Incremental Learning-Based Testing for Reactive Systems

K. Meinke, M. A. Sindhu

School of Computer Science and Communication,
Royal Institute of Technology, 100-44 Stockholm, Sweden,
karlm@nada.kth.se, sindhu@csc.kth.se

Abstract. We show how the paradigm of learning-based testing (LBT) can be applied to automate specification-based black-box testing of reactive systems. Since reactive systems can be modeled as Kripke structures, we introduce an efficient incremental learning algorithm IKL for such structures. We show how an implementation of this algorithm combined with an efficient model checker such as NuSMV yields an effective learning-based testing architecture for automated test case generation (ATCG), execution and evaluation, starting from temporal logic requirements.

1 Introduction

A heuristic approach to automated test case generation (ATCG) from formal requirements specifications known as *learning-based testing* (LBT) was introduced in Meinke [9] and Meinke and Niu [11]. Learning-based testing is an iterative approach to automate specification-based black-box testing. It encompasses both test case generation, execution and evaluation (the oracle step). The aim of this approach is to automatically generate a large number of high-quality test cases by combining a model checking algorithm with an optimised model inference algorithm. For procedural programs, [11] has shown that LBT can significantly outperform random testing in the speed with which it finds errors in a system under test (SUT).

In this paper we consider how the LBT approach can be applied to a quite different class of SUTs, namely *reactive systems*. Conventionally, reactive systems are modeled as *Kripke structures* and their requirements are usually specified using a *temporal logic* (see e.g. [6]). To learn and test such models efficiently, we therefore introduce a new learning algorithm IKL (Incremental Kripke Learning) for Kripke structures. We show that combining the IKL algorithm for model inference together with an efficient temporal logic model checker such as NuSMV yields an effective LBT architecture for reactive systems. We evaluate the effectiveness of this testing architecture by means of case studies.

In the remainder of Section 1 we discuss the general paradigm of LBT, and specific requirements on learning. In Section 2 we review some essential mathematical preliminaries. In Section 3, we consider the technique of bit-sliced learning of Kripke structures. In Section 4, we present a new incremental learning algorithm IKL for Kripke structures that uses distinguishing sequences, bit-slicing,

and lazy partition refinement. In Section 5 we present a complete LBT architecture for reactive systems testing. We evaluate this architecture by means of case studies in Section 6. Finally, in Section 7 we draw some conclusions.

1.1 Learning-Based Testing

Several previous works, (for example Peled et al. [16], Groce et al. [8] and Raffelt et al. [17]) have considered a combination of learning and model checking to achieve testing and/or formal verification of reactive systems. Within the model checking community the verification approach known as *counterexample guided abstraction refinement* (CEGAR) also combines learning and model checking, (see e.g. Clarke et al. [5]). The LBT approach can be distinguished from these other approaches by: (i) an emphasis on testing rather than verification, and (ii) use of *incremental learning algorithms* specifically chosen to make testing more effective and scalable (c.f. Section 1.2).

The basic LBT paradigm requires three components:

(1) a (black-box) *system under test* (SUT) S,

(2) a *formal requirements specification* Req for S, and

(3) a *learned model* M of S.

Now (1) and (2) are common to all specification-based testing, and it is really (3) that is distinctive. Learning-based approaches are *heuristic iterative methods* to automatically generate a sequence of test cases. The heuristic approach is based on *learning a black-box system using tests as queries*.

An LBT algorithm iterates the following four steps:

(Step 1) Suppose that n test case inputs $i_1, \ldots, i_n$ have been executed on S yielding the system outputs $o_1, \ldots, o_n$. The n input/output pairs $(i_1, o_1), \ldots, (i_n, o_n)$ are synthesized into a learned model M_n of S using an *incremental learning algorithm* (see Section 1.2). This step involves *generalization* from the given data, (which represents an incomplete description of S) to all possible data. It gives the possibility to predict previously unseen errors in S during Step 2.

(Step 2) The system requirements Req are satisfiability checked against the learned model M_n derived in Step 1 (aka. *model checking*). This process searches for a *counterexample* i_{n+1} to the requirements.

(Step 3) The counterexample i_{n+1} is executed as the next test case on S, and if S terminates then the output o_{n+1} is obtained. If S fails this test case (i.e. the pair (i_{n+1}, o_{n+1}) does not satisfy Req) then i_{n+1} was a *true negative* and we proceed to Step 4. Otherwise S passes the test case i_{n+1} so the model M_n was inaccurate, and i_{n+1} was a *false negative*. In this latter case, the effort of executing S on i_{n+1} is not wasted. We return to Step 1 and apply the learning algorithm once again to $n+1$ pairs $(i_1, o_1), \ldots, (i_{n+1}, o_{n+1})$ to infer a refined model M_{n+1} of S.

(Step 4) We terminate with a true negative test case (i_{n+1}, o_{n+1}) for S.

Thus an LBT algorithm iterates Steps $1 \ldots 3$ until an SUT error is found (Step 4) or execution is terminated. Possible criteria for termination include a bound on the maximum testing time, or a bound on the maximum number of test cases to be executed.

This iterative approach to TCG yields a sequence of increasingly accurate models M_0, M_1, M_2, ..., of S. (We can take M_0 to be a minimal or even empty model.) So, with increasing values of n, it becomes more and more likely that satisfiability checking in Step 2 will produce a true negative if one exists. If Step 2 does not produce any counterexamples at all then to proceed with the iteration, we must construct the next test case i_{n+1} by some other method, e.g. randomly.

1.2 Efficient Learning Algorithms

As has already been suggested in Step 1 of Section 1.1, for LBT to be effective at finding errors, it is important to use the right kind of learning algorithm. A good learning algorithm should maximise the opportunity of the satisfiability algorithm in Step 2 to find a true counterexample i_{n+1} to the requirements *Req* as soon as possible.

An automata learning algorithm L is said to be *incremental* if it can produce a sequence of hypothesis automata $A_0,\ A_1,\ \ldots$ which are approximations to an unknown automata A, based on a sequence of information (queries and results) about A. The sequence $A_0,\ A_1,\ \ldots$ must finitely converge to A, at least up to behavioural equivalence. In addition, the computation of each new approximation A_{i+1} by L should reuse as much information as possible about the previous approximation A_i (e.g. equivalences between states). Incremental learning algorithms are necessary for efficient learning-based testing of reactive systems for two reasons.

