	20	4983	268

The first method, conformance learning, used the CADP (bisimulation) equivalence checker to compare the model M_I , that we learned for the mutant implementations I , with the model M_R learned for the reference implementation R .² 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 (for each mutant it takes around 3 seconds to find the counterexample). The counterexamples found by CADP are depicted in Table 5.

The second method, conformance model-based testing, 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 average number of IO symbols, i.e. the average number of inputs and outputs in the counterexample, is shown in Table 6 for the different mutant implementations. Because JTorX generates a single long testing sequence without intermediate resets, the resulting counterexamples are rather long sequences and therefore are not shown in the table.

In a JTorX experiment the performance is measured in number of IO symbols whereas in a Learnlib experiment the performance is measured in number of membership and testing queries. Comparing the measurements of both experiments is not trivial. Although each testing query corresponds to 200 IO symbols, since we use a random test suite generating test traces with 100 input symbols, each membership query varies in length determined by the learning algorithm. However, in a crude approximation we could say that in the learning experiment $\#IO\ symbols \approx x * MQ + 200 * TT$ where x is a guessed factor of the average length of the membership queries.

²Essentially the same counterexamples were also found using the JTorX iocoChecker. The JTorX iocoChecker, which is distributed as an extra feature of the JTorX model-based testing tool (Belinfante 2010), is a tool to check analytically whether two models are ioco-related.

Table 7 Conformance checking using the equivalence checker CADP for retransmission counter 10 and value range 0-1. **MQ** refers to the average total number of Membership Queries, **TT** to the average total number of Test Traces. The table shows the learning statistics for reference implementation and mutants 3–6. After learning the reference and mutant models we could find in around 3 seconds a counterexample trace illustrating the difference between the model of the mutant and the model of the reference implementation

		RefImpl	Mut3	Mut4	Mut5	Mut6
vr0-1	states	296	296	296	296	256
rn10	avg. MQ	55417	46509	55417	55417	47166
	std.dev.	3767	1899	3767	3767	3083
	avg. TT	280	357	280	280	280
	std.dev.	171	266	171	171	171
	Succeeded	10/10	10/10	10/10	10/10	10/10

Table 8 Conformance testing the mutants with JTorX for retransmission counter 10 and value range 0-1: using the reference model JTorX will generate a test sequence and applies it to the mutant implementation in order to establish the conformance between the reference model and the mutant. For all mutants it detected the non-conformance and returned a counterexample. The table shows the average number of IO symbols, i.e. the average number of inputs and outputs, needed to find the counterexample for each mutant

		mut1	mut3	mut4	mut5	mut6
vr0-1	avg. IO symbols	5	19	37	109154	178
rn10	std.dev.	1	12	20	90617	228

If we look at Table 6 we immediately see that the number of IO symbols needed for mutant 5 is much bigger than for the other mutants. However, Table 2 shows that learning mutant 5 is not more difficult than learning the other mutants. This can be explained by the fact that if we look at the counterexample for mutant 5 in Table 5 we immediately see that the counterexample for mutant 5 is the longest. When looking at the model for mutant 5 we can see that the number of ITIMEOUT inputs in the counterexample is directly related to the value of the retransmission counter **rn**.

To investigate the effects, we increase the retransmission counter **rn** to 10. The learning results for $rn = 10$ are shown in Table 7 for the different mutants and for conformance testing the mutants with JTorX in Table 8. Note that in learning the mutants for the retransmission counter of 10 we skipped mutant 1, because it was already shown in Table 2 that it couldn't be learned in 2 hours, and we skipped mutant 2 because the only difference between mutant 2 and the reference implementation is a different retransmission number.

After learning the new mutants CADP could again find in around 3 seconds a counterexample trace illustrating the difference between the models of the mutants and the model of the reference implementation. The results in Table 7 show that learning mutant 5 with a larger retransmission counter still is not more difficult than learning the other mutants, whereas Table 8 shows that for mutant 5 model-based testing still requires much longer traces than for the other mutants. However, when comparing Table 7 and Table 8 it shows that for all mutants the number of IO symbols is only increased for mutant 5.

Again, this can be explained by the fact that if we look at the counterexample for mutant 5 in Table 5 we immediately see that the number of ITIMEOUT inputs in the counterexample is directly related to the value of the retransmission counter **rn**. By increasing the retransmission counter **rn**, the counterexample for mutant 5 becomes longer, and since finding longer

counterexamples takes longer, this increases the time required by JTorX to find a counterexample. In contrast, for mutants 1, 3, 4 and 6 the counterexamples have no ITIMEOUT inputs and are independent of the retransmission counter **rn**, which explains why in the experiments for these mutants the number of IO symbols for finding counterexamples doesn't increase for a larger retransmission counter **rn**.

Thus by comparing Table 7 and Table 8 we see that the increase in the number of IO symbols needed for conformance testing of mutant 5 is big. However, this number of IO symbols needed for conformance testing of mutant 5 is comparable in size to the number of membership and testing queries for learning mutant 5: compare 109154 IO symbols from JTorX to $111417 = 1 \cdot 55417 + 200 \cdot 280$ ($\#IO\ symbols \approx x \cdot MQ + 200 \cdot TT$) IO symbols from LearnLib, where we assume a lower bound $x = 1$ on the average length of membership queries.

Since LearnLib tests hypotheses in a way similar to JTorX and because the counterexample of mutant 5 reaches the deepest part of the model, testing during learning the model with LearnLib of mutant 5 and conformance testing with JTorX of mutant 5 become even difficult. Therefore their performance in both experiments becomes comparable.

