Learning-Based Testing for Reactive Systems using Term Rewriting Technology

K. Meinke, F. Niu

School of Computer Science and Communication,
Royal Institute of Technology, 100-44 Stockholm, Sweden,
karlm@nada.kth.se, niu@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 using term rewriting technology. A general model for a reactive system can be given by an extended Mealy automata (EMA) over an abstract data type (ADT). A finite state EMA over an ADT can be efficiently learned in polynomial time using the CGE regular inference algorithm, which builds a compact representation as a complete term rewriting system. We show how this rewriting system can be used to model check the learned automaton against a temporal logic specification by means of narrowing. Combining CGE learning with a narrowing model checker we obtain a new and general architecture for learning-based testing of reactive systems. We compare the performance of this LBT architecture against random testing using a case study.

1 Introduction

Learning-based testing (LBT) is an emerging technology for *specification-based black-box testing* that encompasses the three essential steps of : (1) test case generation (TCG), (2) test execution, and (3) test verdict (the oracle step). It has been successfully applied to testing *procedural systems* in [13] and [15], and *reactive systems* in [16]. The basic idea of LBT is to automatically generate a large number of high-quality test cases by combining a model checking algorithm with an *incremental model inference algorithm*, and integrating these two with the system under test (SUT) in an iterative loop. The use of incremental learning is critical in making this technology both fast and scalable to large systems under test (SUTs). Our previous research ([15] and [16]) has repeatedly shown that LBT has the capability to significantly outperform random testing in the speed with which it finds errors in an SUT.

For testing complex embedded software systems, there is a significant need to generate test cases over *infinite data types* such as integer and floating point types, and *abstract data types* (ADTs) such as strings, arrays, lists and various symbolic data types. Specification-based TCG is essentially a constraint solving problem. So this generalisation from finite to infinite and symbolic data types is highly non-trivial since the satisfiability problem for many logics over abstract

and infinite data types is undecidable. Thus a search for test cases is not guaranteed to terminate.

Model checking over abstract and infinite data types is therefore a state of the art problem. Recently some success has been achieved with the use of satisfiability modulo theories (SMT) solvers such as Z3 [5], which are based on heuristic techniques. However, an alternative approach is to use constraint solving based on a *narrowing algorithm*. Narrowing is a flexible technology, based on term rewriting, which is applicable to any data type for which we can find a complete (confluent and terminating) term rewriting system (see e.g. [1]). It has well understood theoretical properties such as completeness of solutions and conditions for termination. Narrowing has been successfully applied to model checking of infinite state systems in [6]. However, the use of narrowing for test case generation has not yet been considered. In fact, our aim in this paper is much wider. We will show that narrowing combines easily with symbolic learning algorithms for automata such as CGE [14] to yield a new *LBT architecture for specification-based testing of reactive systems computing over abstract data types*. Initial case studies suggest that despite the significant increase in the problem complexity, this new LBT architecture is also competitive with random testing.

The structure of this paper is as follows. In the remainder of Section 1 we review related work. In Section 2, we recall some essential mathematical preliminaries needed to discuss narrowing. In Section 3, we formalise a general model of a reactive system as an *extended Mealy automaton (EMA) over an abstract data type* (ADT). We introduce a *linear time temporal logic* (LTL) for such EMA, and we show how an LTL formula can be translated into constraint sets consisting of equations and negated equations. In Section 4, we present a model checking algorithm based on narrowing applied to a constraint set. In Section 5, we combine this model checking method with a symbolic automata learning algorithm (the CGE learning algorithm of [14]) to define a new LBT architecture for specification-based testing of reactive systems. In Section 6, we present a case study of this LBT architecture applied to testing the TCP protocol. Finally, in Section 7 we draw some conclusions and discuss open questions to be addressed by future work.

1.1 Related Work

In [16], LBT was applied to testing reactive systems modeled as Boolean Kripke structures. Our work here extends this previous work to allow symbolic and infinite data types. Even for finite data types, this approach simplifies the expression of control and data properties of an SUT. For this extension we use a more powerful symbolic learning algorithm, new model checking technology based on term rewriting theory, and a more powerful oracle construction for test verdicts.

Several previous studies, (for example [19], [9] and [20]) 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* (CE-

GAR) also combines learning and model checking, (see e.g. [3] and [2]). The LBT approach described here can be distinguished from these other approaches by: (i) an emphasis on testing rather than verification, (ii) the focus on incremental learning for efficient scalable testing, (iii) the use of narrowing as a model checking technique, and (iv) the introduction of abstract data types.

There is of course an extensive literature on the use of model checkers (without learning) to generate test cases for reactive systems. A recent survey is [8]. Generally this work emphasizes glass-box testing (so no learning is necessary), and the use of structural coverage measures to constrain the search space for test cases. Furthermore, behavioral requirements may or may not be present. By contrast, the LBT approach concerns black-box testing. Furthermore, in LBT behavioral requirements are always present, both to solve the oracle problem and to constrain the search space and guide the search for effective test cases.

In [21], black-box reactive system testing using learning but without model checking is considered. This is also shown to be more effective than random testing. Thus we can conclude that learning and model checking are two mutually independent techniques that can be applied to systems testing separately or together. In the long term we hope to show that the combination of both techniques is ultimately more powerful than using either one alone.

2 Mathematical Preliminaries and Notation

It is helpful to have some familiarity with the theories of abstract data types and term rewriting. Both use the notation and terminology of many-sorted algebra (see e.g. [17]). Let S be a finite set of sorts or types. An *S-sorted signature* Σ consists of an $S^* \times S$-indexed family of sets $\Sigma = \langle \Sigma_{w,s} \mid w \in S^*,\ s \in S \rangle$. For the empty string $\varepsilon \in S^*$, $c \in \Sigma_{\varepsilon,s}$ is a *constant symbol* of sort s. For $w = s_1,\ \ldots,\ s_n \in S^+$, $f \in \Sigma_{w,s}$ is a *function symbol* of *arity* n, *domain type* w and *codomain type* s. An *S-sorted Σ-algebra* A consists of sets, constants and functions that interpret Σ by a particular semantics. Thus A has an S-indexed family of sets $A = \langle A_s \mid s \in S \rangle$, where A_s is termed the *carrier set* of sort s. For each $s \in S$ and constant symbol $c \in \Sigma_{\varepsilon,s}$, $c_A \in A_s$ is a constant, and for each $w = s_1,\ \ldots,\ s_n \in S^+$ and each $f \in \Sigma_{w,s}$, $f_A : A_{s_1} \times \ldots \times A_{s_n} \to A_s$ is a function. Let $X = \langle X_s \mid s \in S \rangle$ be an S-indexed family of disjoint sets X_s of variables of sort s. We assume $X_s \cap \Sigma_{\varepsilon,s} = \emptyset$. The set $T(\Sigma,\ X)_s$ of all *terms of sort* $s \in S$ is defined inductively by: (i) $c \in T(\Sigma,\ X)_s$ for $c \in \Sigma_{\varepsilon,s}$, (ii) $x \in T(\Sigma,\ X)_s$ for $x \in X_s$, and (iii) $f(t_1,\ \ldots,\ t_n) \in T(\Sigma,\ X)_s$ for $f \in \Sigma_{w,s}$ $w = s_1,\ \ldots,\ s_n$ and $t_i \in T(\Sigma,\ X)_{s_i}$ for $1 \leq i \leq n$. We use $\equiv$ to denote syntactic equality between terms. An *equation* e (respectively *negated equation*) over Σ and X is a formula of the form $(t = t')$ (respectively $\neg(t = t')$) for $t,\ t' \in T(\Sigma,\ X)_s$. We let $\mathit{Vars}(t)$ (respectively $\mathit{Vars}(e)$, $\mathit{Vars}(\neg(e))$) denote the set of all variables from X occurring in t (respectively e, $\neg e$).