(1) Real reactive systems may be too big to be completely learned and tested within a feasible timescale. This is due to the typical complexity properties of learning and satisfiability algorithms.

(2) Testing of specific requirements such as use cases may not require learning and analysis of the entire reactive system, but only of a fragment that implements the requirement *Req*.

For testing efficiency, we also need to consider the *type of queries* used during learning. The overhead of SUT execution to answer a membership query during learning can be large compared with the execution time of the learning algorithm itself (see e.g. [3]). So membership queries should be seen as "expensive". Therefore, as many queries (i.e. test cases) as possible should be derived from model checking the hypothesis automaton, since these are all based on checking the requirements *Req*. Conversely as few queries as possible should be derived for reasons of internal book-keeping by the learning algorithm (e.g. for achieving congruence closure prior to automaton construction). Book-keeping queries make no reference to the requirements *Req*, and therefore can only uncover an SUT error by accident. Ideally, *every* query would represent a relevant and interesting requirements-based test case. In fact, if the percentage of internally

generated book-keeping queries is too high then model checking becomes almost redundant. In this case we might think that LBT becomes equivalent to random testing. However [18] shows that this is not the case. Even without model checking, LBT achieves better functional coverage than random testing.

In practise, most of the well-known classical regular inference algorithms such as L* (Angluin [2]) or ID (Angluin [1]) are designed for complete rather incremental learning. Among the much smaller number of known incremental learning algorithms, we can mention the RPNII algorithm (Dupont [7]) and the IID algorithm (Parekh et al. [15]) which learn Moore automata, and the CGE algorithm (Meinke [10]) which learns Mealy automata. To our knowledge, no incremental algorithm for learning Kripke structures has yet been published in the literature. Thus the IKL algorithm, and its application to testing represent novel contributions of our paper.

2 Mathematical Preliminaries and Notation

Let Σ be any set of symbols then Σ^* denotes the set of all finite strings over Σ including the empty string ε. The length of a string $\alpha \in \Sigma^*$ is denoted by $|\alpha|$ and $|\varepsilon| = 0$. For strings $\alpha, \beta \in \Sigma^*$, $\alpha \,.\, \beta$ denotes their concatenation.

For $\alpha, \beta, \gamma \in \Sigma^*$, if $\alpha = \beta\gamma$ then β is termed a *prefix* of α and γ is termed a *suffix* of α. We let $Pref(\alpha)$ denote the prefix closure of α, i.e. the set of all prefixes of α. We can also apply prefix closure pointwise to any set of strings. The *set difference operation* between two sets U, V, denoted by $U - V$, is the set of all elements of U which are not members of V. The *symmetric difference* operation on pairs of sets is defined by $U \oplus V = (U - V) \cup (V - U)$.

A *deterministic finite automaton* (DFA) is a five-tuple $A =< \Sigma, Q, F, q_0, \delta >$ where: Σ is the input alphabet, Q is the state set, $F \subseteq Q$ is the accepting state set and $q_0 \in Q$ is the starting state. The state transition function of A is a mapping $\delta : Q \times \Sigma \to Q$ with the usual meaning, and can be inductively extended to a mapping $\delta^* : Q \times \Sigma^* \to Q$ where $\delta^*(q,\ \varepsilon) = q$ and $\delta^*(q,\ \sigma_1,\ \ldots \sigma_{n+1}) = \delta(\delta^*(q,\ \sigma_1,\ \ldots \sigma_n),\ \sigma_{n+1})$. Since input strings can be used to name states, given a distinguished *dead state* d_0 (from which no accepting state can be reached) we define *string concatenation modulo the dead state* d_0, $f : \Sigma^* \cup \{d_0\} \times \Sigma \to \Sigma^* \cup \{d_0\}$, by $f(d_0,\ \sigma) = d_0$ and $f(\alpha,\ \sigma) = \alpha \,.\, \sigma$ for $\alpha \in \Sigma^*$. This function is used for automaton learning in Section 4. The *language* $L(A)$ *accepted by* A is the set of all strings $\alpha \in \Sigma^*$ such that $\delta^*(q_0,\ \alpha) \in F$. A language $L \subseteq \Sigma^*$ is accepted by a DFA if and only if, L is *regular*, i.e. L can be defined by a regular grammar.

A generalisation of DFA to multi-bit outputs on states is given by deterministic Kripke structures.

2.1. Definition. Let $\Sigma = \{\ \sigma_1,\ \ldots,\ \sigma_n\ \}$ be a finite input alphabet. By a k-bit *deterministic Kripke structure* A we mean a five-tuple

$$A = (\ Q_A,\ \Sigma,\ \delta_A : Q_A \times \Sigma \to Q_A,\ q_A^0,\ \lambda_A : Q_A \to \mathbb{B}^k\ )$$

where Q_A is a state set, δ_A is the state transition function, q_A^0 is the initial state and λ_A is the output function. As before we let $\delta_A^* : Q_A \times \Sigma^* \to Q_A$ denote the iterated state transition function, where $\delta_A^*(q,\ \varepsilon) = q$ and $\delta_A^*(q,\ \sigma_1,\ \ldots,\ \sigma_{i+1}) = \delta_A(\delta_A^*(q,\ \sigma_1,\ \ldots,\ \sigma_i),\ \sigma_{i+1})$. Also we let $\lambda_A^* : \Sigma^* \to \mathbb{B}^k$ denote the iterated output function $\lambda_A^*(\sigma_1,\ \ldots,\ \sigma_i) = \lambda_A(\delta_A^*(q_A^0,\ \sigma_1,\ \ldots,\ \sigma_i))$.

If A is a Kripke structure then the *minimal subalgebra* $Min(A)$ of A is the unique subalgebra of A which has no proper subalgebra. (We implicitly assume that all input symbols $\sigma \in \Sigma$ are constants of A so that $Min(A)$ has a non-trivial state set.) Note that a 1-bit deterministic Kripke structure A is isomorphic to the DFA $A' = (\ Q_A,\ \Sigma,\ \delta_A : Q_A \times \Sigma \to Q_A,\ q_A^0,\ F_{A'}\ )$, where $F_{A'} \subseteq Q_A$ and $\lambda_A(q) = true$ if, and only if $q \in F_{A'}$.

3 Bit-Sliced Learning of Kripke Structures

We will establish a precise basis for learning k-bit Kripke structures using regular inference algorithms for DFA. The approach we take is to bit-slice the output of a k-bit Kripke structure A into k individual 1-bit Kripke structures $A_1,\ \ldots,\ A_k$, which are learned in parallel as DFA by some regular inference algorithm. The k inferred DFA $B_1,\ \ldots,\ B_k$ are then recombined using a subdirect product construction to obtain a Kripke structure that is behaviourally equivalent to A.