In general, however, learning models of proposed implementations takes more time than model-based testing them but also provides more information in the form of a learned model. Even in these cases, it may still be beneficial to use learning tools since the learned models can for instance be used for model checking analysis.

7 Further analysis and improvements

7.1 Why random testing sometimes fails

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 states that provide a counterexample 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.

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 (common in Markov decision processes (Puterman 1994)). Such a logical statement can be expressed in a probabilistic branching-time logic PCTL (Hansson and Jonsson 1994) or CSL (Aziz et al. 1996). The probabilistic models may also contain reward functions and bounds on these can be checked in combination with the time and probability values.

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.

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

Table 9 Equivalence query statistics for mutant 1 with TorXakis. $\mathbf{vrX-Y}$ means $m_i \in [X \dots Y]$, where m_i is a message in $\text{IREQ}(m_1, m_2, m_3)$, e.g. vr0-2 allows values 0, 1, and 2 in a message in IREQ . Similar, \mathbf{rn} refers to the value of the retransmission counter in the model. Changing the number of retransmissions is done by increasing the value in the guard statement ($\text{rn} > 5$ and $\text{rn} \leq 5$) for the ITIMEOUT input from state WA to SF and from WA to SC

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

outgoing $\text{OCONF}(0)$ transition is 0.0247121. This means the probability of reaching a state with an outgoing $\text{OCONF}(0)$ transition within 75 test runs is 0.847. This result explains why LearnLib requires very few test runs to learn a correct model of this system, which it does within seconds. 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 $\text{OCONF}(0)$ transition is only 0.0000010. Hence, finding a counterexample by random testing will take much longer for mutant 1 than for the reference, explaining why LearnLib needs 667761 (see Table 4) test runs on average to find this counterexample ($0.999999^{667761} \approx 0.51$ is the probability of not finding this trace in 667761 tries).

7.2 Using abstraction to speed up testing

A simple way to increase the probability of finding counterexamples for mutant 1 within a short time is to increase the probability of ITIMEOUT inputs, for instance by assigning an equal probability of $1/3$ to the ITIMEOUT , IACK and IREQ inputs. Symbolic testing tools and abstraction learners will use this distribution, because they initially assume that the parameter values are not as important as the input types. TorXakis and Tomte (see Sect. 3) are such tools and we therefore evaluated the effect of using these instead of the basic random testing implemented in LearnLib.

TorXakis For the experiments above we employed LearnLib for both the learning and testing phase. However, it is also possible to perform the equivalence check by an external tool. We experimented with the LTS-based model-based testing tool TorXakis to see whether we can improve on detecting incorrect hypothesis models.

Table 9 summarizes the results obtained with TorXakis when testing mutant 1 against the hypothesized LearnLib model for $\text{rn5}, \text{vr0} - 1, \text{rn7}, \text{vr0} - 1$ and $\text{rn5}, \text{vr0} - 2$. In addition, the results for the scenarios $\text{rn5}, \text{vr0} - 9$ and $\text{rn7}, \text{vr0} - 2$ are presented. LearnLib did not manage to find a counterexample in these cases. The numbers in Table 9 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. We do not measure the timing, because it makes no sense for TorXakis: TorXakis explicitly tests for non-occurrence of outputs by means of a time-out, and while the time value chosen for this time-out is in some sense arbitrary, it has a very strong influence on the total duration of a test.

TorXakis is able to detect counterexamples for the incorrect hypothesized models for $\text{rn5}, \text{vr0} - 1, \text{rn7}, \text{vr0} - 1$, and for $\text{rn5}, \text{vr0} - 2$. Only after the IREQ input has been selected, the message values are randomly selected. This implies that increasing the domain of possible message values does not increase the length of the test case required to detect the

Table 10 Learning statistics for mutant 1 using Tomte. $\mathbf{vrX-Y}$ means $m_i \in [X \dots Y]$, where m_i is a message in $\text{IREQ}(m_1, m_2, m_3)$, e.g. vr0-1 allows values 0 and 1 in a message in IREQ . \mathbf{rn} refers to the value of the retransmission counter in the model. Changing the number of retransmissions is done by increasing the value in the guard statement ($\text{rn} > 5$ and $\text{rn} \leq 5$) for the ITIMEOUT input from state WA to SF and from WA to SC

		rn09	rn10	rn12	rn13	rn14	rn15
$\mathbf{vr0-\infty}$	states	62	68	80	86	92	98
	avg. MQ	1518	1875	2449	2629	2196	1377
	std.dev.	3	4	3	4	1372	1564
	avg. TT	1175	3525	51589	89133	182508	258552
	std.dev.	1998	2816	34687	33648	82534	78432
	Succeeded	10/10	10/10	10/10	10/10	7/10	4/10

counterexample. Combined with the fact that TorXakis generates one very long test case, it is able to find a counterexample for the scenario $\text{rn7}, \text{vr0} - 2$ within reasonable time. In conclusion, TorXakis is able to detect counterexamples within reasonable time, which LearnLib could not detect.

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 such as 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 10 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 4) and, moreover, Tomte can handle a larger value range than LearnLib: compare the results vr0-\infty used by Tomte to vr0-3 used by LearnLib. Work is underway to extend Tomte so that it can also learn the BRP reference implementation and the other mutants.

7.3 Conformance learning with a conformance oracle

Above, we demonstrated how to make random testing a little smarter by making use of abstractions. This results in a speed up in the time random testing requires to find a counterexample, but even this method has its limits as shown in Table 10. We now show that we can remove random testing altogether using the notion of a conformance oracle using the method ‘Conformance learning with a conformance oracle’ described in Sect. 4.2, in which we use CADP as an equivalence checker.

