Learning-based Software Testing using Symbolic Constraint Solving Methods

Fei Niu

School of Computer Science and Communication
KTH Royal Institute of Technology

Licentiate Thesis Presentation, 2011
Outline

1 Motivation and Background
   - Software Testing
   - Black-box Testing

2 Learning-based Testing Frameworks
   - Overview
   - Paper 1: LBT for Numerical Procedures
   - Paper 2: LBT for Reactive Systems

3 Results and Future Work
   - Empirical Performance Evaluation of LBT
   - Future Work
1 Motivation and Background
   - Software Testing
   - Black-box Testing

2 Learning-based Testing Frameworks
   - Overview
   - Paper 1: LBT for Numerical Procedures
   - Paper 2: LBT for Reactive Systems

3 Results and Future Work
   - Empirical Performance Evaluation of LBT
   - Future Work
Background

- Software systems
  - Ubiquitous
  - Increasingly complex and critical

- Software testing
  - One of the most important approaches to high-quality software
  - Remains time-consuming and ad-hoc

- High economic impact of software testing:
  - Cost up to 30-50% of a project budget
  - Achievable savings up to $22.2 billion in U.S (2002)
What is Software Testing

Software Testing In One Sentence

Software testing consists of the *dynamic* verification of the behaviour of a system under test (SUT) on a *finite* set of test cases, suitably *selected* from the usually infinite execution domain, against the *expected behaviour*.

- Test case generation (TCG)
  - Test objectives, selection strategies and adequacy criteria
- Test case execution
- Test case evaluation (the oracle)
- Automate them all!
Software Testing In One Sentence

Software testing consists of the *dynamic* verification of the behaviour of a system under test (SUT) on a *finite* set of test cases, suitably *selected* from the usually infinite execution domain, against the *expected* behaviour.

- Test case generation (TCG)
  - Test objectives, selection strategies and adequacy criteria
- Test case execution
- Test case evaluation (the oracle)
- Automate them all!
What is Software Testing

Software Testing In One Sentence

Software testing consists of the *dynamic* verification of the behaviour of a system under test (SUT) on a *finite* set of test cases, suitably *selected* from the usually infinite execution domain, against the *expected* behaviour.

- Test case generation (TCG)
  - Test objectives, selection strategies and adequacy criteria
- Test case execution
- Test case evaluation (the oracle)
- Automate them all!
Classes of Testing Methods

- Glass-box testing
  - Internal structure (source code)

- Black-box testing
  - Various specifications or models
  - More scalable and suitable in many cases
  - Functional testing (Input&Output)

- Non-functional testing
  - Quality of software such as scalability, reliability, security...
Classes of Testing Methods

- Glass-box testing
  - Internal structure (source code)

- Black-box testing
  - Various specifications or models
  - More scalable and suitable in many cases
  - Functional testing (Input & Output)

- Non-functional testing
  - Quality of software such as scalability, reliability, security...
Outline

1 Motivation and Background
   - Software Testing
   - Black-box Testing

2 Learning-based Testing Frameworks
   - Overview
   - Paper 1: LBT for Numerical Procedures
   - Paper 2: LBT for Reactive Systems

3 Results and Future Work
   - Empirical Performance Evaluation of LBT
   - Future Work
Black-box Testing Strategies

- Random testing (without a strategy)
- Boundary value analysis
- Equivalence partitioning
- Inductive testing
- Specification-based testing
- Model-based testing
  - e.g. all-verdict/path/state/transition coverage criteria
- Learning-based testing (LBT)
Characteristics of Learning-based Testing

- Functional specification-based black-box testing
- Test objective:
  - generates test cases that falsify given specification
- Make use of machine learning techniques
  - Model mining
- Make use of the great power of formal methods
  - Automated test case generation and evaluation
- Fully automated testing method
Outline

1 Motivation and Background
   - Software Testing
   - Black-box Testing

2 Learning-based Testing Frameworks
   - Overview
   - Paper 1: LBT for Numerical Procedures
   - Paper 2: LBT for Reactive Systems

3 Results and Future Work
   - Empirical Performance Evaluation of LBT
   - Future Work
Generic Framework of Learning-based Testing

- Iterative testing
- Inductive learning
- Constraint solving
Generic Framework of Learning-based Testing