A *variable assignment* $\alpha : X \to A$ is an S-indexed family of mappings $\alpha_s : X_s \to A_s$. A *substitution* σ is a variable assignment $\sigma : X \to T(\Sigma,\ X)$ such that $\sigma_s(x) \neq x$ for just finitely many $s \in S$ and variables $x \in X_s$, and this

set of variables is the *domain* of σ_s. The result of applying a substitution σ to a term $t \in T(\Sigma,\ X)_{s'}$ is defined inductively in the usual way and denoted by $\sigma(t)$. If σ and σ' are substitutions then their *composition* $\sigma \circ \sigma'$ is defined by $\sigma \circ \sigma'(x) = \sigma(\sigma'(x))$. A *variable renaming* is a family of bijective substitutions $\sigma_s : X_s \to X_s$. A substitution σ is *more general* than a substitution τ, denoted $\sigma \leq \tau$ if there exists a substitution δ such that $\delta \circ \sigma = \tau$.

A *disunification problem* $S = \{\ (t_i\ Q_i\ t'_i)\ \mid\ i = 1,\ \ldots,\ n,\ n \geq 0,\ Q_i \in \{\ =\ ,\ \neq\ \}\ \}$ is a finite (possibly empty) set of equations and negated equations over Σ and X. A substitution $\sigma : X \to T(\Sigma,\ X)$ is a *syntactic unifier* of a set S if for all $1 \leq i \leq n$, if Q_i is $=$ then $\sigma(t_i) \equiv \sigma(t'_i)$, and if Q_i is $\neq$ then $\sigma(t_i) \not\equiv \sigma(t'_i)$. We let $U(S)$ denote the set of all syntactic unifiers of S. A unifier $\sigma \in U(S)$ is a *most general unifier* (mgu) if $\sigma \leq \tau$ for all $\tau \in U(S)$.

If $t \in T(\Sigma,\ X)_s$ is a term then $O(t)$ denotes the set of all *positions* in t, i.e. all nodes in the parse tree of t and is inductively defined by $O(c) = O(x) = \{\ \varepsilon\ \}$ and $O(f(t_1,\ \ldots,\ t_n)) = \{\ \varepsilon,\ k.\bar{i}\ \mid\ 1 \leq k \leq n,\ \bar{i} \in O(t_k)\ \}$. We write $t|_p$ for the *subterm of* t *found at position* $p \in O(t)$, and if $t|_p,\ u \in T(\Sigma,\ X)_s$ then $t[u]_p$ denotes the term obtained by replacing the subterm found at p in t by u. We say that $p \in O(t)$ is a *non-variable position* if $t|_p$ is not a variable, and let $\overline{O}(t)$ denote the set of all such non-variable positions.

A *term rewriting rule* is an expression of the form $l \to r$ for $l,\ r \in T(\Sigma,\ X)_s$ and $s \in S$ such that $\mathit{Vars}(r) \subseteq \mathit{Vars}(l)$ and a *term rewriting system* (TRS) R is a set of rewriting rules. If $\sigma_s : X_s \to X_s$ is a family of variable renamings then $\sigma(l) \to \sigma(r)$ is a *variant* of $l \to r$. The *rewrite relation* $\xrightarrow{R}$ associated with a TRS R is a binary relation on terms defined by $t \xrightarrow{R} t'$ if there exists a rule $l \to r \in R$, a position $p \in O(t)$ and a substitution σ such that $t|_p \equiv \sigma(l)$ and $t' \equiv t[\sigma(r)]_p$. We call $t \xrightarrow{R} t'$ a *rewrite step*. We let $\xrightarrow{R^*}$ denote the reflexive transitive closure of $\xrightarrow{R}$. A TRS R is *strongly normalising* if there is no infinite sequence of rewrite steps $t_0 \xrightarrow{R} t_1 \xrightarrow{R} t_2 \xrightarrow{R} \ldots$ and R is *confluent* (or *Church-Rosser*) if for any terms $t,\ t_1,\ t_2 \in T(\Sigma,\ X)_s$ if $t \xrightarrow{R^*} t_1$ and $t \xrightarrow{R^*} t_2$ then there exists $t' \in T(\Sigma,\ X)_s$ such that $t_1 \xrightarrow{R^*} t'$ and $t_2 \xrightarrow{R^*} t'$. A *complete* TRS is confluent and strongly normalising.

3 Mealy Automata over Abstract Data Types

In this section we formalise a general model of a reactive system as an extended Mealy automaton (EMA) over an abstract data type. We then introduce the syntax and semantics of a linear time temporal logic (LTL) as a language for expressing user requirements on EMA. Finally, we define a syntactic translation of LTL into equations and negated equations, and establish the soundness and completeness of this translation with respect to satisfiability.

We can model a Mealy automaton over an abstract data type as a many-sorted algebraic structure by considering inputs, states and outputs as distinguished data sorts (or types). The input and output types will be typically chosen

from some well known data types such as *int*, *string*, *array*, *list* etc. that provide a high level of data abstraction.

Definition 1. *A* **signature** $(S,\ \Sigma,\ input,\ output)$ *for an extended Mealy automaton is a four-tuple, where* $S = \{\ state,\ s_1,\ \ldots,\ s_n\ \}$ *is a sort set,* Σ *is an* S*-sorted signature with distinguished constant and function symbols*

$$q^0 \in \Sigma_{\varepsilon,state},\ \delta \in \Sigma_{state\ input,state},\ \lambda \in \Sigma_{state\ input,output},$$

and $input,\ output \in \{\ s_1,\ \ldots,\ s_n\ \}$ *are distinguished input and output types.*

Definition 2. *Let* $(S,\ \Sigma,\ input,\ output)$ *be a signature for an EMA. An* **extended Mealy automaton** A *(of signature* Σ*) is an* S*-sorted* Σ *algebra* A*.*

As usual q^0_A is the initial state, $\delta_A : A_{state} \times A_{input} \to A_{state}$ is the state transition function, and $\lambda_A : A_{state} \times A_{input} \to A_{output}$ is the output function.