The results of our experiments can be found in Table 11 in which for each mutant we found non-conformance with the reference implementation. The number of states listed are the number of states of the last hypothesis found in step 5 of the conformance learning with a conformance oracle algorithm described in Sect. 4.2. Compared with Table 10, we only have the number of states and the number of membership queries listed, because when using a conformance oracle we do no random testing after learning anymore. Thus, by using a conformance oracle instead of random testing, the learning process becomes completely deterministic and, therefore, we only had to do each experiment in Table 11 once.

Table 11 Finding counterexamples using a conformance oracle, case $rn = 5$. In fact we are learning the mutant using the reference model as a conformance oracle until we found a counterexample which proves both the learned mutant model and the mutant implementation different from the reference implementation

		mut1	mut2	mut3	mut4	mut5	mut6
vr0-1	states	16	127	16	16	155	20
rn05	MQ	1771	17798	1771	1771	21718	2211

Table 12 Finding a counterexample for mutant 5 using a conformance oracle. In fact we are learning the mutant using the reference model as a conformance oracle until we found a counterexample which proves both the learned mutant model and the mutant implementation different from the reference implementation

		rn09	rn10	rn11	rn12	rn13	rn14
vr0-1	states	267	295	323	351	379	407
	MQ	42751	50189	54949	63227	68267	77385

Table 13 Conformance testing mutant 5 using JTorX. $vrX-Y$ means $m_i \in [X \dots Y]$, where m_i is a message in $IREQ(m_1, m_2, m_3)$, e.g. vr0-1 allows values 0 and 1 in a message in $IREQ$. rn refers to the value of the retransmission counter in the model. Changing the number of retransmissions is done by increasing the value in the guard statement ($rn > 5$ and $rn \leq 5$) for the $ITIMEOUT$ input from state WA to SF and from WA to SC

		rn09	rn10	rn11	rn12	rn13	rn14
vr0-1	avg. IO symbols	54701	109154	141285	475398	497128	1066146
	std.dev.	39385	90617	95454	475627	541333	803093

Table 11 shows that finding a counterexample using the conformance oracle is the most difficult for mutants 2 and 5, however, it is nearly trivial for the other four mutants, because for these mutants the final hypothesis only has between 16–20 states. We further investigate mutant 5 by scaling up the rn parameter in order to discover the limits of using the conformance oracle setup and to compare its performance to that of a state-of-the-art model-based testing tool.

Table 5 shows that for mutant 5 the counterexample does depend on the rn parameter: with a higher rn value, more $ITIMEOUT$ inputs are required in the counterexample for proving mutant 5 to be different from the reference implementation with the same rn value. We investigate mutant 5 by doing both conformance model-based testing and conformance learning using a conformance oracle for increasing rn values as shown in Tables 12 and 13.

Testing using a conformance oracle is really quick: the experiment typically runs within a few seconds. Furthermore, the number of membership queries required to build the hypothesis seems to increase only linearly with increasing rn . In contrast, as Table 13 shows, the number of IO symbols and its related testing time required by JTorX increases much faster. The reason for this difference between the two methods is that for conformance testing the time required depends on the probability of reaching a state that provides a counterexample, while learning using a conformance oracle is fully deterministic and thus is independent of this probability.

8 Conclusion

We show how to apply active state machine learning methods to a real-world use case from software engineering: 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 (LearnLib and Tomte) 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 use 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. All implementations were treated as black-box software systems. The goal of our experiments was to discover how active learning tools can be used to establish the conformance between the mutant and reference implementations. The results of these experiments can be summarized as follows:

- The problem of test selection is a big bottleneck for state-of-the-art active learning tools. Existing model-based testing tools can be used to make this bottleneck less severe.
- Increasing the number of message values **vr** (the alphabet size) increases the time required for test selection as well as the time needed for finding a hypothesis (using membership queries).
- Increasing the maximal number of retransmissions **rn** (the length of counterexamples) increases the time required for test selection much more than the time needed for finding a hypothesis.
- Establishing conformance using an equivalence checker and two learned models is very fast. Using only a single learned model and a model-based testing tool is also fast, but can run into problems because test selection takes much longer than equivalence checking.
- Interestingly, there are cases where learning a mutant model (and checking equivalence) is as fast as model-based conformance testing. In general, however, learning a single model and subsequent testing is faster than learning two models.

Furthermore, we noticed in our experiments that 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 symbolic evaluation of equivalence queries, using a new learning method based on abstraction refinement (Tomte), and introducing a new way of learning based on the novel concept of a conformance oracle. Such a conformance oracle effectively learns two models at once and uses the (partially) learned models in an equivalence checker to quickly answer equivalence queries asked by either learner. When the models are similar, this can greatly reduce the cost of learning.

The concept of a conformance oracle opens up several interesting directions for future work. In particular, since it can also be used as a model-based tester, it would be interesting to further investigate exactly when and why it can be used to establish conformance more quickly than state-of-the-art model-based test tools. Our study already found one such case in mutant 5, where tools based on random testing are troubled by the low probability of reaching a state that leads to a counterexample. A conformance oracle in combination with an active learning tool finds the same counterexample deterministically and in fewer steps.

The concept is also closely linked to transfer learning: it can use a previously learned model to speed up the process of learning a new (similar) model. A conformance oracle, however, can transfer these models during the learning process itself, making it an interesting approach for distributed learning settings.