- Iterative testing
- Inductive learning
- Constraint solving
Inductive Learning

- Inductive learning
  - Infer functions or languages from examples
  - e.g. automata learning (L*, RPNI, CGE, etc)

- Classes of (inductive) learning algorithms
  - **Active** v.s. **Passive**
  - **Incremental (Online)** v.s. **Offline**

- In LBT, we use **incremental learning**
  - Iterative
  - The sequence of hypothesis converges after $n$ iterations
    $$M_n \approx M_{n+1} \approx M_{n+2} \approx \ldots$$ (Learning in the limit)
Generic Framework of Learning-based Testing

- Iterative testing
- Inductive learning
- Constraint solving
Constraint Solving Techniques

- Satisfiability problem: is a formula $\phi$ true over a model $M$ for some assignment $\alpha$, denoted by

  \[ M, \alpha \models \varphi \]

- In LBT, we use constraint solving algorithms to compute $\alpha_t$ that violates the specification $\text{Req}$, denoted by

  \[ M_L, \alpha_t \models \neg \text{Req} \]

  where $M_L$ is the learned model and test cases can be derived from $\alpha_t$.

- Alternative test case generation method (random)
  - The first test iteration where the model is empty
  - No solution can be found by the constraint solver
Generic Framework of Learning-based Testing

- Iterative testing
- Inductive learning
- Constraint solving
- Instantiated with different type of SUTs
Procedural and Reactive Systems

- Procedural systems: Compute from an input to an output (if terminate)

\[ f : I \rightarrow O \]

- Reactive systems: Continuously interact with the environment

\[ F : (T \rightarrow I) \rightarrow (T \rightarrow O) \]
Publications


Outline

1 Motivation and Background
   - Software Testing
   - Black-box Testing

2 Learning-based Testing Frameworks
   - Overview
   - Paper 1: LBT for Numerical Procedures
   - Paper 2: LBT for Reactive Systems

3 Results and Future Work
   - Empirical Performance Evaluation of LBT
   - Future Work
Contributions

- Proposed the learning based testing framework
- An efficient modelling method for numerical procedures
- A test case generation algorithm using an existing quantifier elimination algorithm
- Experiments showing the efficiency of LBT compared with random testing
Specifying Numerical Procedures

- Requirements specification language
  - Hoare triple: \{Pre\} S \{Post\}
  - Pre and Post are (quantified) first-order logic formulas over real closed fields

Example

Computing square root by Newton’s method:

\{in \geq 0\} S \{|out * out - in| \leq \varepsilon\}
Modelling and Inferring Numerical Procedures (1)

Piecewise polynomial interpolation for $S(x)$ with $m$ tests

$$S(x) \approx f_1(x) \uplus f_2(x) \uplus \cdots \uplus f_m(x)$$

$$(d = 3, n = 1, pts = (d + 1)^n = 4)$$
Modelling and Inferring Numerical Procedures (2)

- A numerical procedure $S$ with $n$-dimension input and $m$ executed test cases
  
  $$S(\bar{x}) \approx y_m \equiv f_1(\bar{x}) \cup f_2(\bar{x}) \cup \cdots \cup f_m(\bar{x})$$

- On the next testing iteration with $m+1$ executed test cases
  
  $$S(\bar{x}) \approx y_{m+1} \equiv f_1(\bar{x}) \cup f_2'(\bar{x}) \cup \cdots \cup f_n(\bar{x}) \cup f_{m+1}(\bar{x})$$

- $S$ is efficiently and incrementally learned
  - Local updating
  - Learning in the limit (by Weierstrass Approximation Theorem)
  - Convergence by Monte Carlo method
Test Case Generation for Numerical Procedures

- Choose the best converged polynomial of center point \( \bar{c} = (c_0, c_1, \ldots, c_{n-1}) \) and radius \( r \)

\[
\mathbb{R} \models |\bar{x} - \bar{c}|^2 \leq r^2 \land Pre(\bar{x}) \land \neg Post(\bar{x}) \land f_{\bar{c}}(\bar{x}) = 0
\]

- Constraint solving algorithm for test case generation
  - Quantifier elimination method: Hoon-Collins cylindric algebraic decomposition (CAD)
  - Implemented in Mathematica
Outline