This approach has three advantages: (1) We can make use of *any* regular inference algorithm to learn the individual 1-bit Kripke structures A_i. Thus we have access to the wide range of known regular inference algorithms. (2) We can reduce the total number of book-keeping queries by *lazy book-keeping*. This technique maximises re-use of book-keeping queries among the 1-bit structures A_i. In Section 4, we illustrate this technique in more detail. (3) We can learn just those bits which are necessary to test a specific temporal logic requirement. This *abstraction technique* improves the scalability of testing.

It usually suffices to learn automata up to behavioural equivalence.

3.1. Definition. Let A and B be k-bit Kripke structures over a finite input alphabet Σ. We say that A and B are *behaviourally equivalent*, and write $A \equiv B$ if, and only if, for every finite input sequence $\sigma_1,\ \ldots,\ \sigma_i \in \Sigma^*$ we have

$$\lambda_A^*(\ \sigma_1,\ \ldots,\ \sigma_i\ ) = \lambda_B^*(\ \sigma_1,\ \ldots,\ \sigma_i\ ).$$

Clearly, by the isomorphism identified in Section 2 between 1-bit Kripke structures and DFA, for such structures we have $A \equiv B$ if, and only if, $L(A') = L(B')$. Furthermore, if $Min(A)$ is the minimal subalgebra of A then $Min(A) \equiv A$.

Let us make precise the concept of bit-slicing a Kripke structure.

3.2. Definition. Let A be a k-bit Kripke structure over a finite input alphabet Σ,

$$A = (\ Q_A,\ \Sigma,\ \delta_A : Q_A \times \Sigma \to Q_A,\ q_A^0,\ \lambda_A : Q_A \to \mathbb{B}^k\ ).$$

For each $1 \leq i \leq k$ define the *i-th projection* A_i of A to be the 1-bit Kripke structure where

$$A_i = (\ Q_A,\ \Sigma,\ \delta_A : Q_A \times \Sigma \to Q_A,\ q_A^0,\ \lambda_{A_i} : Q_A \to \mathbb{B}\ ),$$

and $\lambda_{A_i}(q) = \lambda_A(q)_i$, i.e. $\lambda_{A_i}(q)$ is the i-th bit of $\lambda_A(q)$.

A family of k individual 1-bit Kripke structures can be combined into a single k-bit Kripke structure using the following subdirect product construction. (See e.g. [13] for a general definition of subdirect products and their universal properties.)

3.3. Definition. Let $A_1,\ \ldots,\ A_k$ be a family of 1-bit Kripke structures,

$$A_i = (\ Q_i,\ \Sigma,\ \delta_i : Q_i \times \Sigma \to Q_i,\ q_i^0,\ \lambda_i : Q \to \mathbb{B}\ )$$

for $i = 1,\ \ldots,\ k$. Define the *product Kripke structure*

$$\prod_{i=1}^{k} A_i = (\ Q,\ \Sigma,\ \delta : Q \times \Sigma \to Q,\ q^0,\ \lambda : Q \to \mathbb{B}^k\ ),$$

where $Q = \prod_{i=1}^{k} Q_i = Q_1 \times \ldots \times Q_k$ and $q^0 = (\ q_1^0,\ \ldots,\ q_k^0\ )$. Also

$$\delta(q_1,\ \ldots,\ q_k,\ \sigma) = (\ \delta_1(q_1,\ \sigma),\ \ldots,\ \delta_k(q_k,\ \sigma)\ ),$$

$$\lambda(q_1,\ \ldots,\ q_k) = (\ \lambda_1(q_1),\ \ldots,\ \lambda_k(q_k)\ ).$$

Associated with the direct product $\prod_{i=1}^{k} A_i$ we have i-th *projection mapping*

$$proj_i : Q_1 \times \ldots \times Q_k \to Q_i, \quad proj_i(q_1,\ \ldots,\ q_k) = q_i,\ 1 \leq i \leq k$$

Let $Min(\ \prod_{i=1}^{k} A_i\ )$ be the minimal subalgebra of $\prod_{i=1}^{k} A_i$.

The reason for taking the minimal subalgebra of the direct product $\prod_{i=1}^{k} A_i$ is to avoid the state space explosion due to a large number of unreachable states in the direct product itself. The state space size of $\prod_{i=1}^{k} A_i$ grows exponentially with k. On the other hand since most of these states are unreachable from the initial state, then from the point of view of behavioural analysis these states are irrelevant. Note that this minimal subalgebra can be computed in linear time from its components A_i (w.r.t. state space size).

As is well known from universal algebra, the i-th projection mapping $proj_i$ is a homomorphism.

3.4. Proposition. *Let $A_1,\ \ldots,\ A_k$ be any minimal 1-bit Kripke structures.*
(i) *For each $1 \leq i \leq k$, $proj_i : Min(\ \prod_{i=1}^{k} A_i\ ) \to A_i$ is an epimorphism, and hence $Min(\ \prod_{i=1}^{k} A_i\ )$ is a subdirect product of the A_i.*

(ii) $Min(\ \prod_{i=1}^{k} A_i\ ) \equiv \prod_{i=1}^{k} A_i$.

Proof. (i) Immediate since the A_i are minimal. (ii) Follows from $Min(A) \equiv A$.

The following theorem justifies bit-sliced learning of k-bit Kripke structures using conventional regular inference methods for DFA.

3.5. Theorem. *Let A be a k-bit Kripke structure over a finite input alphabet Σ. Let $A_1, \ldots, A_k$ be the k individual 1-bit projections of A. For any 1-bit Kripke structures $B_1, \ldots, B_k$, if, $A_1 \equiv B_1$ & ... & $A_k \equiv B_k$ then*

$$A \equiv Min(\prod_{i=1}^{k} B_i).$$

Proof. Use Proposition 3.4.

4 Incremental Learning for Kripke Structures

In this section we present a new algorithm for incremental learning of Kripke structures. We will briefly discuss its correctness and termination properties, although a full discussion of these is outside the scope of this paper and is presented elsewhere in [12]. Our algorithm applies bit-slicing as presented in Section 3, and uses distinguishing sequences and lazy partition refinement for regular inference of the 1-bit Kripke structures. The architecture of the IKL algorithm consists of a main learning algorithm and two sub-procedures for lazy partition refinement and automata synthesis. Distinguishing sequences were introduced in Angluin [1] as a method for learning DFA.