The BRP use case, including the models as well as all the scripts used to link the different tools together, have been made available on-line for testing and learning by fellow researchers at <http://www.italia.cs.ru.nl>.

Acknowledgements Supported by STW project 11763 Integrating Testing And Learning of Interface Automata (ITALIA).

We thank Colin de la Higuera and Bernard Steffen for suggesting to use the learned reference model for answering equivalence queries, and Axel Belinfante for assisting us with JTorX.

Appendix A: Mutants of the BRP sender

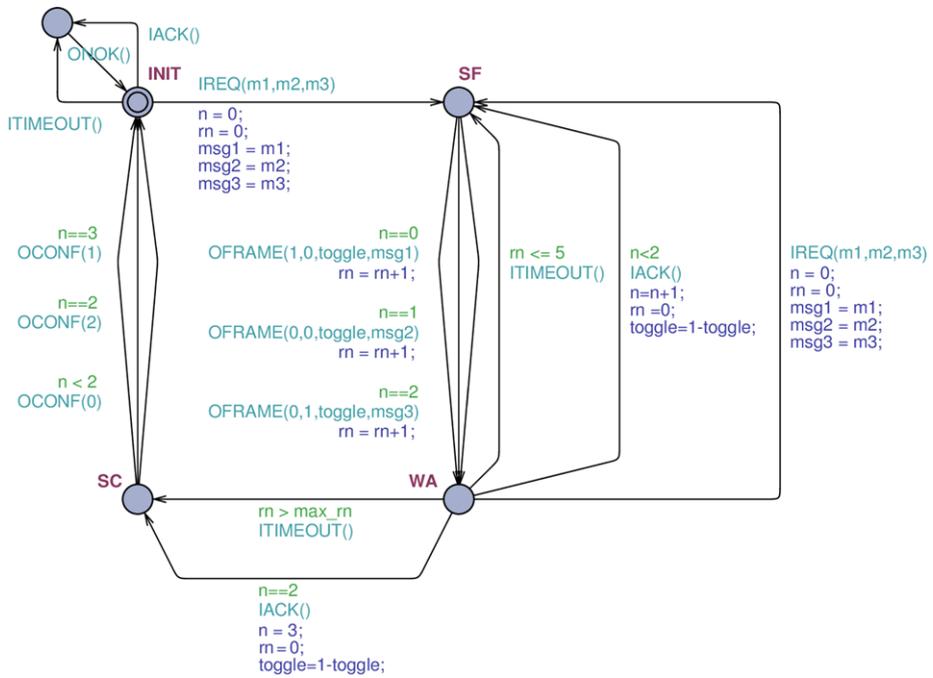


Fig. 7 Mutant 1

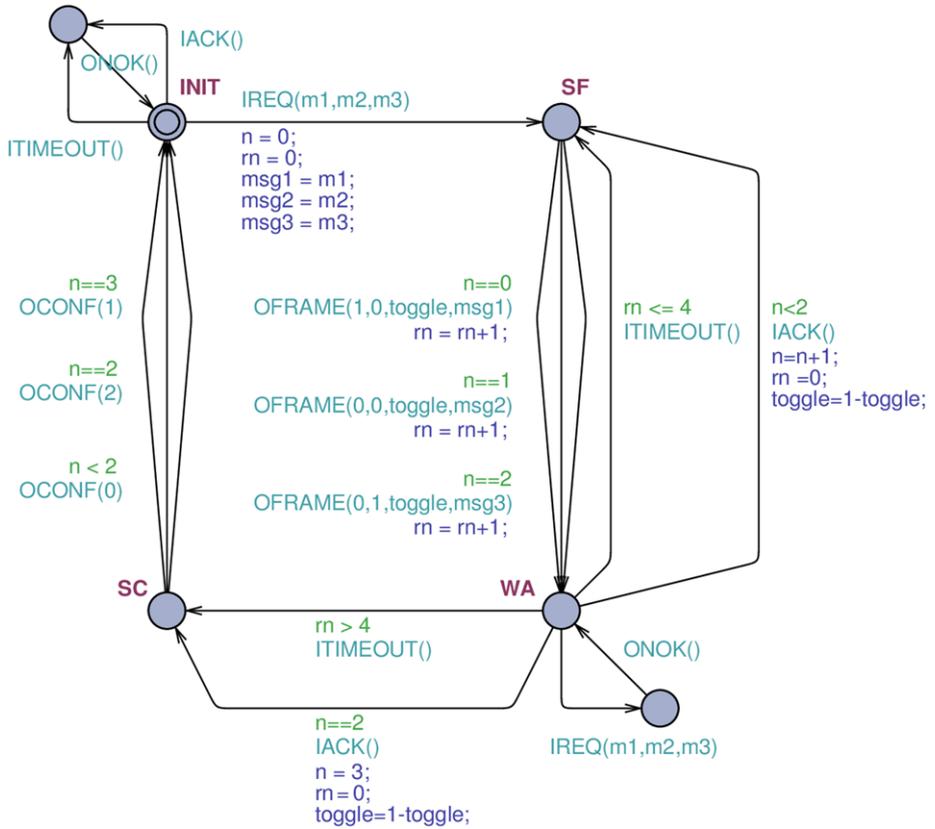


Fig. 8 Mutant 2

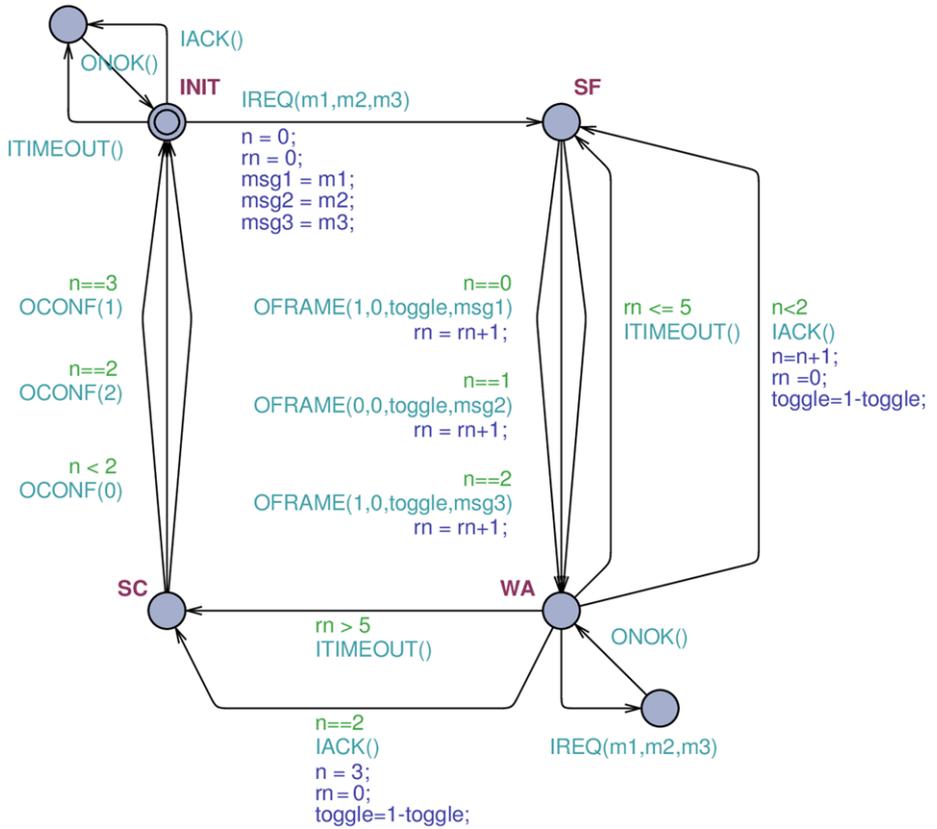


Fig. 10 Mutant 4

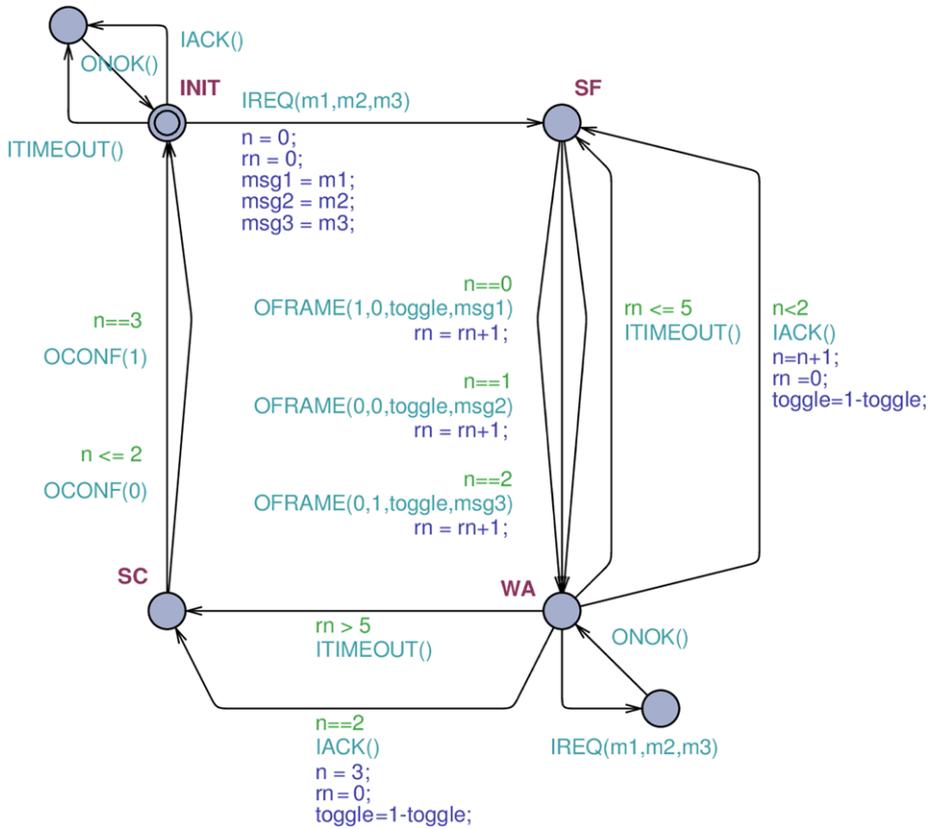


Fig. 11 Mutant 5

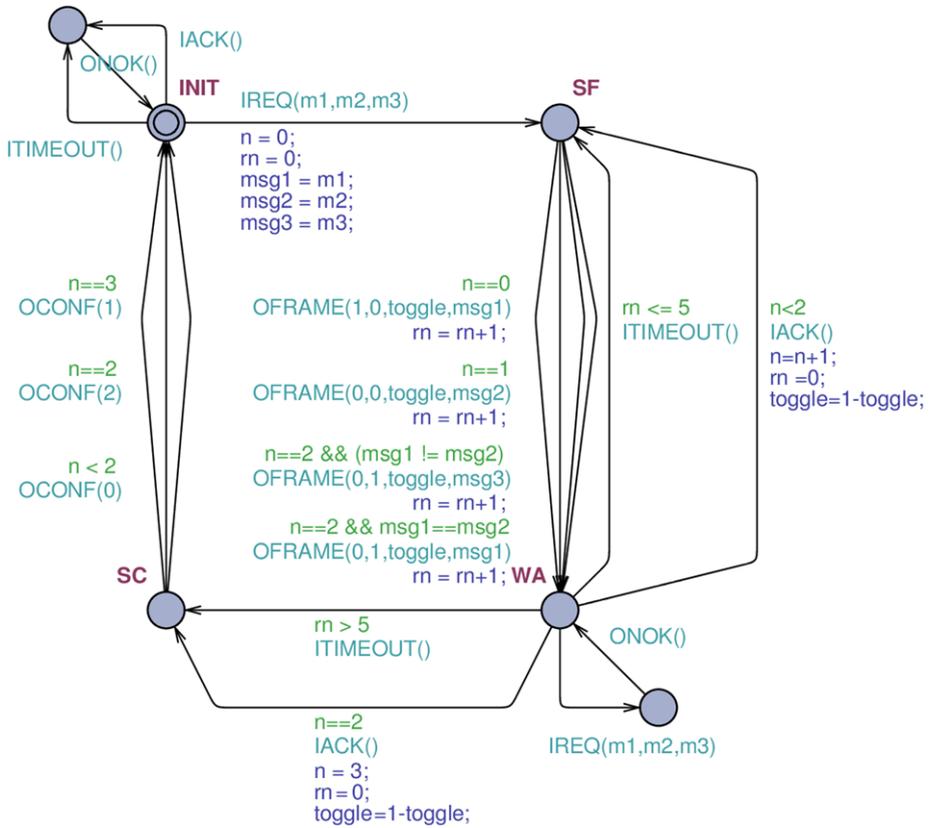


Fig. 12 Mutant 6

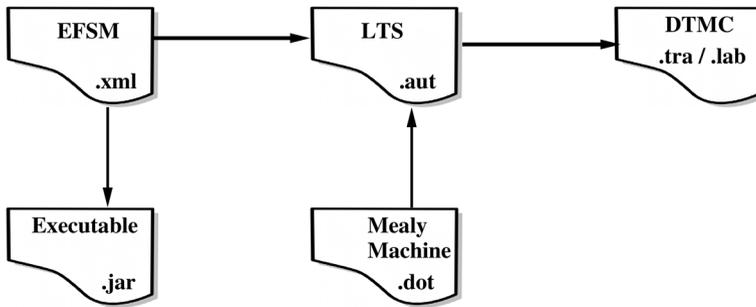


Fig. 13 Formats for representing state machines and implemented conversions

Appendix B: Conversions between input formats

Figure 13 summarizes the various representations of state machines that we use in this paper, and the conversions between these formats that we have implemented.