1 Motivation and Background
   - Software Testing
   - Black-box Testing

2 Learning-based Testing Frameworks
   - Overview
   - Paper 1: LBT for Numerical Procedures
   - Paper 2: LBT for Reactive Systems

3 Results and Future Work
   - Empirical Performance Evaluation of LBT
   - Future Work
Contributions

- A modelling method for reactive systems: extended Mealy automata over abstract data types (ADTs)
- A bounded model checking algorithm utilising narrowing techniques
- Experiments that show the efficiency of LBT compared with random testing.
Extended Mealy Automata over ADTs

- Extended Mealy automata over abstract data types (EMA)

Signature $\Sigma_A = (S, \Sigma, \text{input}, \text{output})$ of EMA $A$

- Initial state: $q_0^A \in A_{state}$
- State transition function: $\delta_A : A_{state} \times A_{input} \rightarrow A_{state}$
- Output function: $\lambda_A : A_{state} \times A_{input} \rightarrow A_{output}$

Functions to be inferred:

- General state transition function: $\delta_A^* : A_{state} \times A_{input}^* \rightarrow A_{state}$
- General output function: $\lambda_A^* : A_{state} \times A_{input}^+ \rightarrow A_{output}$
Abstract Data types - STACK

- Abstract data type \( \text{STACK} \) specified as \((\Sigma_{\text{stack}}, E_{\text{stack}})\)
  - Sorts: \(\{\text{Stack}, \text{Element}\}\)
  - Constant and functional symbols
    - \(e_0, e_1, \ldots, e_n, \text{error} : \rightarrow \text{Element}\)
    - \(\text{empty} : \rightarrow \text{Stack}\)
    - \(\text{Push} : \text{Element} \times \text{Stack} \rightarrow \text{Stack}\)
    - \(\text{Pop} : \text{Stack} \rightarrow \text{Stack}\)
    - \(\text{Top} : \text{Stack} \rightarrow \text{Element}\)
  - Axioms \(E_{\text{Stack}} (x \in X_{\text{Element}}, s \in X_{\text{Stack}})\)
    - \(\text{Pop}(\text{Push}(x, s)) = s\)
    - \(\text{Top}(\text{Push}(x, s)) = x\)
    - \(\text{Pop}(\text{empty}) = \text{empty}\)
    - \(\text{Top}(\text{empty}) = \text{error}\)
Term Rewriting

- A symbolic computational model with ADTs
- A rewriting rule is a directed equation: \( l \rightarrow r \)
- Term rewriting system (TRS) \( R \)
  - A set of rewriting rules
  - Rewriting relations (One step rewriting):
    \[
    t \rightarrow^R t'
    \]
    if \( \exists l \rightarrow r \in R, \sigma, p \in \tilde{O}(t) \) such that \( t|_p = \sigma l \) and \( t' \equiv t[\sigma r]|_p \)
  - Reflexive transitive closure (many step rewriting) \( t \rightarrow^{R^*} t' \)

Example

\[
\text{top(pop(push(e_1,push(e_0,s))))} \rightarrow^{R^*} e_0, \quad R \equiv E_{Stack}
\]
Term Rewriting

- A symbolic computational model with ADTs
- A rewriting rule is a directed equation: $l \rightarrow r$
- Term rewriting system (TRS) $R$
  - A set of rewriting rules
  - Rewriting relations (One step rewriting):
    $$ t \rightarrow^R t' $$
    if $\exists l \rightarrow r \in R, \sigma, \rho \in \tilde{O}(t)$ such that $t|\rho = \sigma l$ and $t' \equiv t[\sigma r]|\rho$
  - Reflexive transitive closure (many step rewriting) $t \rightarrow^{R^*} t'$

Example

$$ \text{top} \left( \text{pop} \left( \text{push} \left( e_1, \text{push} \left( e_0, s \right) \right) \right) \right) \rightarrow^{R^*} e_0, \ R \equiv \text{EStack} $$
Complete TRS

- Strongly normalising
  - there is no infinite sequence of rewrite steps
- Confluent (Church-Rosser property)
Inferring and Specifying EMAs

- CGE - an incremental learning algorithm for Mealy automata
  - Automata learning: infer equivalence classes of states
  - Builds a complete TRS $R$ of states (input strings)
  - More compact than tables

