Covering both Stack and States while Testing Push-down Systems

P.-C. Héam
FEMTO-ST
Université de Franche Comté - CNRS - INRIA
16 route de Gray - 25030 Besançon, France

H. M’Hemdi
FEMTO-ST
Université de Franche Comté - CNRS - INRIA
16 route de Gray - 25030 Besançon, France
and
LIP2 Laboratory and INSAT,
University of Carthage, Tunisia

Abstract—In this paper we address the problem of generating abstract test cases from a system modelled by a push-down automaton. Existing classical coverage criteria are based on states, transitions or loops in the automaton. This paper is based on a known theoretical result claiming that the accessible stack configurations in a push-down automaton form a regular language. We propose a new coverage criteria based both on states and on the configurations of the stack. Experimental results on a model of the Shunting Yard Algorithm are also presented.

Keywords—Model based Testing, Push-down automaton, Coverage criterion

I. INTRODUCTION

The development of safe, secure and bug-free programs is one of the most difficult problems of computer science. Testing is an important activity during the development process to ensure system quality. There exist two classes of testing: (1) structural or "white-box" testing that is based on the analysis of the source code of the implementation, (2) functional or "black-box" testing that consists on comparing the system under test to a specification. Systems are commonly modelled by means of transition systems such as finite automata, etc. Model-based testing (MBT) [1] is a technique to validate software systems by generating finite size test cases automatically from models. The context of this paper is to generate test cases from a push-down automata that are automata equipped with a stack, and can be used for modelling recursive systems, parser and compiler or programm with a stack. Testing is often incomplete by nature. It cannot cover all possible system behaviors. Coverage criteria qualify the relation between test cases and model. There exist many tools for generating test cases from a finite state machine, for example SpecExplorer [4] and TGV [5]. Coverage criteria will be used to guide generation of new test cases for examples coverage states and transitions.

Test Generation from Push-down/Grammar Systems. Push-down (like) systems are frequently used in model based testing. Test generation from a grammar is frequently used for generating structured inputs for example in [6] for testing parser or refactoring engines [7]. A generic tool exploiting coverage criteria for generating test data from grammars has been proposed in [8]. In [9][10][11] several approaches for random testing from grammar specifications are proposed. A method of biased random grammar-based testing for covering all non-terminals symbols of a grammar is proposed in [12]. Random testing on push-down automata are investigated in [13] and [14].

Reachability in Push-down Automata. The reachability problem is the problem of deciding whether an automaton can reach a particular location from an initial location. This problem is decidable [15], [16]. Finkel et al. [15] propose a polynomial method for checking locations reachability in push-down automata.

B. Formal Background

If $X$ is a finite set, $X^+$ denotes respectively the set of finite words over $X$. The empty word (on every alphabet) is denoted $\varepsilon$. In this paper $\Sigma$ denotes a finite alphabet.

A finite automaton with $\varepsilon$ moves (or simply a finite automaton) is a tuple $(Q, \Sigma, \Delta, I, F)$ where $Q$ is a finite set of states, $\Sigma$ is a finite alphabet, $I \subseteq Q$ is the initial state, $F$ is...
In this example, \( \Delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q \) is the set of final states and \( \Delta \) is a set of transitions. A \textit{successful path} in a finite automaton is a (possibly empty) finite sequence of elements of \( Q \times \Sigma \times Q \) of the form \( (p_1, a, q_1) \ldots (p_n, a_n, q_n) \) such that \( p_1 \in I, q_n \in F \) and for each \( i, q_i = p_{i+1} \) and \( (p_i, a_i, q_i) \in \Delta \). The integer \( n \) is the length of the path and \( a_1 \ldots a_n \) is its label. The \textit{language accepted} by a finite automaton is the set of words which are the label of a successful path. Given two states \( p \) and \( q \) we write \( p \xrightarrow{a,q} \) if there exists a path in the automaton from \( p \) to \( q \) labeled by \( \varepsilon \). An example of finite automaton is depicted on Fig. 1. In this example, \( (q_0, \varepsilon, q_1) \) \( (q_1, Q, a, q_3) \) is a successful path: the word \( \varepsilon a = a \) is accepted. The accepted language is \( b^*a \).

A \textit{normalized push-down automaton} (NPDA for short) is a tuple \( A = (Q, \Sigma, \Gamma, \Delta, I, F) \) where \( Q \) is a finite set of \textit{states}, \( \Sigma \) and \( \Gamma \) are disjoint finite alphabets - \( \Sigma \) is the alphabet of the actions and \( \Gamma \) is the stack alphabet - \( I \subseteq Q \) is the \textit{initial state}, \( F \subseteq Q \) is the set of \textit{final states} and \( \Delta \) is a subset of \( Q \times \Sigma \times Q \cup Q \times (\Gamma \times \{+, -\}) \times Q \) is the set of \textit{transition}. A transition of the form \((p, a, q)\) with \( a \in \Sigma \) is called an \textit{action-transition}; a transition of the form \((p, X, +, q)\) (resp. \((p, X, -, q)\)) is called a \textit{push-transition} (resp. \textit{pop-transition}). A \textit{configuration} is a pair \((p, w)\) where \( p \in Q \) and \( w \in \Gamma^* \). An example of NPDA is depicted on Fig. 2.