We define the extended state transition and output functions

$$\delta^*_A : A_{state} \times A^*_{input} \to A_{state}, \qquad \lambda^*_A : A_{state} \times A^+_{input} \to A_{output}$$

in the usual way for any $q \in A_{state}$, $\bar{i} \in A_{input}$ and $j \in A_{input}$ by $\delta^*_A(q,\ \varepsilon) = q$ and $\delta^*_A(q,\ \bar{i}\ .\ j) = \delta_A(\ \delta^*_A(q,\ \bar{i}),\ j)$, also $\lambda^*_A(q,\ \bar{i}\ .\ j) = \lambda_A(\ \delta^*_A(q,\ \bar{i}),\ j)$.

If A_{state} is finite then A is termed a *finite state EMA* , otherwise A is termed an *infinite state EMA*.

Next we introduce a linear time temporal logic (LTL) that can be used to express user requirements on EMA. For this it is necessary to integrate the underlying data type signature Σ in an appropriate way. In the sequel we assume that $(S,\ \Sigma,\ input,\ output)$ is a given EMA signature. Let $X = \langle X_s \mid s \in S - \{\ state\ \}\rangle$ be any indexed family of sets X_s of variable symbols of sort s. We assume that $in \in X_{input}$ and $out \in X_{output}$ are two distinguished variable symbols.

Definition 3. *The set* $LTL(\Sigma,\ X)$ *of all* **linear temporal logic formulas** *over* Σ *and* X *is defined to be the smallest set of formulas containing the atomic proposition* **true** *and all equations* $(t = t')$ *for each sort* $s \in S - \{\ state\ \}$ *and all terms* $t,\ t' \in T(\Sigma,\ X)_s$*, which is closed under negation* $\neg$*, conjunction* $\wedge$ *, disjunction* $\vee$ *, and the next* $\mathbf{X}$*, always future* $\mathbf{G}$*, sometime future* $\mathbf{F}$*, always past* $\mathbf{G^{-1}}$*, and sometime past* $\mathbf{F^{-1}}$ *temporal operators*

As usual, $\mathbf{X}(\phi)$ denotes that ϕ is true in the next time instant, while $\mathbf{G}(\phi)$ (respectively $\mathbf{F}(\phi)$) denotes that ϕ is always (respectively at some time) true in the future of a run. On the other hand $\mathbf{G^{-1}}(\phi)$ (respectively $\mathbf{F^{-1}}(\phi)$) denotes that ϕ was always (respectively at some time) true in the past of a run. While not strictly necessary, including these *past operators* makes this LTL exponentially more succinct, as shown in [12]. This increases the efficiency of our narrowing model checker. We let $(\ \phi \implies \psi\ )$ denote the formula $(\ \neg\phi \vee \psi\ )$, and $t \neq t'$ denotes $\neg(t = t')$. Then for example, the formula

$$\mathbf{G}(\ (in = x) \wedge \mathbf{X}(\ (in = y) \implies \mathbf{X}(out = x + y)\ )\ )$$

is an LTL formula that expresses that at all times, if the current input is x and next input is y then in two time steps from now the output will be the sum $x+y$. So in this LTL we can express both control and data properties of reactive systems.

Definition 4. *Let A be an EMA, let $n \in \mathbb{N}$, let $\overline{i} = i_0, i_1, \ldots \in A^{\omega}_{input}$ be an infinite sequence of inputs for A, and let $Val_{A,\alpha} : T(\Sigma, X)_s \to A_s$ be the valuation mapping on terms given a variable assignment $\alpha : X \to A$. We define the* **satisfaction relation** *$A, n, \overline{i}, \alpha \models \phi$ for each formula $\phi \in LTL(\Sigma, X)$ by induction.*

*(i) $A, n, \overline{i}, \alpha \models$ **true**.*

(ii) $A, n, \overline{i}, \alpha \models t = t'$ if, and only if, $Val_{A,\beta}(t) = Val_{A,\beta}(t')$, where

$$\beta = \alpha[\ in \mapsto i_n,\ out \mapsto \lambda_A(\delta^*_A(q^0_A,\ i_0,\ \ldots,\ i_{n-1}),\ i_n))\ ].$$

(iii) $A, n, \overline{i}, \alpha \models \neg\phi \ \Leftrightarrow\ A, n, \overline{i}, \alpha \not\models \phi$.

(iv) $A, n, \overline{i}, \alpha \models \phi \wedge \psi$ if, and only if, $A, n, \overline{i}, \alpha \models \phi$ and $A, n, \overline{i}, \alpha \models \psi$.

(v) $A, n, \overline{i}, \alpha \models \phi \vee \psi$ if, and only if, $A, n, \overline{i}, \alpha \models \phi$ or $A, n, \overline{i}, \alpha \models \psi$.

(vi) $A, n, \overline{i}, \alpha \models \mathbf{X}\phi$ if, and only if, $A, n+1, \overline{i}, \alpha \models \phi$.

(vii) $A, n, \overline{i}, \alpha \models \mathbf{G}\phi$ if, and only if, for all $k \geq n$ $A, k, \overline{i}, \alpha \models \phi$.

(viii) $A, n, \overline{i}, \alpha \models \mathbf{F}\phi$ if, and only if, for some $k \geq n$, $A, k, \overline{i}, \alpha \models \phi$.
(ix) $A, n, \overline{i}, \alpha \models \mathbf{G^{-1}}\phi$ if, and only if, for all $k \leq n$ $A, k, \overline{i}, \alpha \models \phi$.

(x) $A, n, \overline{i}, \alpha \models \mathbf{F^{-1}}\phi$ if, and only if, for some $k \leq n$ $A, k, \overline{i}, \alpha \models \phi$.

A formula $\phi \in LTL(\Sigma, X)$ is **satisfiable** *with respect to A if there exists an infinite sequence $\overline{i} \in A^{\omega}_{input}$ and an assignment $\alpha : X \to A$ such that $A, 0, \overline{i}, \alpha \models \phi$.*

As is well known, for every formula $\phi \in LTL(\Sigma, X)$ there exists a logically equivalent formula $\phi' \in LTL(\Sigma, X)$ in *negation normal form* (NNF) where negations only occur in front of atomic subformulas. To solve LTL formulas by narrowing we translate an NNF formula ϕ into a finite set $S = \{\ S_1, \ldots, S_n\ \}$ of *constraint sets*, where a constraint set S_i consists of equations and negated equations. This translation requires an additional set $\overline{X} = \{\ \overline{x_i} \mid x_i \in X_{input}\ \}$ of fresh variable symbols ranging over input sequence elements.