- Requirements specification language $LTL(\Sigma, X)$
  - Quantifier-free first-order linear temporal logic (FO-LTL) with equality and free variables $in \in X_{input}$ and $out \in X_{output}$
  - Includes past time temporal operators i.e. $F^{-1}, G^{-1}$

Example

$$G( (in = x) \land X((in = y) \implies X(out = x + y)))$$
Inferring and Specifying EMAs

- CGE - an incremental learning algorithm for Mealy automata
  - Automata learning: infer equivalence classes of states
  - Builds a complete TRS \( R \) of states (input strings)
  - More compact than tables

- Requirements specification language \( LTL(\Sigma, X) \)
  - Quantifier-free first-order linear temporal logic (FO-LTL) with equality and free variables \( in \in X_{\text{input}} \) and \( out \in X_{\text{output}} \)
  - Includes past time temporal operators i.e. \( F^{-1}, G^{-1} \)

Example:

\[
G((in = x) \land X((in = y) \implies X(out = x + y)))
\]
Given an EMA \( A \), a satisfaction relation for this LTL language is defined

\[
A, n, \bar{i}, \alpha \models \phi
\]

\[
\phi \equiv true \mid t = t' \mid \neg \phi \mid \phi \land \varphi \mid \cdots \mid F \phi \mid \cdots
\]

To generate test cases for \( A \) that is specified in \( Req \)

\[
A, 0, \bar{i}, \alpha \models \neg Req
\]

This is a model checking problem.
Model Checking EMA for TCG (2)

\[ \neg \text{Req} \rightarrow SatSet_0(\neg \text{Req}) \] that is inductively defined by

- \[ SatSet_n(in = t) \rightarrow \{ \{ \bar{x}_n = t \} \} \]
- \[ SatSet_n(out = t) \rightarrow \{ \{ \lambda (\delta^*(q^0, \bar{x}_0 \bar{x}_1 \ldots \bar{x}_{n-1}), \bar{x}_n) = t \} \} \]
- \[ SatSet_n(in \neq t) \rightarrow \{ \{ \bar{x}_n \neq t \} \} \]
- \[ SatSet_n(out \neq t) \rightarrow \{ \{ \lambda (\delta^*(q^0, \bar{x}_0 \bar{x}_1 \ldots \bar{x}_{n-1}), \bar{x}_n) \neq t \} \} \]

... 

- \[ SatSet_n(\phi \land \psi) \rightarrow \{ S_\phi \cup S_\psi \mid S_\phi \in SatSet_n(\phi), S_\psi \in SatSet_n(\psi) \} \]

... 

- \[ SatSet_n(X \phi) \rightarrow SatSet_{n+1}(\phi) \]
- \[ SatSet_n(F \phi) \rightarrow \bigcup_{k=0}^{\text{loopbound}} SatSet_{n+k}(\phi) \]

... 

\[ \neg \text{Req} \rightarrow S = \{ S_1, \ldots, S_m \} \text{ and } S_i \text{ is a disunification problem} \]

\[ \{ s_1 = ? t_1, s_2 \neq ? t_2, \ldots, s_n = ? t_n \} \]

modulo \( A = E \cup R \), which can be solved by narrowing
Model Checking EMA for TCG (2)

- \( \neg \text{Req} \rightarrow \text{SatSet}_0(\neg \text{Req}) \) that is inductively defined by
  - \( \text{SatSet}_n(\text{in} = t) \rightarrow \{\{\bar{x}_n = t\}\} \)
  - \( \text{SatSet}_n(\text{out} = t) \rightarrow \{\{\lambda(\delta^*(q^0, \bar{x}_0 \bar{x}_1 \ldots \bar{x}_{n-1}), \bar{x}_n) = t\}\} \)
  - \( \text{SatSet}_n(\text{in} \neq t) \rightarrow \{\{\bar{x}_n \neq t\}\} \)
  - \( \text{SatSet}_n(\text{out} \neq t) \rightarrow \{\{\lambda(\delta^*(q^0, \bar{x}_0 \bar{x}_1 \ldots \bar{x}_{n-1}), \bar{x}_n) \neq t\}\} \)
  - \( \ldots \)
  - \( \text{SatSet}_n(\phi \land \psi) \rightarrow \{S_\phi \cup S_\psi | S_\phi \in \text{SatSet}_n(\phi), S_\psi \in \text{SatSet}_n(\psi)\} \)
  - \( \ldots \)
  - \( \text{SatSet}_n(\text{X } \phi) \rightarrow \text{SatSet}_{n+1}(\phi) \)
  - \( \text{SatSet}_n(\text{F } \phi) \rightarrow \bigcup_{k=0}^{\text{loop bound}} \text{SatSet}_{n+k}(\phi) \)
  - \( \ldots \)