Algorithm 1 is the main algorithm for bit-sliced incremental learning. It learns a sequence $M_1, \ldots, M_l$ of n-bit Kripke structures that successively approximate a single n-bit Kripke structure A, which is given as the teacher. In LBT, the teacher is always the SUT.

The basic idea of Algorithm 1 is to construct in parallel a family $E^1_{i_1}, \ldots, E^n_{i_n}$ of n different equivalence relations on the same set T_k of state names. For each equivalence relation $E^j_{i_j}$, a set V_j of distinguishing strings is generated iteratively to split pairs of equivalence classes in $E^j_{i_j}$ until a congruence is achieved. Then a quotient DFA M^j can be constructed from the partition of T_k by $E^j_{i_j}$. The congruences are constructed so that $E^j_i \subseteq E^j_{i+1}$ and thus the IKL algorithm is incremental, and fully reuses information about previous approximations, which is efficient.

Each n-bit Kripke structure M_t is constructed using synthesis algorithm 3, as a subdirect product of n individual quotient DFA $M^1, \ldots, M^n$ (viewed as 1-bit Kripke structures). When the IKL algorithm is applied to the problem of LBT, the input strings $s_i \in \Sigma^*$ to IKL are generated as counterexamples to correctness (i.e. test cases) by executing a model checker on the approximation M_{t-1} with respect to some requirements specification ϕ expressed in temporal logic. If no counterexamples to ϕ can be found in M_{t-1} then s_i is randomly chosen, taking care to avoid all previously used input strings.

Algorithm 2 implements lazy partition refinement, to extend $E^1_{i_1}, \ldots, E^n_{i_n}$ from being equivalence relations on states to being a family of congruences with respect to the state transition functions $\delta_1, \ldots, \delta_n$ of $M^1, \ldots, M^n$.

Algorithm 1 IKL: Incremental Kripke Structure Learning Algorithm

Input: A file $S = s_1, \ldots, s_l$ of input strings $s_i \in \Sigma^*$ and a Kripke structure A with n-bit output as teacher to answer queries $\lambda^*_A(s_i) = ?$
Output: A sequence of Kripke structures M_t with n-bit output for $t = 0, \ldots, l$.

1. **begin**
2. //Perform Initialization
3. **for** c = 1 to n **do** { $i_c = 0$, $v_{i_c} = \varepsilon$, $V_c = \{v_{i_c}\}$ }
4. $k = 0$, $t = 0$,
5. $P_0 = \{\varepsilon\}$, $P'_0 = P_0 \cup \{d_0\}$, $T_0 = P_0 \cup \Sigma$
6. //Build equivalence classes for the dead state d_0
7. **for** c = 1 to n **do** { $E^c_0(d_0) = \emptyset$ }
8. //Build equivalence classes for input strings of length zero and one
9. $\forall \alpha \in T_0$ {
10. $(b_1, \ldots, b_n) = \lambda^*_A(\alpha)$
11. **for** c = 1 to n **do**
12. **if** b_c **then** $E^c_{i_c}(\alpha) = \{v_{i_c}\}$ **else** $E^c_{i_c}(\alpha) = \emptyset$
13. }
14. //Refine the initial equivalence relations $E^1_0, \ldots, E^n_0$
15. //into congruences using Algorithm 2
16.
17. //Synthesize an initial Kripke structure M_0 approximating A
18. //using Algorithm 3.
19.
20. //Process the file of examples.
21. **while** $S \neq \mathit{empty}$ **do** {
22. read(S, α)
23. k = k+1, t = t+1
24. $P_k = P_{k-1} \cup Pref(\alpha)$ //prefix closure
25. $P'_k = P_k \cup \{d_0\}$
26. $T_k = T_{k-1} \cup Pref(\alpha) \cup \{\alpha \,.\, b \mid \alpha \in P_k - P_{k-1}, b \in \Sigma\}$ //for prefix closure
27. $T'_k = T_k \cup \{d_0\}$
28. $\forall \alpha \in T_k - T_{k-1}$ {
29. **for** c = 1 to n **do** $E^c_0(\alpha) = \emptyset$ //initialise new equivalence class $E^c_0(\alpha)$
30. **for** j = 0 to i_c **do** {
31. // Consider adding distinguishing string $v_j \in V_c$
32. // to each new equivalence class $E^c_j(\alpha)$
33. $(b_1, \ldots, b_n) = \lambda^*_A(\alpha \,.\, v_j)$
34. **if** b_c **then** $E^c_j(\alpha) = E^c_j(\alpha) \cup \{ v_j \}$
35. }
36. }
37. //Refine the current equivalence relations $E^1_{i_1}, \ldots, E^n_{i_n}$
38. // into congruences using Algorithm 2
39.
40. **if** α is consistent with M_{t-1}
41. **then** $M_t = M_{t-1}$
42. **else** synthesize Kripke structure M_t using Algorithm 3.
43. }
44. **end.**

Algorithm 2 Lazy Partition Refinement

```
1. while (∃ 1 ≤ c ≤ n, ∃α, β ∈ P'_k and ∃σ ∈ Σ such that E^c_{i_c}(α) = E^c_{i_c}(β) but
   E^c_{i_c}(f(α, σ)) ≠ E^c_{i_c}(f(β, σ)) do {
2.   //Equivalence relation E^c_{i_c} is not a congruence w.r.t. δ_c
3.   //so add a new distinguishing sequence.
4.   Choose γ ∈ E^c_{i_c}(f(α, σ)) ⊕ E^c_{i_c}(f(β, σ))
5.   v = σ . γ
6.   ∀α ∈ T_k {
7.     (b_1, ..., b_n) = λ*_A(α . v)
8.     for c = 1 to n do {
9.       if E^c_{i_c}(α) = E^c_{i_c}(β) and E^c_{i_c}(f(α, σ)) ≠ E^c_{i_c}(f(β, σ)) then {
10.        // Lazy refinement of equivalence relation E^c_{i_c}
11.        i_c = i_c + 1, v_{i_c} = v, V_c = V_c ∪ {v_{i_c}}
12.        if b_c then E^c_{i_c}(α) = E^c_{i_c−1}(α) ∪ {v_{i_c}} else E^c_{i_c}(α) = E^c_{i_c−1}(α)
13.      }
14.    }
15. }
```