Definition 5. *Let A be an EMA, and let loopbound be the length of the longest loop-free path in A. For each NNF formula $\phi \in LTL(\Sigma, X)$ we define the* **satisfiability set** *$SatSet_n(\phi)$ as a finite collection of constraint sets by structural induction on ϕ.*

$$SatSet_n(t\ Q\ t') = \{\ \{\ (\theta_n(t)\ Q\ \theta_n(t'))\ \}\ \}$$

where $Q \in \{\ =, \neq\ \}$ and θ_n is the substitution defined by

$$\theta_n = \{\ in \to \overline{x}_n,\ out \to \lambda(\delta^*(q^0,\ \overline{x}_0,\ \ldots,\ \overline{x}_{n-1}),\ \overline{x}_n)\ \}$$

$$SatSet_n(\ \phi\ \wedge\ \psi\ ) = \{\ S_\phi \cup S_\psi\ \mid\ S_\phi \in SatSet_n(\phi),\ S_\psi \in SatSet_n(\phi)\ \}$$

$$SatSet_n(\ \phi\ \vee\ \psi\ ) = SatSet_n(\phi) \cup SatSet_n(\neg\phi\ \wedge\ \psi)$$

$$SatSet_n(\ \mathbf{X}(\phi)\ ) = SatSet_{n+1}(\phi)$$

$$SatSet_n(\ \mathbf{F}(\phi)\ ) = \bigcup_{k=0}^{loopbound} SatSet_{n+k}(\phi)$$

$$SatSet_n(\ \mathbf{F^{-1}}(\phi)\ ) = \bigcup_{k=0}^{loopbound} SatSet_{n-k}(\phi)$$

$$SatSet_n(\ \mathbf{G}(\phi)\ ) =$$

$$\bigcup_{h=0}^{loopbound} \bigcup_{l=1}^{loopbound} \{\quad \{\ \overline{x}_{n+h+k.l+i} = \overline{x}_{n+h+i}\ \mid\ 1 \le k,\ 0 \le i \le l-1\ \}$$

$$\cup\{\ \delta^*(q^0,\ \overline{x}_0,\ \ldots,\ \overline{x}_{n+h+l-1}) = \delta^*(q^0,\ \overline{x}_0,\ \ldots,\ \overline{x}_{n+h-1})\ \}$$

$$\cup \bigcup_{i=0}^{h+l-1} S_i\ \mid\ S_i \in SatSet_{n+i}(\phi),\ 0 \le i \le h+l-1\quad \}$$

$$SatSet_n(\ \mathbf{G^{-1}}(\phi)\ ) = \{\ \bigcup_{i=0}^{n} S_i\ \mid\ S_i \in SatSet_i(\phi)\ \ for\ \ 0 \le i \le n\ \}$$

The translation $SatSet_n(\phi)$ preserves solutions of ϕ as follows.

Theorem 1. *Let A be an EMA, and loopbound be the length of the longest loop-free path in A. Let $\phi \in LTL(\Sigma,\ X)$ be in NNF, and let $n \in \mathbb{N}$.*

(i) (Soundness of Translation) For any assignment $\alpha : X \to A$ and input sequence $\overline{i} = i_0,\ i_1,\ \ldots \in A^\omega_{input}$ there exists $S \in SatSet_n(\phi)$ such that

$$A,\ n,\ \overline{i},\ \alpha \models \phi \quad \Longrightarrow \quad A,\ \beta(\overline{i}),\ \alpha \models S,$$

where the assignment $\beta(\overline{i}) : \overline{X} \to A_{input}$ is given by $\beta(\overline{i})(\overline{x_n}) = i_n$.
(ii) (Completeness of Translation) For any assignments $\alpha : X \to A$ and $\beta : \overline{X} \to A_{input}$ if there exists $S \in SatSet_n(\phi)$ such that $A,\ \beta,\ \alpha \models S$ then there exists an input sequence $\overline{\beta} \in A^\omega_{input}$ such that

$$A,\ n,\ \overline{\beta},\ \alpha \models \phi.$$

Thus by Theorem 1, to solve an NNF formula ϕ it is necessary and sufficient to solve one of the constraint sets $S_1,\ \ldots,\ S_n \in SatSet_0(\phi)$. We will consider a method to solve constraint sets by narrowing in the next section.

4 Model Checking by Narrowing

The problem of finding solutions to a set $\{\ t_1 = t'_1,\ \ldots,\ t_n = t'_n\ \}$ of equations is the well known unification problem about which much has been written (see e.g. [1]). More generally, in the case that a set $\{\ t_1 = t'_1,\ \ldots,\ t_n = t'_n,\ u_1 \neq u'_1,\ \ldots,\ u_n \neq u'_n\ \}$ of equations and negated equations must be solved, this problem is known as the disunification problem (see e.g. [4]).

Let Σ be a many-sorted data type signature and E be an equational data type specification having a complete rewrite system R. Then the disunification problem is complicated by the fact that we seek solutions modulo R (and hence E) in the following sense.

Definition 6. *Let R be a term rewriting system. The relation of R-***conversion** *denote by $=_R$ is the reflexive symmetric and transitive closure of $\xrightarrow{R}$. Let $S = \{\ (t_i\ Q_i\ t'_i)\ |\ i = 1,\ \ldots,\ n,\ n \geq 0,\ Q_i \in \{\ =,\ \neq\ \}\ \}$ be a disunification problem. A substitution $\sigma : X \to T(\Sigma,\ X)$ is an R-***unifier** *of S if for all $1 \leq i \leq n$, if Q_i is $=$ then $\sigma(t_i) =_R \sigma(t'_i)$, and if Q_i is $\neq$ then $\sigma(t_i) \neq_R \sigma(t'_i)$. We let $U_R(S)$ denote the set of all R-unifiers of S.*

In the special case where $E = R = \emptyset$, these problems are known as syntactic unification and syntactic disunification, and both problems are decidable. However in many important cases, both the unification and disunification problems are undecidable. Nevertheless, these problems are semidecidable and one can consider complete search algorithms which always terminate when a solution is to be found. The method of narrowing gives such a complete search algorithm, and can be used whenever the data type specification E can be represented by a complete term rewriting system R.

The basic idea of narrowing is a systematic search of the space of possible solutions using the rules of R. If some equation $t_i = t'_i$ cannot be syntactically unified then we can apply a substitution $\sigma : X \to T(\Sigma,\ X)$ to t_i (or t'_i) such that the resulting term $\sigma(t_i)$ is not in R normal form and then reduce this in one step. This requires unifying t_i (or t'_i) with the left hand side l of a rule $l \to r$ in R, and replacing with a suitable instance of r so that a new equation is obtained. A similar process can be applied to negated equations, and can be iterated for all formulas until syntactic unification of the entire set becomes possible, though the narrowing process may not terminate. If it terminates, the resulting sequence of substitutions $\sigma_k : X \to T(\Sigma,\ X)$ can be composed together with the final syntactic unifier θ to yield an R-unifier.

Definition 7. *We say that a term t is R-***narrowable** *into a term t' if there exists a non-variable position $p \in \overline{O}(t)$, a variant $l \to r$ of a rewrite rule in R and a substitution σ such that:*

(i) σ is a most general syntactic unifier of $t|_p$ and l, and

(ii) $t' \equiv \sigma(t[r]_p)$.