- \( \neg \text{Req} \rightarrow S = \{S_1, \ldots, S_m\} \) and \( S_i \) is a disunification problem

\[
\{s_1 = ? t_1, s_2 \neq ? t_2, \ldots, s_n = ? t_n\}
\]

modulo \( A = E \cup R \), which can be solved by narrowing
Disunification Problems

- Formalise an $E$-disunification problem

\[ S = \{ s_1 = ? t_1, s_2 \neq ? t_2, \ldots , s_n = ? t_n \} \]

- $R \equiv E$
- $\equiv_R$ reflexive symmetric and transitive closure of $\rightarrow^R$
- A $R$-unifier $\sigma$ such that
  - $\forall (s = ? t) \in S, \sigma s \equiv_R \sigma t$
  - $\forall (s \neq ? t) \in S, \sigma s \not\equiv_R \sigma t$

- Consider the case $R = \emptyset$
  - Find a unifier $\sigma$ such that $\sigma s_1 \equiv \sigma t_1 \land \sigma s_1 \not\equiv \sigma t_1 \land \cdots$
  - Syntactic unification by pattern matching
  - Decidable
Disunification Problems

- Formalise an $E$-disunification problem

$$S = \{s_1 =? t_1, s_2 \neq? t_2, \ldots, s_n =? t_n\}$$

- $R \equiv E$
- $=_{R}$ reflexive symmetric and transitive closure of $\rightarrow^{R}$
- A $R$-unifier $\sigma$ such that
  - $\forall (s =? t) \in S, \sigma s =_{R} \sigma t$
  - $\forall (s \neq? t) \in S, \sigma s \neq_{R} \sigma t$

- Consider the case $R = \emptyset$
  - Find a unifier $\sigma$ such that $\sigma s_1 \equiv \sigma t_1 \land \sigma s_1 \neq \sigma t_1 \land \cdots$
  - Syntactic unification by pattern matching
  - Decidable
Consider the case $R \neq \emptyset$: semidecidable

- Narrowing
  - Apply a (minimum) unifier on a term and the left hand side of a rule of $R$ to make it reducible by $R$

- Narrowing relation $\sim$ 

\[
t \sim [p,l \rightarrow r,\sigma] t'
\]

if $\exists p \in \bar{O}(t), l \rightarrow r \in R, \sigma$ such that $\sigma t|_p = \sigma l, t' = \sigma(t[r]_p)$

- Narrowing all terms in $S$ until a syntactic unifier can be found
Narrowing and Disunification

- Consider the case $R \neq \emptyset$: semidecidable
- Narrowing
  - Apply a (minimum) unifier on a term and the left hand side of a rule of $R$ to make it reducible by $R$
- Narrowing relation $\rightsquigarrow$

$$t \rightsquigarrow [p, l \rightarrow r, \sigma] t'$$

if $\exists p \in \bar{O}(t), l \rightarrow r \in R, \sigma$ such that $\sigma t|_p = \sigma l, t' = \sigma(t[r]_p)$
- Narrowing all terms in $S$ until a syntactic unifier can be found
Consider the TRS

\[ R = \begin{cases} 
0 + x & \rightarrow x \\
S(x) + y & \rightarrow S(x + y) 
\end{cases} \]

Step a: \( z + z \sim_{[r, \sigma]} S(x + S(x)) \)

- \( r : S(x) + y \rightarrow S(x + y) \)
- \( \sigma = \{ y, z \rightarrow S(x) \} \)

\[ z + z = ? S(S(0)) \]