The Mealy machine models learned by LearnLib are 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 $q'' \xrightarrow{o} q'$.

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.

We use JTorX to establish conformance of mutant implementations to a model of the reference implementation, represented as an `.aut` file.

CADP is used in this paper to check strong bisimulation equivalence of labeled transition systems represented as `.aut` files.

References

- Aarts, F., & Vaandrager, F. (2010). Learning I/O automata. In *Lecture notes in computer science: Vol. 6269. Proceedings of the 21st international conference on concurrency theory, CONCUR* (pp. 71–85). Berlin: Springer.
- Aarts, F., Jonsson, B., & Uijen, J. (2010a). Generating models of infinite-state communication protocols using regular inference with abstraction. In *Proceedings of the 22nd IFIP WG 6.1 international conference on testing software and systems, ICTSS'10* (pp. 188–204). Berlin: Springer.
- Aarts, F., Schmaltz, J., & Vaandrager, F. (2010b). Inference and abstraction of the biometric passport. In *Proceedings of the 4th international conference on leveraging applications of formal methods, verification, and validation—volume part I* (pp. 673–686). Berlin: Springer.
- Aarts, F., Heidarian, F., Kuppens, H., Olsen, P., & Vaandrager, F. (2012a). Automata learning through counterexample-guided abstraction refinement. In *Lecture notes in computer science: Vol. 7436. Proceedings of the 18th international symposium on formal methods (FM 2012)* (pp. 10–27). Berlin: Springer.
- Aarts, F., Kuppens, H., Tretmans, J., Vaandrager, F., & Verwer, S. (2012b). Learning and testing the bounded retransmission protocol. In *JMLR workshop and conference proceedings: Vol. 21. Proceedings of the 11th international conference on grammatical inference (ICGI 2012)* (pp. 4–18). JMLR.
- Alur, R., & Dill, D. L. (1994). A theory of timed automata. *Theoretical Computer Science*, 126, 183–235.
- Ammons, G., Bodik, R., & Larus, J. R. (2002). Mining specifications. In *Proceedings of the 29th symposium on principles of programming languages* (pp. 4–16). New York: ACM.
- Angluin, D. (1987). Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2), 87–106.