*We write $t \rightsquigarrow_{[p,l\to r,\sigma]} t'$ or simply $t \rightsquigarrow_\sigma t'$. The relation $\rightsquigarrow$ is called R-***narrowing**.

The R-narrowing relation on terms can be extended to equations and negated equations in an obvious way. A formula $(t\ Q\ t')$ (where Q is $=$ or $\neq$) is R-narrowable into a formula $(u\ Q\ u')$ if there exists a variant $l \to r$ of a rewrite rule in R and a substitution σ such that either $t \leadsto_{[p,l\to r,\sigma]} u$ for some non-variable occurrence $p \in \overline{O}(t)$ or $t' \leadsto_{[q,l\to r,\sigma]} u'$ for some non-variable occurrence $q \in \overline{O}(t')$. We write $(t\ Q\ t') \leadsto_{[p,l\to r,\sigma]} (u\ Q\ u')$ or simply $(t\ Q\ t') \leadsto_{\sigma} (u\ Q\ u')$. Generalising the R-narrowing relation still further to sets of equations and negated equations we will write $S \leadsto_{[p,l\to r,\sigma]} S'$ or $S \leadsto_{\sigma} S'$.

We can relativise the concept of a substitution σ being more general than a substitution τ (c.f. Section 2) to R as follows. Let V be any S-indexed family of sets V_s of variables. We define $\sigma \leq_R \tau[V]$ if for some substitution δ, $\delta \circ \sigma(x) =_R \tau(x)$ for all $s \in S$ and $x \in V_s$. Now we can discuss the soundness and completeness of narrowing.

Theorem 2. *Let* $S = \{\ (t_i\ Q_i\ t_i')\ \mid\ i = 1,\ \ldots,\ n,\ n \geq 0,\ Q_i \in \{\ =,\ \neq\ \}\ \}$ *be a disunification problem.*

(i) (Soundness of Narrowing) Let

$$S \leadsto_{\sigma_1} S_1 \leadsto_{\sigma_2},\ \ldots,\ \leadsto_{\sigma_n} S_n$$

be a terminated R-narrowing derivation such that S_n is syntactically unifiable by a substitution θ. Then $\theta \circ \sigma_n \circ \ldots \circ \sigma_1$ is an R-unifier of S.

(ii) (Completeness of Narrowing) If S is R-unifiable then let ρ be any R-unifier and V be a finite set of variables containing $Vars(S)$. There exists a terminated R-narrowing derivation

$$S \leadsto_{\sigma_1} S_1 \leadsto_{\sigma_2},\ \ldots,\ \leadsto_{\sigma_n} S_n$$

such that S_n is syntactically unifiable. Let μ be a most general syntactic unifier of S_n then $\mu \circ \sigma_n \circ \ldots \circ \sigma_1 \leq \rho[V]$.

The search space of narrowing is large and narrowing procedures frequently fail to terminate. Many proposals have been made to increase the efficiency of narrowing. One important restriction on the set of occurrences available for narrowing, termed *basic narrowing*, was introduced in [11], and has since been widely studied, e.g. [18].

A basic narrowing derivation is very similar to a narrowing derivation as given in Definition 7 above. However, in a basic narrowing derivation, narrowing is never applied to a subterm introduced by a previous narrowing substitution. This condition is quite complex to define precisely, and the reader is referred to [11].

Theorem 4 of [11] can be used to show that basic narrowing for equations and negated equations is also sound and complete in the sense of Theorem 2. However, for basic narrowing [11] also establishes sufficient conditions to guarantee termination. This property is important in test case generation, where we need to know if a test case exists at all.

Theorem 3. *([11]) Let* $R = \{\ l_i \ \rightarrow\ r_i \ \mid\ i = 1,\ \ldots,\ n\ \}$ *be a complete rewrite system such that any basic* R*-narrowing derivation from any of the* r_i*'s terminates. Then every* R*-narrowing derivation terminates.*

Many examples of TRS satisfying Theorem 3 are known, including TRS for all finite ADTs. This general termination result can be applied to establish that basic R-narrowing yields a decision procedure for LTL model checking (i.e. basic R-narrowing is sound, complete and terminating) because of the following new result about the CGE symbolic learning algorithm.

Theorem 4. *Let* $(R_n^{state}, R_n^{output})$ *be the output of the CGE learning algorithm after a sequence of* n *observations of the I/O behavior of an EMA* A*. Then* $R_n = R_n^{state} \cup R_n^{output}$ *is a complete rewrite system and every* R_n*-narrowing derivation terminates.*

Proof. Proposition 4.5 of [14] establishes that R_n is complete. To establish termination, consider that every rule $l \ \rightarrow\ r \in R_n$ is ground by Definitions 3.12 and 4.4 of [14]. Hence the result is a special instance of Theorem 3 above and Example 3 in [11].

We have constructed an implementation of model checking by basic narrowing. We explain how this is integrated into learning-based testing in Section 5.

5 An LBT Architecture for Testing Reactive Systems

Learning-based testing (LBT) is a general paradigm for black-box specification-based testing that requires three basic 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.

Given such components, the paradigm provides a *heuristic iterative method* to search for and automatically generate a sequence of test cases. The basic idea is to *incrementally learn* an approximating sequence of models M_i for $i = 1,\ 2,\ \ldots$ of the unknown SUT S by using test cases as queries. During this learning process, we model check each approximation M_i on-the-fly searching for counterexamples to the validity of *Req*. Any such counterexample can be confirmed as a true negative by taking it as the next test case. At step i, if model checking does not produce any counterexamples then to proceed with the iteration, the next test case is constructed by another method, e.g. randomly.

In [16], LBT was applied to testing reactive systems modeled as Boolean Kripke structures. In this paper we consider the case where the SUT S is a reactive system that can be modeled by an EMA over the appropriate abstract data types, and *Req* is an LTL formula over the same data types. Thus we extend the scope of our previous work to deal with both control and data by applying new learning algorithms and model checking technology.

For LBT to be effective at finding errors quickly, it is important to use an incremental learning algorithm. In [16] this was empirically demonstrated by using the IKL incremental learning algorithm for Boolean Kripke structures. However, learning algorithms for finite data types such as IKL do not extend to infinite data types. The CGE learning algorithm of [14] was designed to implement learning EMA over abstract data types. Furthermore, this algorithm is incremental since its output is a sequence of representations $R_1,\ R_2,\ \ldots$ of the hypothesis EMA $M_1,\ M_2,\ \ldots$ which are the approximations to S. Each representation R_i is a complete TRS that encodes M_i as the corresponding quotient of the prefix tree automaton. Details of this representation can be found in [14]. Furthermore, CGE has many technical advantages over IKL. For example, the number of queries (test cases) between construction of successive approximations R_k and R_{k+1} can be arbitrarily small and even just one query. By contrast, IKL and other table based learning algorithms usually have intervals of tens or hundreds of thousands of queries between successive approximations of large SUTs. As a consequence, model checking can only be infrequently applied.