- No solution

\[ 0 = ? S(S(0)) \]
- No solution

\[ S(x + S(x)) = ? S(S(0)) \]

\[ S(S(0)) = ? S(S(0)) \]
- True
- \( R \)-unifier: \( \{ z \rightarrow S(0) \} \)

\[ S(S(x' + S(S(x'))) = ? S(S(0)) \]
- No solutions
Consider the TRS

\[ R = \begin{cases} 
0 + x & \rightarrow x \\
S(x) + y & \rightarrow S(x + y) 
\end{cases} \]

Step a: \( z + z \sim_{[r,\sigma]} S(x + S(x)) \)

- \( r : S(x) + y \rightarrow S(x + y) \)
- \( \sigma = \{ y, z \rightarrow S(x) \} \)

\[ z + z =? S(S(0)) \]

\[ 0 =? S(S(0)) \]

No solution

\[ S(x+S(x)) =? S(S(0)) \]

\[ S(S(0)) =? S(S(0)) \]

True

\( R \)-unifier: \( \{ z \rightarrow S(0) \} \)

\[ S(S(x' + S(S(x'))) =? S(S(0)) \]

No solutions
Consider the TRS

\[ R = \begin{cases} 
0 + x & \rightarrow x \\
S(x) + y & \rightarrow S(x + y) 
\end{cases} \]

\[ \{ z \rightarrow S(0) \} \]

\[ S(0) + S(0) \rightarrow^R S(0 + S(0)) \rightarrow^R S(S(0)) \]

An example of Narrowing Search

\[ z + z = ? S(S(0)) \]

\[ 0 = ? S(S(0)) \]

No solution

\[ S(x + S(x)) = ? S(S(0)) \]

\[ S(S(0)) = ? S(S(0)) \]

True

\[ R \text{-unifier: } \{ z \rightarrow S(0) \} \]

\[ S(S(x' + S(S(x')))) = ? S(S(0)) \]

No solutions
Narrowing
- Solution complete
- Infeasibly large search space and frequently fails to terminate

Basic narrowing - a more efficient narrowing by Hullot
- Restrictions on allowable reducing positions
- Termination can be guaranteed

Model checking EMA by basic narrowing
- Bounded model checking
- Decision procedure for some complete TRSs
- Sound, complete and terminating
Outline

1. Motivation and Background
   - Software Testing
   - Black-box Testing

2. Learning-based Testing Frameworks
   - Overview
   - Paper 1: LBT for Numerical Procedures
   - Paper 2: LBT for Reactive Systems

3. Results and Future Work
   - Empirical Performance Evaluation of LBT
   - Future Work
Methodology of Performance Evaluation

- Iterative random testing (IRT) as the benchmark
- Benchmark what?
  - Fault detection power
    - is LBT faster than IRT at finding the *first* error?
  - NOT coverage measurement
- All faults are injected by mutating SUTs
- All results are averaged over a large number of SUT runs
Case Studies for Numerical Procedures (1)

- Bubble sort algorithm
- Random generated SUTs
- Each randomly generated SUT
  - A set of polynomials of random coefficients and degrees
  - Randomly generated specifications: equational & inequational
  - Faults of different sizes are injected in a controlled way
Benchmark Results

- Bubble sort algorithm
  - LBT is 10 times faster than IRT
- Random SUTs

![Performance ratio IRT/LBT graph]

- Equational spec.
- Inequational spec.
TCP/IP Stack Case Study for Reactive Systems

- 11 states, 12 input symbols, 6 output symbols
- 5 requirements
Benchmark Results for TCP/IP Model

- At the level of logical performance
  - LBT is always (5.5 to 400 times) faster than IRT

- At the level of real-time performance
  - LBT is often but not always significantly faster than IRT (107 times faster in the best case)
  - Much better performance can be easily obtained by better implementation
Outline

1 Motivation and Background
   - Software Testing
   - Black-box Testing

2 Learning-based Testing Frameworks
   - Overview
   - Paper 1: LBT for Numerical Procedures
   - Paper 2: LBT for Reactive Systems

3 Results and Future Work
   - Empirical Performance Evaluation of LBT
   - Future Work
Future Work

- More sophisticated learning algorithms
  - More efficient automata learning algorithms based on CGE
  - Learning algorithms for infinite-state EMAs
  - …

- More narrowing strategies
  - Combined with theorem proving techniques
  - Narrowing strategies to deal with more abstract data types

- More case studies