Algorithm 3 Kripke Structure Synthesis

```
1. for c = 1 to n do {
2.     // Synthesize the quotient DFA (1-bit Kripke structure) M^c
3.      The states of M^c are the sets E^c_{i_c}(α), where α ∈ T_k
4.      Let q^c_0 = E^c_{i_c}(ε)
5.      The accepting states are the sets E^c_{i_c}(α) where α ∈ T_k and ε ∈ E^c_{i_c}(α)
6.      The transition function δ_c of M^c is defined as follows:
7.        ∀α ∈ P'_k {
8.            if E^c_{i_c}(α) = ∅ then ∀b ∈ Σ { let δ_c(E^c_{i_c}(α), b) = E^c_{i_c}(α) }
9.            else ∀b ∈ Σ { δ_c(E^c_{i_c}(α), b) = E^c_{i_c}(α . b) }
10.       }
11.       ∀β ∈ T_k − P'_k {
12.           if ∀α ∈ P'_k { E^c_{i_c}(β) ≠ E^c_{i_c}(α) } and E^c_{i_c}(β) ≠ ∅ then
13.              ∀b ∈ Σ { δ_c(E^c_{i_c}(β), b) = ∅ }
14.       }
15. // Compute M_t in linear time as a subdirect product of the M^c
16. M_t = Min( ∏_{c=1}^{n} M^c )
```

Thus line 1 searches for congruence failure in any one of the equivalence relations $E^1_{i_1}, \ldots, E^n_{i_n}$. In lines 6-14 we apply lazy partition refinement. This technique implies reusing the new distinguishing string v wherever possible to refine each equivalence relation $E^j_{i_j}$ that is not yet a congruence. On the other hand, any equivalence relation $E^j_{i_j}$ that is already a congruence is not refined, even though the result b_j of the new query $\alpha \,.\, v$ might add some new information to M^j. This helps minimise the total number of partition refinement queries (cf. Section 1.2).

Algorithm 3 implements model synthesis. First, each of the n quotient DFA $M^1, \ldots, M^n$ are constructed. These, reinterpreted as 1-bit Kripke structures, are then combined in linear time as a subdirect product to yield a new n-bit approximation M_t to A (c.f. Section 3).

4.1 Correctness and Termination of the IKL algorithm.

The sequence $M_1, \ldots, M_l$ of hypothesis Kripke structures which are incrementally generated by IKL can be proven to finitely converge to A up to behavioural equivalence, for sufficiently large l. The key to this observation lies in the fact that we can identify a finite set of input strings such that the behavior of A is completely determined by its behaviour on this finite set.

Recall that for a DFA $A =< \Sigma, Q, F, q_0, \delta >$ a state $q \in Q$ is said to be *live* if for some string $\alpha \in \Sigma^*$, $\delta^*(q, \alpha) \in F$. A finite set $C \subseteq \Sigma^*$ of input strings is said to be *live complete* for A if for every reachable live state $q \in Q$ there exists a string $\alpha \in C$ such that $\delta^*(q_0, \alpha) = q$. More generally, given a finite collection $A_1, \ldots, A_k$ of DFA, then $C \subseteq \Sigma^*$ is *live complete* for $A_1, \ldots, A_k$ if, and only if, for each $1 \le i \le k$, C is a live complete set for A_i. Clearly, for every finite collection of DFA there exists at least one live complete set of strings.

4.1.1. Theorem. *Let A be a k-bit Kripke structure over a finite input alphabet Σ. Let $A_1, \ldots, A_k$ be the k individual 1-bit projections of A. Let $C = \{ s_1, \ldots, s_l \} \subseteq \Sigma^*$ be a live complete set for $A_1, \ldots, A_k$. The IKL algorithm terminates on C and for the final hypothesis structure M_l we have*

$$M_l \equiv A.$$

Proof. See [12].

5 A Learning-Based Testing Architecture using IKL.

Figure 1 depicts an architecture for learning-based testing of reactive systems by combining the IKL algorithm of Section 4 with a model checker for Kripke stuctures and an oracle. In this case we have chosen to use the NuSMV model checker (see e.g. Cimatti et al. [4]), which supports the satisfiability analysis of Kripke structures with respect to both linear temporal logic (LTL) and computation tree logic (CTL) [6].

To understand this architecture, it is useful to recall the abstract description of learning-based testing as an iterative process, given in Section 1.1. Following the account of Section 1.1, we can assume that at any stage in the testing process we have an inferred Kripke structure M_n produced by the IKL algorithm from previous testing and learning. Test cases will have been produced as counterexamples to correctness by the model checker, and learning queries will have been produced by the IKL algorithm during partition refinement. (Partition refinement queries are an example of what we termed book-keeping queries in Section 1.2.)

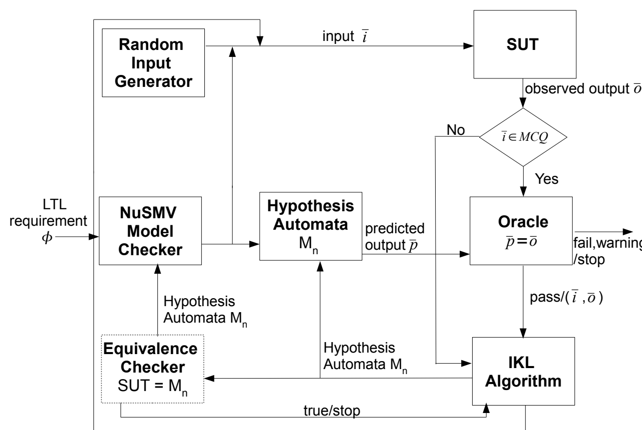

Fig. 1. A Learning-Based Testing Architecture using the IKL algorithm.

In Figure 1, the output M_n of the IKL algorithm is passed to an *equivalence checker*. Since this architectural component is not normally part of an LBT framework, we should explain its presence carefully. We are particularly interested in benchmarking the performance of LBT systems, both to compare their performance with other testing methodologies, and to make improvements to existing LBT systems. (See Section 6.) In realistic testing situations, we do not anticipate that an entire SUT can be learned in a feasible time (c.f. the discussion in Section 1.2). However, for benchmarking with the help of smaller case studies (for which complete learning is feasible) it is useful to be able to infer the earliest time at which we can say that testing is complete. Obviously testing must be complete at time t_{total} when we have learned the entire SUT (c.f. Section 6). Therefore the equivalence checker allows us to compute the time t_{total} simply to conduct benchmarking studies. (Afterwards the equivalence checker is removed.) The equivalence checker compares the current Kripke structure M_n with the SUT. A positive result from this equivalence test stops all further learning and testing after one final model check. The algorithm we use has

been adapted from the quasi-linear time algorithm for DFA equivalence checking described in [14] and has been extended to deal with k-bit Kripke structures.