The input to CGE is a series of pairs $(\overline{i_1},\ \overline{o_1}),\ (\overline{i_2},\ \overline{o_2}),\ \ldots$ consisting of a query string $\overline{i_k}$ for S and the corresponding output string $\overline{o_k}$ from S. In an LBT setting, the query strings $\overline{i_k}$ come from model checker counterexamples and random queries. Finite convergence of the sequence $R_1,\ R_2,\ \ldots$ to some TRS R_n can be guaranteed if S is a finite state EMA (see [14]) and the final hypothesis automaton M_n is behaviorally equivalent with S. So with an increasing number of queries, it becomes more likely that model checking will produce a true negative if one exists, as the unknown part of S decreases to nothing. By combining CGE with the narrowing model checker of Section 4, we arrive at a new LBT architecture for reactive systems shown in Figure 1.

Figure 1 illustrates the basic iterative loop of the LBT architecture between: (i) learning, (ii) model checking, (iii) test execution, and (iv) test verdict by an oracle. This iterative loop is terminated by an *equivalence checker*. This component can be used to detect that testing is complete when the SUT is sufficiently small to be completely learned. Obviously testing must be complete by the time we have learned the entire SUT, since model checking by narrowing is solution complete. The equivalence checker compares the current model representation R_k with S for behavioural (rather than structural) equivalence. A positive result from this equivalence test stops all further learning, after one final model check of R_k searches for any residual errors. In practical applications of LBT technology, real world SUTs are usually too large to be completely learned. It is this pragmatic constraint that makes incremental learning algorithms necessary for scalable LBT. In such cases the iterative loop must ultimately be terminated by some other method such as a time constraint or a coverage measure.

Figure 1 shows that the current model R_k is also passed from the CGE algorithm to the basic narrowing model checker, together with a user requirement represented as an LTL formula ϕ. This formula is fixed for a particular testing session. The model checker uses R_k to identify at least one counterexample to ϕ as an *input sequence* $\overline{i_{k+1}}$ over the underlying input data type. If ϕ is a safety

formula then this input sequence will usually be finite

$$\overline{i_{k+1}} = (\ i_1,\ \ldots,\ i_j\ ) \in T(\Sigma)^*_{input}.$$

If ϕ is a liveness formula then the input sequence $\overline{i_{k+1}}$ may be finite or infinite. Since infinite counterexamples to liveness formulas can be represented as infinite strings of the form $\overline{x}\ \overline{y}^{\omega}$, in this case $\overline{i_{k+1}}$ is truncated to a finite initial segment that would normally include at least one execution of the infinite loop $\overline{y}^{\omega}$, such as $\overline{i_{k+1}} = \overline{x}\ \overline{y}$. Observing the failure of infinite test cases is of course impossible, and the LBT architecture implements a compromise solution that executes the truncated input sequence only, and issues a warning rather than a definite test failure.

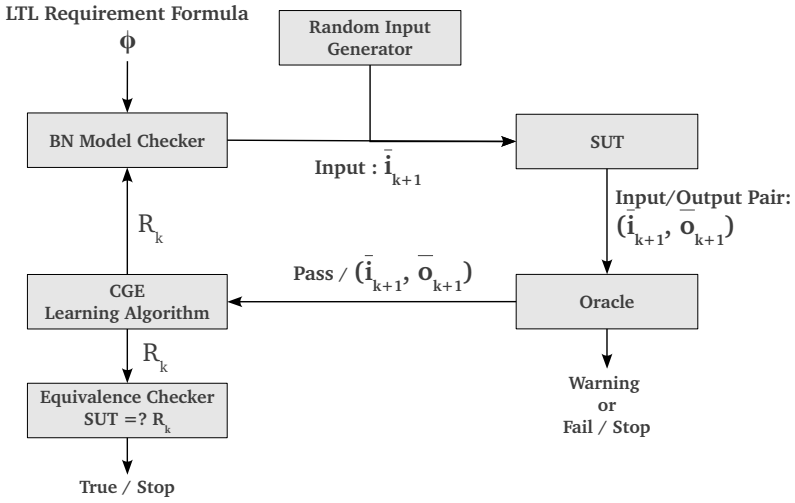

Fig. 1. A Learning-based Testing Architecture for Reactive Systems

If the next test case $\overline{i_{k+1}}$ cannot be constructed by model checking then in order to proceed with iterative testing a *random input string generator* (see Figure 1) is used to generate $\overline{i_{k+1}}$. During this random generation process, any random string that has been used as a previous test case is discarded to avoid redundant replicate tests.

Thus from one of two possible sources (model checking or random generation) a new test case $\overline{i_{k+1}}$ is constructed. Figure 1 shows that this new test case $\overline{i_{k+1}}$ is then executed on the SUT S to yield an *actual output sequence* $\overline{o_{k+1}} = o_1,\ \ldots,\ o_j$. The pair $(\ \overline{i_{k+1}},\ \overline{o_{k+1}}\ )$ is then passed to an oracle to compute the $k+1$-th test verdict.

The oracle we have developed for this LBT architecture is more powerful than the one described in [16], and is based on the following two step process.

Step 1. A test verdict can often be derived quickly and simply by computing a *predicted output* $\overline{p_{k+1}} = p_1, \ldots, p_j$ obtained by simulating the behavior of M_k on $\overline{i_{k+1}}$. This is easily derived by applying the TRS R_k to rewrite the input string $\overline{i_{k+1}}$ into its normal form, i.e. $\overline{i_{k+1}} \xrightarrow{R_k}{}^{*} \overline{p_{k+1}}$. Recall that R_k is a complete TRS, so this normal form is always well defined. We then implement a simple Boolean test $\overline{o_{k+1}} = \overline{p_{k+1}}$. If this equality test returns *true* and the test case $\overline{i_{k+1}}$ was originally a finite test case then we can conclude that the test case $\overline{i_{k+1}}$ is definately *failed*, since the behaviour $\overline{p_{k+1}}$ is by construction a counterexample to the correctness of ϕ. In this case we can decide to stop testing. If the equality test returns *true* and the test case $\overline{i_{k+1}}$ was finitely truncated from an infinite test case (a counterexample to a liveness requirement) then the verdict is weakened to a *warning* (but testing is not stopped). This is because the most we can conclude is that we have not yet seen any difference between the observed behaviour $\overline{o_{k+1}}$ and the incorrect behaviour $\overline{p_{k+1}}$.