- Antunes, J., Neves, N., & Verissimo, P. (2011). Reverse engineering of protocols from network traces. In *Proceedings of the working conference on reverse engineering* (pp. 169–178).
- Aziz, A., Sanwal, K., Singhal, V., & Brayton, R. (1996). Verifying continuous time Markov chains. In *Lecture notes in computer science: Vol. 1102. Proceedings of 8th international conference on computer aided verification (CAV)* (pp. 269–276). Berlin: Springer.
- Balcázar, J., Diaz, J., Gavaldá, R., & Watanabe, O. (1997). Algorithms for learning finite automata from queries: a unified view. *Advances in Algorithms, Languages, and Complexity*, 53–72.
- Bartlett, K., Scantlebury, R., & Wilkinson, P. (1969). A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12, 260–261.
- Behrmann, G., David, A., & Larsen, K. (2004). A tutorial on Uppaal. In *Lecture notes in computer science: Vol. 3185. Formal methods for the design of real-time systems* (pp. 33–35). Berlin: Springer.
- Belinfante, A. (2010). Jtorx: a tool for on-line model-driven test derivation and execution. In *Lecture notes in computer science: Vol. 6015. Proceedings of the 16th international conference on tools and algorithms for the construction and analysis of systems (TACAS)* (pp. 266–270). Berlin: Springer.
- Berg, T., Grinchtein, O., Jonsson, B., Leucker, M., Raffelt, H., & Steffen, B. (2005). On the correspondence between conformance testing and regular inference. In *Lecture notes in computer science: Vol. 3442. Proceedings of the 8th international conference on fundamental approaches to software engineering (FASE)* (pp. 175–189). Berlin: Springer.
- Bertolino, A., Inverardi, P., Pelliccione, P., & Tivoli, M. (2009). Automatic synthesis of behavior protocols for composable web-services. In *Proceedings of the joint meeting of the 12th European software engineering conference and the 17th ACM SIGSOFT symposium on the foundations of software engineering* (pp. 141–150). New York: ACM.
- Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., & Pretschner, A. (Eds.) (2005). *Lecture notes in computer science: Vol. 3472. Model-based testing of reactive systems*. Berlin: Springer.
- Castro, J., & Gavaldá, R. (2008). Towards feasible PAC-learning of probabilistic deterministic finite automata. In *Lecture notes in computer science: Vol. 5278. Proceedings of the 9th international colloquium on grammatical inference: algorithms and applications (ICGI)* (pp. 163–174). Berlin: Springer.
- Cho, C. Y., Babic, D., Shin, E. C. R., & Song, D. (2010). Inference and analysis of formal models of botnet command and control protocols. In *Proceedings of the 17th ACM conference on computer and communications security* (pp. 426–439). New York: ACM.
- Clark, A., & Thollard, F. (2004). PAC-learnability of probabilistic deterministic finite state automata. *Journal of Machine Learning Research*, 473–497.
- Clarke, E. (1997). Model checking. In *Lecture notes in computer science: Vol. 1346. Proceedings of the 17th conference on foundations of software technology and theoretical computer science* (pp. 54–56). Berlin: Springer.
- Combe, D., de la Higuera, C., & Janodet, J.-C. (2010). Zulu: an interactive learning competition. In *Lecture notes in computer science: Vol. 6062. Proceedings of the 9th international workshop on finite-state methods and natural language processing* (pp. 139–146). Berlin: Springer.
- Comparetti, P., Wondracek, G., Kruegel, C., & Kirda, E. (2009). Prospec: protocol specification extraction. In *Proceedings of the 30th IEEE symposium on security and privacy* (pp. 110–125). New York: IEEE.
- Cook, J. E., & Wolf, A. L. (1998). Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology*, 7, 215–249.
- Cui, W., Kannan, J., & Wang, H. J. (2007). Discoverer: automatic protocol reverse engineering from network traces. In *Proceedings of 16th USENIX security symposium* (p. 14).
- Dalal, S., Jain, A., Karunanithi, N., Leaton, J., Lott, C., Patton, G., & Horowitz, B. (1999). Model-based testing in practice. In *Proceedings of the 1999 international conference on software engineering, 1999* (pp. 285–294). New York: IEEE.
- Dallmeier, V., Lindig, C., Wasylkowski, A., & Zeller, A. (2006). Mining object behavior with ADABU. In *Proceedings of the 2006 international workshop on dynamic systems analysis (WODA)* (pp. 17–24). New York: ACM.
- D'Argenio, P., Katoen, J.-P., Ruys, T., & Tretmans, J. (1997). The bounded retransmission protocol must be on time! In *Lecture notes in computer science: Vol. 1217. Proceedings of the 3rd workshop on tools and algorithms for the construction and analysis of systems* (pp. 416–431). Berlin: Springer.
- de la Higuera, C. (2010). *Grammatical inference: learning automata and grammars*. New York: Cambridge University Press.
- de la Higuera, C., & Janodet, J.-C. (2004). Inference of omega-languages from prefixes. *Theoretical Computer Science*, 313(2), 295–312.
- Denis, F., Lemay, A., & Terlutte, A. (2000). Learning regular languages using non deterministic finite automata. In *Proceedings of the 6th international colloquium on grammatical inference (ICGI)* (pp. 39–50).
- Dijkstra, E. (1969). Notes on structured programming.