In Figure 1, the inferred model M_n is passed to a model checker, together with a user requirement represented as a temporal logic formula ϕ. This formula is constant during a particular testing experiment. The model checker attempts to identify at least one counterexample to ϕ in M_n as an *input sequence* $\bar{i}$. If ϕ is a safety formula then this input sequence will usually be finite $\bar{i} = i_1, \ldots, i_k$. If ϕ is a liveness formula then this input sequence $\bar{i}$ may be finite or infinite. Recall that infinite counterexamples to liveness formulas can be represented as infinite sequences of the form $\bar{x}\,\bar{y}^\omega$. In the case that $\bar{i} = \bar{x}\,\bar{y}^\omega$ then $\bar{i}$ is truncated to a finite initial segment that would normally include the handle $\bar{x}$ and at least one execution of the infinite loop $\bar{y}^\omega$, such as $\bar{i} = \bar{x}\,\bar{y}$ or $\bar{i} = \bar{x}\,\bar{y}\,\bar{y}$. Observing the failure of an infinite test case is of course impossible. The LBT architecture implements a compromise solution that runs the truncated sequence only, in finite time, and issues a warning rather than a fail verdict.

Note that if the next input sequence $\bar{i}$ cannot be constructed either by partition refinement or by model checking then in order to proceed with iterative testing and learning, another way to generate $\bar{i}$ must be found. (See the discussion in Section 1.1.) One simple solution, shown in Figure 1, is to use a *random input sequence generator* for $\bar{i}$, taking care to discard any previously used sequences.

Thus from one of three possible sources (partition refinement, model checking or randomly) a new input sequence $\bar{i} = i_1, \ldots, i_k$ is constructed. Figure 1 shows that if $\bar{i}$ is obtained by model checking then the current model M_n is applied to $\bar{i}$ to compute a *predicted output* $\bar{p} = p_1, \ldots, p_k$ for the SUT that can be used for the oracle step. However, this is not possible if $\bar{i}$ is random or a partition refinement since then we do not know whether $\bar{i}$ is a counterexample to ϕ. Nevertheless, in all three cases, the input sequence $\bar{i}$ is passed to the SUT and executed to yield an *actual* or *observed output sequence* $\bar{o} = o_1, \ldots, o_k$.

The final stage of this iterative testing architecture is the *oracle step*. Figure 1 shows that if a predicted output $\bar{p}$ exists (i.e. the input sequence $\bar{i}$ came from model checking) then actual output $\bar{o}$ and the predicted output $\bar{p}$ are both passed to an *oracle* component. This component implements the Boolean test $\bar{o} = \bar{p}$. If this equality test returns *true* and the test case $\bar{i} = i_1, \ldots, i_k$ was originally a finite test case then we can conclude that the test case $\bar{i}$ is definitely *failed*, since the behaviour $\bar{p}$ is by construction a counterexample to the correctness of ϕ. If the equality test returns *true* and the test case $\bar{i}$ is finitely truncated from an infinite test case (a counterexample to a liveness requirement) then the verdict is weakened to a *warning*. This is because the most we can conclude is that we have not yet seen any difference between the observed behaviour $\bar{o}$ and the incorrect behaviour $\bar{p}$. The system tester is thus encouraged to consider a potential SUT error.

On the other hand if $\bar{o} \neq \bar{p}$, or if no output prediction $\bar{p}$ exists then it is quite difficult to issue an immediate verdict. It may or may not be the case that the observed output $\bar{o}$ is a counterexample to the correctness of ϕ. In some cases the syntactic structure of ϕ is simple enough to semantically evaluate the formula

ϕ on the fly with its input and output variables bound to $\bar{i}$ and $\bar{o}$ respectively. However, sometimes this is not possible since the semantic evaluation of ϕ also refers to global properties of the automaton. Ultimately, this is not a problem for our approach, since M_{n+1} is automatically updated with the output behaviour $\bar{o}$. Model checking M_{n+1} later on will confirm $\bar{o}$ as an error if this is the case.

5.1 Correctness and Termination of the LBT Architecture.

It is important to establish that the LBT architecture always terminates, at least in principle. Furthermore, the SUT coverage obtained by this testing procedure is complete, in the sense that if the SUT contains any counterexamples to correctness then a counterexample *will* be found by the testing architecture. When the SUT is too large to be completely learned in a feasible amount of time, this completeness property of the testing architecture still guarantees that there is no *bias* in testing so that one could somehow never discover an SUT error. Failure to find an error in this case is purely a consequence of insufficient testing time.

The termination and correctness properties of the LBT architecture depend on the following correctness properties of its components:

(i) the IKL algorithm terminates and correctly learns the SUT given a finite set C of input strings which is live complete (c.f. Theorem 4.1.1);

(ii) the model checking algorithm used by NuSMV is a terminating decision procedure for the validity of LTL formulas;

(iii) each input string $i \in \Sigma^*$ is generated with non-zero probability by the random input string generator.

5.1.1. Theorem. *Let A be a k-bit Kripke structure over an input alphabet Σ.*

(i) The LBT architecture (with equivalence checker) terminates with probability 1.0*, and for the final hypothesis structure M_l we have*

$$M_l \equiv A.$$

(ii) If there exists a (finite or infinite) input string $\bar{i}$ over Σ which witnesses that an LTL requirement ϕ is not valid for A, then model checking will eventually find such a string $\bar{i}$ and the LBT architecture will generate a test fail or test warning message after executing $\bar{i}$ as a test case on A.

Proof. (i) Clearly by Theorem 4.1.1, the IKL algorithm will learn the SUT A up to behavioural equivalence, given as input a live complete set C for the individual 1-bit projections $A_1, \ldots, A_k$ of A. Now, we cannot be sure that the strings generated by model checking counterexamples and partition refinement queries alone constitute a live complete set C for $A_1, \ldots, A_k$. However, these sets of queries are complemented by random queries. Since a live complete set is finite, and every input string is randomly generated with non-zero probability, then with probability 1.0 the IKL algorithm will eventually obtain a live complete set and converge. At this point, equivalence checking the final hypothesis structure M_l with the SUT will succeed and the LBT architecture will terminate.