Step 2. If the Boolean test $\overline{o_{k+1}} = \overline{p_{k+1}}$ in Step 1 returns *false* then more work is needed to determine a verdict. We must decide whether the observed output $\overline{o_{k+1}}$ is some other counterexample to the correctness of ϕ than $\overline{p_{k+1}}$. This situation easily occurs when the requirement ϕ is a loose specification of the SUT behavior, such as a constraint or value interval. In this case we can evaluate the requirement formula ϕ instantiated by the input and actual output sequences $\overline{i_{k+1}}$ and $\overline{o_{k+1}}$ to determine whether ϕ is true or false. For this we perform a translation similar to $SatSet_0(\phi)$ but with the variables $\overline{x}_n$ and *out* instantiated by the appropriate components of $\overline{i_{k+1}}$ and $\overline{o_{k+1}}$ respectively. We then evaluate all resulting sets of variable free equations and negated equations by rewriting. By Theorem 1, this approach will produce a correct verdict if $\overline{o_{k+1}}$ is a counterexample to ϕ.

Note that while Step 1 was already described in [16], Step 2 is an additional and more powerful step made possible by the translation of LTL into equational logic, and the use of term rewriting to implement the latter.

When the conditions of Theorem 3 are satisfied by the underlying data type then the LBT architecture of Figure 1 can be proven to terminate since the CGE algorithm correctly learns in the limit and basic narrowing is terminating and solution complete. A detailed analysis of this property will be published in an extended version of this paper. The argument is similar to that presented in [16].

6 A Case Study of LBT for Reactive Systems

Since the overhead of model checking an EMA by narrowing is high, it is important to study the performance of our LBT architecture in practice using case studies. Now many communication protocols can be modeled as EMA computing over (freely generated) symbolic data types. Thus the LTL decidability results of Section 4 apply to this class of examples.

The Transmission Control Protocol (TCP) is a widely used transport protocol over the Internet. We present here a performance evaluation of our LBT

architecture applied to testing a simplified model of the TCP/IP protocol as the 11 state EMA shown in Figure 2.

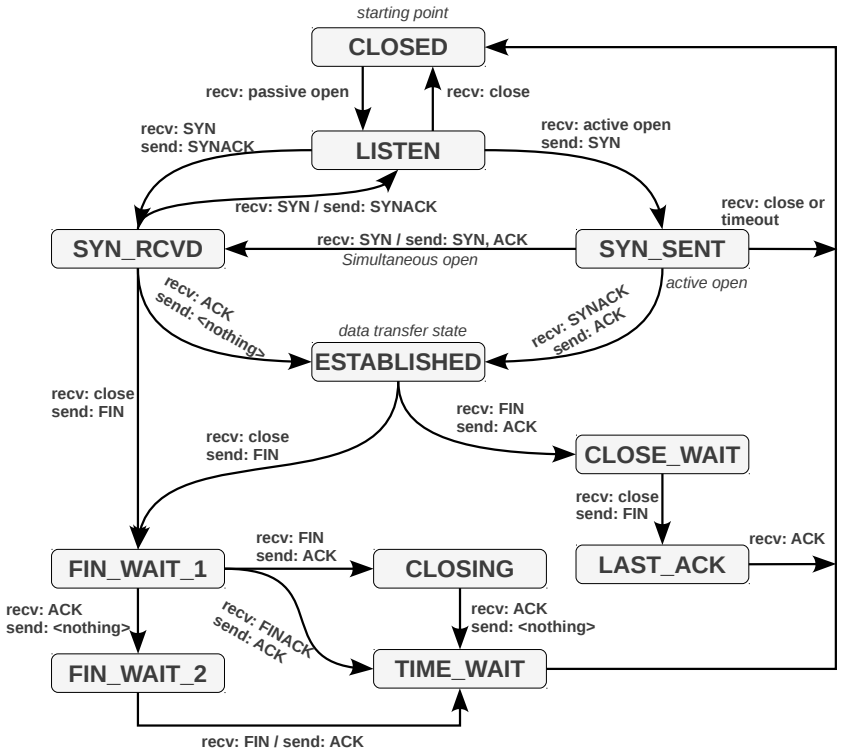

Fig. 2. TCP Mealy Machine Model

In this performance evaluation, we considered the fault detection capability of LBT compared with random testing. A coverage comparison of learning-based testing with random testing is for example [21], which even considers the same TCP case study. The methodology for comparison was to start from the concrete model of TCP in Figure 2 and consider a variety of correctness requirements as LTL formulas (including use cases). We then injected transition mutations into the SUT which falsified each individual requirement separately. In this way, several different kinds of bugs were introduced into the protocol model, such as mutating the input/output on transitions, adding extraneous transitions or states and so on. Some of these artificial mutations reflect realistic defects that have been discovered in several TCP/IP implementations [10].

Below we informally define five requirements on the TCP/IP protocol and give an LTL formalization of each.

1. Use case. Whenever the entity receives an *active_ open* and sends out a *SYN*, the entity will send out a *SYNACK* if it receives a *SYN*, or send out an *ACK* if it receives a *SYNACK*, and send nothing when receiving other inputs.

$$G(\,(in = active\_open \wedge out = \text{SYN}) \rightarrow$$

$$X((in = \text{SYN} \rightarrow out = \text{SYNACK}) \wedge (in = \text{SYNACK} \rightarrow out = \text{ACK}))\,)$$

2. Use case. Whenever the entity receives an *active_ open* and sends out a *SYN* and then receives a *SYNACK*, the entity will send out an *ACK* and then will send out an *ACK* if it receives a *FIN*.

$$G(\,(in = active\_open \wedge out = \text{SYN} \wedge X\ in = \text{SYNACK}) \rightarrow$$

$$(X\ out = \text{ACK} \wedge X^2\,(in = \text{FIN} \rightarrow out = \text{ACK}))\,)$$

3. Use case. Whenever the entity performs the IO (*active_ open*, *SYN*) and receives *SYNACK* followed by *FIN* it will send out *ACK* followed by *ACK* and then send out *FIN* if it receives *CLOSE*.

$$G(\,(in = active\_open \wedge out = \text{SYN} \wedge X\ in = \text{SYNACK} \wedge X^2\ in = \text{FIN}) \rightarrow$$

$$(X\ out = \text{ACK} \wedge X^2\ out = \text{ACK} \wedge X^3\,(in = close \rightarrow out = \text{FIN}))\,)$$

4. Whenever the entity receives a *close* and sends out a *FIN*, or receives a *FIN* and sends out an *ACK*, the entity has either sent a *passive_ open* or received an *active_ open* before, and either sent or received a *SYN* before.

$$G(\,((in = close\ \wedge\ out = \text{FIN}) \vee (in = \text{FIN}\ \wedge\ out = \text{ACK})) \rightarrow$$

$$(F^{-1}\,(in = pass\_open \vee in = active\_open) \wedge F^{-1}\,(in = \text{SYN} \vee out = \text{SYN}))\,)$$

5. Whenever the entity performs the IO (*FINACK*, *ACK*) it must have received or sent *SYN* in the past and performed the IO (*close*, *FIN*) in the past.

$$G(\,(in = \text{FINACK}\ \wedge\ out = \text{ACK}) \rightarrow$$

$$(F^{-1}\,(in = \text{SYN}\ \vee\ out = \text{SYN}) \wedge F^{-1}\,(in = close\ \wedge\ out = \text{FIN}))\,)$$

6.1 Results and Analysis

To compare LBT with random testing on the TCP/IP stack model, we measured two related parameters, namely: (i) the time t_{first} (in seconds), and (ii) the total number of queries (i.e. test cases) Q_{first} needed to *first discover an injected error in the SUT*. To conduct random testing, we simply switched off the CGE and model checker algorithms. The performance of LBT is non-deterministic due to the presence of random queries. Therefore each value of t_{first} and Q_{first} is an average obtained from over 1000 LBT runs using the same injected error.