In this example, \( Q = \{q_0, q_1, q_2, q_3\} \), \( \Sigma = \{a, b\} \), \( \Gamma = \{X, Y\} \), \( I = \{q_0\} \), \( F = \{q_3\} \). The action-transitions are \((q_0, b, p_0), (q_1, a, q_2) \) and \((p_3, a, q_3)\), the push-transitions are \((p_0, X, +, q_0)\) and \((q_0, Y, +, q_1)\), and the pop transitions are \((q_2, Y, -, q_3)\) and \((q_3, X, -, p_3)\).

An \textit{initial configuration} is a configuration of the form \((q_{\text{init}}, \varepsilon)\), with \( q_{\text{init}} \in I \). Let \((q, u)\) be a configuration and \( d \) be a transition. We denote by \((q, u) \cdot d\) the configuration \((p, v)\) such that either \( u = v \) or there exists an action transition of the form \((p, a, q)\), or \( v = uX \) and there exists a push transition of the form \((p, X, +, q)\), or \( vX = u \) and there exists a pop transition of the form \((p, X, -, q)\). Two configurations \( C_1 \) and \( C_2 \) are \textit{consecutive} if there exists a transition \( d \) such that \( C_2 = C_1 \cdot d \). In the NPDA of Fig. 2, there is a unique initial configuration: \((q_0, \varepsilon)\). One has for instance \((q_1, XY) \cdot (q_1, a, q_2) = (q_2, XY) \) and \((q_2, XY) \cdot (q_2, Y, -, q_3) = (q_3, X)\).

We inductively extend the notation \( \cdot \) to non empty finite sequences of transitions: \( C_2 = C_1 \cdot (d_1 d_2 \ldots d_k) = (C_1 d_1)(d_2 \ldots d_k) \). The notation implicitly implies that all the involved configurations exists. A \textit{path} in a NPDA from \( C_1 \) to \( C_2 \) is a finite sequence of transitions \( d_1 \ldots d_k \), such that \( C_1 \cdot d_1 \ldots d_k = C_2 \). It is \textit{successful} if \( C_1 \) is initial and \( C_2 \) is of the form \((p, \varepsilon)\) with \( p \in F \). In the NPDA of Fig. 2, the path \((q_0, Y, +, q_1)(q_1, a, q_2)(q_2, Y, -, q_3)\) is successful. A configuration \((p, w)\) is said \textit{accessible} if there exists a path from the initial configuration to \((p, w)\). It is said \textit{co-accessible} if there exists a path from \((p, w)\) to a configuration of the form \((q, \varepsilon)\), with \( q \in F \). A transition which is both accessible and co-accessible is said \textit{fair}. It corresponds to the configurations that are visited by successful paths.

The following result is proved in [15], [16] and is the theoretical base of our work.

**Theorem 1.** Let \( \mathcal{A} \) be a NPDA. For each state \( s \), one can compute in polynomial time a finite automaton \( \mathcal{A}_s \) on \( \Gamma \) accepting exactly the set of words \( v \) such that \((s, v)\) is an accessible configuration.

The automaton \( \mathcal{A}_s \) is computed using the Algorithm 1. Note that all the automata \( \mathcal{A}_s \)‘s are equal up to the final state: it suffices to run the algorithm only once to get all of them.

**Algorithm 1**

**Inputs:** a NPDA \( \mathcal{A} = (Q, \Sigma, \Gamma, \Delta, I, F) \) and \( s \in Q 

**Output:** \( \mathcal{A}_s \)

1: \( \Delta_0 := \emptyset \)
2: \( \Delta_1 := \{(p, X, q) \mid (p, X, +, q) \in \Delta \} \)
3: \( \Delta_1 := \Delta_1 \cup \{(p, \varepsilon, q) \mid (p, a, q) \in \Delta \} \)
4: while \( \Delta_0 \neq \Delta_1 \) do
5: \( \Delta_0 := \Delta_1 \)
6: for \((p, X, q) \in \Delta_1 \) do
7: \( (r, X, -, t) \in \Delta \) do
8: if \( q \xrightarrow{\varepsilon, \Delta_1} r \) then
9: \( \Delta_1 := \Delta_1 \cup \{(p, \varepsilon, t)\} \)
10: end if
11: end for
12: end for
13: end while
14: return \((Q, \Sigma, \Delta_1, q_{\text{init}}, \{s\})\)

Applying Algorithm 1 to the NPDA of Fig. 2, after the third line, one has the automaton depicted in Fig. 3.