(ii) Suppose there is at least one (finite or infinite) counterexample string $\bar{i}$ over Σ to the validity of an LTL requirement ϕ for A. In the worst case, by part (i), the LBT architecture will learn the entire structure of A. Since the model checker implements a terminating decision procedure for validity of LTL formulas, it will return a counterexample $\bar{i}$ from the final hypothesis structure M_l, since by part (i), $M_l \equiv A$ and A has a counterexample. For such $\bar{i}$, comparing the corresponding predicted output $\bar{p}$ from M_l and the observed output $\bar{o}$ from A we must have $\bar{p} = \bar{o}$ since $M_l \equiv A$. Hence the testing architecture will issue a fail or warning message.

6 Case Studies and Performance Benchmarking

In order to evaluate the effectiveness of the LBT architecture described in Section 5, we conducted a number of testing experiments on two SUT case studies, namely an 8 state cruise controller and a 38 state 3-floor elevator model[1].

For each SUT case study we chose a collection of safety and liveness requirements that could be expressed in linear temporal logic (LTL). For each requirement we then injected an error into the SUT that violated this requirement and ran a testing experiment to discover the injected error. The injected errors all consisted of *transition mutations* obtained by redirecting a transition to a wrong state. This type of error seems quite common in our experience.

There are a variety of ways to measure the performance of a testing system such as this. One simple measure that we chose to consider was to record the *first time* t_{first} at which an error was discovered in an SUT, and to compare this with the *total time* t_{total} required to completely learn the SUT. (So $t_{first} \leq t_{total}$.) This measure is relevant if we wish to estimate the benefit of using incremental learning instead of complete learning.

Because some random queries are almost always present in each testing experiment, the performance of the LBT architecture has a degree of variation. Therefore, for the same correctness formula and injected error, we ran each experiment ten times to try to average out these variations in performance. This choice appeared adequate to obtain a representative average. Subsections 6.1 and 6.2 below set out the results obtained for each case study.

6.1 The Cruise Controller Model

The cruise controller model we chose as an SUT is an 8 state 5-bit Kripke structure with an input alphabet of 5 symbols. Figure 2 shows its structure[2]. The four requirements shown in Table 1 consist of: (1,2) two requirements on speed maintenance against obstacles in cruise mode, and (3,4) disengaging cruise mode by means of the brake and gas pedals. To gain insight into the LBT

[1] Our testing platform was based on a PC with a 1.83 GHz Intel Core 2 duo processor and 4GB of RAM running Windows Vista.

[2] The following binary data type encoding is used. Modes: 00 = manual, 01 = cruise, 10 = disengaged. Speeds: 00 = 0, 01 = 1, 10 = 2.

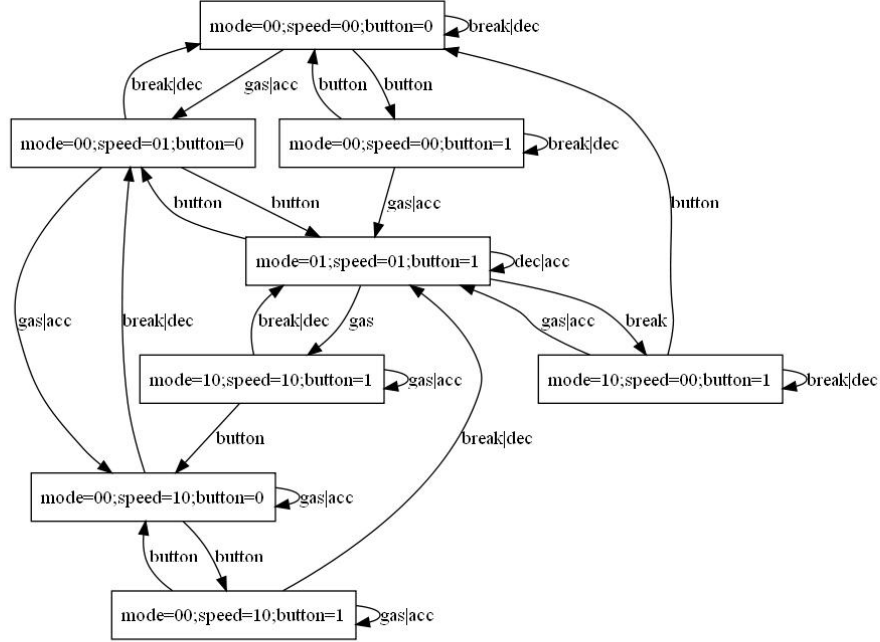

Fig. 2. The cruise controller SUT.

Req 1	G(mode = cruise & speed = 1 & in = dec -> X(speed = 1))
Req 2	G(mode = cruise & speed = 1 & in = acc -> X(speed = 1))
Req 3	G(mode = cruise & in = brake -> X(mode = disengaged))
Req 4	G(mode = cruise & in = gas -> X(mode = disengaged))

Table 1. Cruise Controller Requirements as LTL formulas.

Requirement	t_{first} (sec)	t_{total} (sec)	MCQ_{first}	MCQ_{total}	PQ_{first}	PQ_{total}	RQ_{first}	RQ_{total}
Req 1	3.5	21.5	3.2	24.3	7383	30204	8.2	29.3
Req 2	2.3	5.7	5.5	18.2	8430	27384	10.4	23.1
Req 3	2.3	16.0	1.7	33.7	6127	34207	6.8	38.8
Req 4	2.9	6.1	4.7	20.9	7530	24566	10.4	20.9

Table 2. LBT performance for Cruise Controller Requirements.

architecture performance, Table 2 shows average figures at times t_{first} and t_{total} for the numbers:

(i) MCQ_{first} and MCQ_{total} of model checker generated test cases,
(ii) PQ_{first} and PQ_{total} of partition refinement queries,
(iii) RQ_{first} and RQ_{total} of random queries.

In Table 2, columns 2 and 3 show that the times required to first discover an error in the SUT are between 14% and 47% of the total time needed to completely learn the SUT. The large query numbers in columns 6 and 7 show that partition refinement queries dominate the total number of queries. Columns 8 and 9 show that the number of random queries used is very low, of and of the same order of magnitude as the number of model checking queries (columns 4 and 5). Thus partition refinement queries and model checker generated test cases come quite close to achieving a live complete set, although they do not completely suffice for this (c.f. Section 4.1).