- Frantzen, L., Tretmans, J., & Willemse, T. (2005). Test generation based on symbolic specifications. In *Lecture notes in computer science: Vol. 3395. Proceedings of the 5th international workshop on formal approaches to software testing* (pp. 1–15). Berlin: Springer.
- Garavel, H., Lang, F., Mateescu, R., & Serwe, W. (2011). CADP 2010: a toolbox for the construction and analysis of distributed processes. In *Lecture notes in computer science: Vol. 6605. Proceedings of the 17th international conference on tools and algorithms for the construction and analysis of systems* (pp. 372–387). Berlin: Springer.
- Gold, E. M. (1978). Complexity of automaton identification from given data. *Information and Control*, 37(3), 302–320.
- Grinchev, O., Jonsson, B., & Pettersson, P. (2006). Inference of event-recording automata using timed decision trees. In *Lecture notes in computer science: Vol. 4137. Proceedings of the 17th international conference on concurrency theory (CONCUR)* (pp. 435–449). Berlin: Springer.
- Hansson, H., & Jonsson, B. (1994). A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6, 512–535.
- Helmink, L., Sellink, M., & Vaandrager, F. (1994). Proof-checking a data link protocol. In *Lecture notes in computer science: Vol. 806. Proceedings international workshop TYPES'93* (pp. 127–165). Berlin: Springer.
- Henzinger, T., Nicollin, X., Sifakis, J., & Yovine, S. (1994). Symbolic model checking for real-time systems. *Information and Computation*, 111(2), 193–244.
- Hungar, H., Niese, O., & Steffen, B. (2003). Domain-specific optimization in automata learning. In *Lecture notes in computer science: Vol. 2725. Proceedings of the 15th international conference on computer aided verification (CAV)* (pp. 315–327). Berlin: Springer.
- Ip, C., & Dill, D. (1996). Better verification through symmetry. *Formal Methods in System Design*, 9(1/2), 41–75.
- Katoen, J.-P., Zapreev, I. S., Hahn, E. M., Hermanns, H., & Jansen, D. N. (2011). The ins and outs of the probabilistic model checker MRMC. *Performance Evaluation*, 68(2), 90–104.
- Kearns, M. J., & Vazirani, U. V. (1994). *An introduction to computational learning theory*. Cambridge: MIT Press.
- Lee, D., & Yannakakis, M. (1996). Principles and methods for testing finite state machines—a survey. *Proceedings of the IEEE*, 84(8), 1090–1123.
- Margaria, T., Niese, O., Raffelt, H., & Steffen, B. (2004). Efficient test-based model generation for legacy reactive systems. In *Proceedings of the 9th IEEE international high-level design validation and test workshop (HLDVT)* (pp. 95–100). Washington: IEEE Computer Society.
- Mariani, L., Pastore, F., & Pezze, M. (2011). Dynamic analysis for diagnosing integration faults. *IEEE Transactions on Software Engineering*, 37, 486–508.
- Meinke, K., & Walkinshaw, N. (2012). Model-based testing and model inference. In *Lecture notes in computer science: Vol. 7609. Proceedings of the 5th international symposium on leveraging applications of formal methods, verification and validation (ISoLA)* (pp. 440–443). Berlin: Springer.
- Mostowski, W., Poll, E., Schmaltz, J., Tretmans, J., & Wichers Schreur, R. (2009). Model-based testing of electronic passports. In *Lecture notes in computer science: Vol. 5825. Proceedings of the 15th international workshop on formal methods for industrial critical systems* (pp. 207–209). Berlin: Springer.
- Puterman, M. L. (1994). *Markov decision processes: discrete stochastic dynamic programming* (1st ed.). New York: Wiley.
- Raffelt, H., Steffen, B., Berg, T., & Margaria, T. (2009). Learnlib: a framework for extrapolating behavioral models. *International Journal on Software Tools for Technology Transfer*, 11, 393–407.
- Settles, B. (2010). *Active learning literature survey* (Technical report). University of Wisconsin-Madison.
- Shafique, M., & Labiche, Y. (2010). *A systematic review of model based testing tool support* (Technical Report SCE-10-04). Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada.
- Sudkamp, T. A. (2006). *Languages and machines: an introduction to the theory of computer science* (3rd ed.). Reading: Addison-Wesley.
- Tretmans, J. (2008). Model based testing with labelled transition systems. In *Lecture notes in computer science: Vol. 4949. Formal methods and testing* (pp. 1–38). Berlin: Springer.
- Tretmans, J., & Brinksma, E. (2003). Torx: automated model-based testing. In *Proceedings of the 1st European conference on model-driven software engineering* (pp. 31–43).
- Utting, M., & Legeard, B. (2007). *Practical model-based testing: a tools approach*. San Mateo: Morgan-Kaufmann.
- van der Aalst, W. M. P. (2011). *Process mining—discovery, conformance and enhancement of business processes*. Berlin: Springer.
- Verwer, S. (2010). *Efficient identification of timed automata: theory and practice*. PhD thesis, Delft University of Technology.

- Verwer, S., de Weerd, M., & Witteveen, C. (2011). Efficiently identifying deterministic real-time automata from labeled data. *Machine Learning*, 1–39.
- Walkinshaw, N., Bogdanov, K., Holcombe, M., & Salahuddin, S. (2007). Reverse engineering state machines by interactive grammar inference. In *Proceedings of the 14th working conference on reverse engineering* (pp. 209–218). New York: IEEE.
- Walkinshaw, N., Bogdanov, K., Damas, C., Lambeau, B., & Dupont, P. (2010a). A framework for the competitive evaluation of model inference techniques. In *Proceedings of the 1st international workshop on model inference in testing* (pp. 1–9). New York: ACM.
- Walkinshaw, N., Bogdanov, K., Derrick, J., & Paris, J. (2010b). Increasing functional coverage by inductive testing: a case study. In *Lecture notes in computer science: Vol. 6435. Proceedings of the 22nd IFIP WG 6.1 international conference on testing software and systems (ICTSS)* (pp. 126–141). Berlin: Springer.
- Weyuker, E. J. (1983). Assessing test data adequacy through program inference. *ACM Transactions on Programming Languages and Systems*, 5(4), 641–655.
- Yokomori, T. (1993). Learning non-deterministic finite automata from queries and counterexamples. In *Machine Intelligence* (pp. 196–189). University Press.