Requirement	Random Testing		LBT				
	Q_{first}	$t_{first}(sec)$	Q_{first}	$t_{first}(sec)$	MCQ	RQ	Hyp_size
Req 1	101.4	0.11	19.11	0.07	8.12	10.99	2.2
Req 2	1013.2	1.16	22.41	0.19	9.11	13.3	2.8
Req 3	11334.7	36.7	29.13	0.34	10.3	18.83	3.1
Req 4	582.82	1.54	88.14	2.45	23.1	65.04	3.3
Req 5	712.27	2.12	93.14	3.13	31.8	61.34	4.1

Table 1. Random testing versus LBT: a performance comparison

The results of testing the requirements Req1 to Req 5 are listed in Table 1. Note that Q_{first} is the combined sum of the number of model checking queries MCQ and random queries RQ. These are also listed in columns 6 and 7 to provide deeper insight into the strengths and weaknesses of our method. In the final column, Hyp_size is the state space size of the learned hypothesis automaton at time t_{first}. Since Hyp_size is always considerably less than 11 (the state space size of our SUT), this confirms the advantages of using an incremental learning algorithm such as CGE.

We wish to draw two main conclusions from Table 1.

(i) At the level of *logical performance*, (comparing Q_{first} for LBT against Q_{first} for random testing) we see that LBT *always* finds errors with significantly fewer test cases ranging between 0.25% and 18% of the number required by random testing. Therefore, if the overheads of model checking and learning can be reduced then LBT also has the *potential* to outperform random testing in real-time performance.

(ii) At the level of *real-time performance* (comparing t_{first} for LBT against t_{first} for random testing) we see that LBT is *often but not always* significantly faster than random testing, ranging between 0.9% and 160% of the time required by random testing. This reflects the actual real-time overhead of performing both model checking and learning for the SUT and each requirement.

Looking more closely at the results for Reqs 4 and 5, where LBT is somewhat slower than random testing, we can gain deeper insight into these real-time performance issues. For Reqs 4 and 5 both the values MCQ and the ratios RQ/MCQ are significantly higher than for Reqs 1, 2 and 3. In these cases, basic narrowing is performing a large number of constraint solving tasks on unsatisfiable sets of constraints. However, basic narrowing fails very slowly when no solutions can be found. After this, random test cases are applied to proceed with the task of learning the SUT, but these do not necessarily test the actual requirements.

These preliminary results are nevertheless promising, and based on them we make some suggestions for how to further improve narrowing in Section 7. Thus we can improve the overall real-time performance of our current LBT architecture to achieve a real-time performance closer to the logical performance. It should also be pointed out that as real-time measurement involves factors such

as efficiency of implementation, there exists further scope for improvement on the implementation level.

7 Conclusions

In this paper we have shown how a model checker based on narrowing can be combined with a symbolic automaton learning algorithm such as CGE to give a new architecture for black-box specification-based testing using the learning-based testing (LBT) paradigm. We have benchmarked this LBT architecture against random testing, and shown that it compares favorably, with the potential for future improvement.

The results of Section 6.1 suggest that a pure narrowing procedure could be significantly improved by interleaving it with theorem proving techniques to detect unsatisfiability. This is because counterexamples to correctness may be sparse, in which case narrowing fails very slowly. Term rewriting could be applied to this problem too. Furthermore, it is known that basic narrowing modulo theories is incomplete and suggestions such as the *variant narrowing* of [7] could be considered. Finally, we observe that the CGE algorithm does not currently learn infinite state Mealy automata, and this is another extension of our work that must be considered for EMA.

Acknowledgement

We gratefully acknowledge financial support for this research from the Chinese Scholarship Council (CSC), the Swedish Research Council (VR) and the European Union under project HATS FP7-231620. We also acknowledge the helpful comments of the referees in producing the final version of this report.

References

1. F. Baader and W. Snyder. Unification theory. In *Handbook of Automated Reasoning*, pages 447–531. Elsevier, 2001.
2. P. Chauhan, E. M. Clarke, J. H. Kukula, S. Sapra, H. Veith, and D. Wang. Automated abstraction refinement for model checking large state spaces using sat based conflict analysis. In *Proc. 14th International Conference On Formal Methods in Computer-Aided Design (FMCAD02)*, 2002.
3. 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.
4. Hubert Comon. Disunification: a survey. In *Computational Logic: Essays in Honor of Alan Robinson*, pages 322–359. MIT Press, 1991.
5. L. de Moura and N. Bjorner. An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.
6. S. Escobar and J. Meseguer. Symbolic model checking of infinite-state systems using narrowing. In *Proc. Rewriting Techniques and Applications (RTA 2007)*, number 4533 in LNCS. Springer, 2007.

7. S. Escobar, J. Meseguer, and R. Sasse. Variant narrowing and equational unification. In *Proc. WRLA 2008*, number 238(3) in ENTCS. Springer, 2009.
8. G. Fraser, F. Wotawa, and P.E. Ammann. Testing with model checkers: A survey. Tech. rep. 2007-p2-04, TU Graz, 2007.
9. A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. *Logic Journal of the IGPL*, 14(5):729–744, 2006.
10. B. Guha and B. Mukherjee. Network security via reverse engineering of tcp code: vulnerability analysis and proposed solutions. *IEEE Network*, 11(4):40–49, 2007.
11. J.M. Hullot. Canonical forms and unification. In *Proc. Fifth Int. Conf. on Automated Deduction*, number 87 in LNCS, pages 122–128. Springer, 1980.
12. N. Markey. Temporal logic with past is exponentially more succinct. In *EATCS Bulletin 79*, pages 122–128, 2003.
13. 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.
14. 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.
15. 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.
16. K. Meinke and M. Sindhu. Incremental learning-based testing for reactive systems. In *Proc Fifth Int. Conf. on Tests and Proofs (TAP2011)*, number 6706 in LNCS, pages 134–151. Springer, 2011.
17. K. Meinke and J.V. Tucker. Universal algebra. In *Handbook of Logic in Computer Science: Volume 1*, pages 189–411. Oxford University Press, 1993.
18. A. Middeldorp and E. Hamoen. Completeness results for basic narrowing. *Applicable Algebra in Engineering, Communication and Computing*, 5(3–4):213–253, 1994.
19. 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.
20. 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.
21. 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.