6.2 The Elevator Model

The elevator model we chose as an SUT is a 38 state 8-bit Kripke structure with an input alphabet of 4 symbols. Figure 3 shows its condensed structure as a hierarchical statechart. The six requirements shown in Table 3 consist of

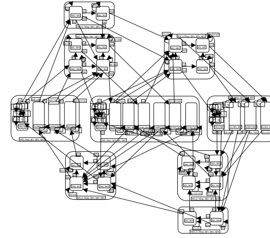

Fig. 3. The 3-floor elevator SUT (condensed Statechart).

Req 1	G(Stop -> (@1 | @2 | @3))
Req 2	G(!Stop -> cl)
Req 3	G(Stop & X(!Stop) -> X(!cl))
Req 4	G(Stop & @1 & cl & in=c1 & X(@1) -> X(!cl))
Req 5	G(Stop & @2 & cl & in=c2 & X(@2) -> X(!cl))
Req 6	G(Stop & @3 & cl & in=c3 & X(@3) -> X(!cl))

Table 3. Elevator Requirements as LTL formulas.

Requirement	t_{first} (sec)	t_{total} (sec)	MCQ_{first}	MCQ_{total}	PQ_{first}	PQ_{total}	RQ_{first}	RQ_{total}
Req 1	0.34	1301.3	1.9	81.7	1574	729570	1.9	89.5
Req 2	0.49	1146	3.9	99.6	2350	238311	2.9	98.6
Req 3	0.94	525	1.6	21.7	6475	172861	5.7	70.4
Req 4	0.052	1458	1.0	90.3	15	450233	0.0	91
Req 5	77.48	2275	1.2	78.3	79769	368721	20.5	100.3
Req 6	90.6	1301	2.0	60.9	129384	422462	26.1	85.4

Table 4. LBT performance for Elevator Requirements.

requirements that: (1) the elevator does not stop between floors, (2) doors are closed when in motion, (3) doors open upon reaching a floor, and (4, 5, 6) closed doors can be opened by pressing the same floor button when stationary at a floor.

Table 4 shows the results of testing the requirements of Table 3. These results confirm several trends seen in Table 2. However, they also show a significant increase in the efficiency of using incremental learning, since the times required to first discover an error in the SUT are now between 0.003% and 7% of the total time needed to completely learn the SUT. These results are consistent with observations of [12] that the convergence time of IKL grows quadratically with state space size. Therefore incremental learning gives a more scalable testing method than complete learning.

7 Conclusions

We have presented a novel incremental learning algorithm for Kripke structures, and shown how this can be applied to learning-based testing of reactive systems. Using two case studies of reactive systems, we have confirmed our initial hypothesis of Section 1.2, that incremental learning is a more scalable and efficient method of testing than complete learning. These results are consistent with similar results for LBT applied to procedural systems in [11].

Further research could be carried out to improve the performance of the architecture presented here. For example the performance of the oracle described in Section 5 could be improved to yield a verdict even for random and partition queries, at least for certain kinds of LTL formulas. Further research into scalable learning algorithms would be valuable for dealing with large hypothesis automata. The question of learning-based coverage has been initially explored in [18] but further research here is also needed.

We gratefully acknowledge financial support for this research from the Swedish Research Council (VR), the Higher Education Commission (HEC) of Pakistan, and the European Union under project HATS FP7-231620.

References

1. D. Angluin. A note on the number of queries needed to identify regular languages. *Information and Control*, 51(1):76–87, October 1981.
2. D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(1):87–106, November 1987.
3. T. Bohlin and B. Jonsson. Regular inference for communication protocol entities. Technical Report 2008-024, Dept. of Information Technology, Uppsala University, 2008.
4. A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: a new symbolic model verifier. In *Proceeding of International Conference on Computer-Aided Verification (CAV'99)*, number 1633 in LNCS. Springer, 1999.
5. E. Clarke, A. Gupta, J. Kukula, and O. Strichman. Sat-based abstraction refinement using ilp and machine learning. In *Proc. 21st International Conference On Computer Aided Verification (CAV'02)*, 2002.
6. E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 1999.
7. P. Dupont. Incremental regular inference. In *Proceedings of the Third ICGI-96*, number 1147 in LNAI, 1996.
8. A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. *Logic Journal of the IGPL*, 14(5):729–744, 2006.
9. K. Meinke. Automated black-box testing of functional correctness using function approximation. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 143–153, New York, NY, USA, 2004. ACM.
10. K. Meinke. Cge: A sequential learning algorithm for mealy automata. In *Proc. Tenth Int. Colloq. on Grammatical Inference (ICGI 2010)*, number 6339 in LNAI, pages 148–162. Springer, 2010.
11. K. Meinke and F. Niu. A learning-based approach to unit testing of numerical software. In *Proc. Twenty Second IFIP Int. Conf. on Testing Software and Systems (ICTSS 2010)*, number 6435 in LNCS, pages 221–235. Springer, 2010.
12. K. Meinke and M. Sindhu. Correctness and performance of an incremental learning algorithm for kripke structures. Technical report, School of Computer Science and Communication, Royal Institute of Technology, Stockholm, 2010.
13. K. Meinke and J.V. Tucker. Universal algebra. In *Handbook of Logic in Computer Science: Volume 1*, pages 189–411. Oxford University Press, 1993.
14. D.A. Norton. Algorithms for testing equivalence of finite state automata, with a grading tool for jflap. Technical report, Rochester Institute of Technology, Department of Computer Science, 2009.
15. R.G. Parekh, C. Nichitiu, and V.G. Honavar. A polynomial time incremental algorithm for regular grammar inference. In *Proc. Fourth Int. Colloq. on Grammatical Inference (ICGI 98)*, LNAI. Springer, 1998.
16. D. Peled, M.Y. Vardi, and M. Yannakakis. Black-box checking. In *Formal Methods for Protocol Engineering and Distributed Systems FORTE/PSTV*, pages 225–240. Kluwer, 1999.
17. H. Raffelt, B. Steffen, and T. Margaria. Dynamic testing via automata learning. In *Hardware and Software: Verification and Testing*, number 4899 in LNCS, pages 136–152. Springer, 2008.
18. N. Walkinshaw, K. Bogdanov, J. Derrick, and J. Paris. Increasing functional coverage by inductive testing: a case study. In *Proc. Twenty Second IFIP Int. Conf. on Testing Software and Systems (ICTSS 2010)*, number 6435 in LNCS, pages 126–141. Springer, 2010.