One has \((q_0, Y, q_1) \in \Delta_1\), \((q_1, Y, -, q_3) \in \Delta \) and \( q_1 \xrightarrow{\varepsilon, \Delta_1} q_2 \), therefore, at Line 9, the transition \((q_0, \varepsilon, q_3)\) is added to \( \Delta_1 \). At the next loop, the transition \((p_0, \varepsilon, p_3)\) is added to \( \Delta_1 \). The automaton \( \mathcal{A}_{q_3} \) is depicted in Fig 4: the stack in \( q_3 \) is in \( X^* \). A similar computation will show that the stack in \( q_1 \) is in \( X^* Y \).
II. TESTING USING PUSH-DOWN AUTOMATA

The objective is to generate successful paths for a given NPDA according to a coverage criterion based on its fair configurations. For this purpose, it will be necessary to generate successful paths visiting a given configuration (Section II-A). The coverage criterion will be defined in Section II-B as well as the testing algorithm.

A. Building Successful Paths

The goal of this section is to show how to generate a successful path visiting a given fair configuration of an NPDA. This is done in two steps. The first one consists in describing how to generate a path from an initial state to a given accessible configuration of a NPDA. Next we explain how to generate a path from a co-accessible configuration to a final state.

Given a NPDA $A = (Q, \Sigma, \Gamma, \Delta, I, F)$ and an accessible configuration $(s, w)$, by Theorem 1, $w$ is accepted by $A$. First we build a partial function $E$ from $Q \times Q$ into $\{ \bot \} \cup \Delta^*$ such that $p \rightarrow_{s, A, q}^* q$ iff $E(p, q) \neq \bot$. Moreover, and for every $p, q$, if $E(p, q) \neq \bot$, then $(p, \varepsilon) \cdot E(p, q) = (q, \varepsilon)$.

Using Algorithm 2 on the example of Fig.2, we obtain four $\varepsilon$-transitions (see Fig.4). In this example, $E(q_1, q_2) = (q_1, a, q_2)$, $E(q_0, p_0) = (p_0, b, q_0)$, $E(q_3, p_3) = (q_3, a, p_3)$.

$$E(q_0, q_3) = (q_0, Y, +, q_1)E(q_1, q_2)(q_2, Y, -, q_3)$$

$$= (q_0, Y, +, q_1)(q_1, a, q_2)(q_2, Y, -, q_3)$$

and

$$E(p_0, p_3) = (p_0, X, +, q_0)E(q_0, q_3)(q_3, X, -, p_3)$$

$$= (p_0, X, +, q_0)(q_0, Y, +, q_1)(q_1, a, q_2)(q_2, Y, -, q_3).$$

Moreover, $E(q_0, p_3) = E(q_0, p_0)E(p_0, p_3)$ and $E(p_0, q_3) = E(p_0, p_3)E(p_3, q_3)$.

The following result can be easily checked.

**Proposition 2.** Let $w \in L(A_p)$ and $(p_1, a_1, p_2) \ldots (p_k, a_k, p_{k+1})$ be a successful path accepting $w$ (in particular $p_{k+1} = p$). For each $(p_i, a_i, p_{i+1})$ let $d_i = (p_i, a_i, +, p_{i+1})$ if $a_i \in \Gamma$ and $d_i = E(p_i, p_{i+1})$ if $a_i = \varepsilon$. One has $(p_1, \varepsilon) \cdot (d_1 \ldots d_k) = (p, w)$.

Proposition 2 allows the construction of a path in $A$ from an initial configuration to a given accessible configuration $(p, w)$. Consider for instance in $A_{q_3}$ the successful path $(q_0, \varepsilon, p_0)(p_0, X, q_0)(q_0, \varepsilon, q_3)$. One has $d_3 = E((q_0, p_0)) = (q_0, b, p_0)$, $d_2 = (p_0, X, +, q_0)$, $d_3 = E(q_0, q_3) = (q_0, Y, +, q_1)(q_1, a, q_2)(q_2, Y, -, q_3)$.

Now, from the automaton $A = (Q, \Sigma, \Gamma, \Delta, I, F)$, one can define for each transition $d \in \Delta$, the tuple $d^{R}$ as follows: if $d = (p, a, q)$, then $d^{R} = (q, a, p)$; if $d = (p, X, +, q)$, then $d^{R} = (q, X, -, p)$ and if $d = (p, X, -, q)$, then $d^{R} = ...$
Algorithm 3

**Inputs:** a NPDA $A = (Q, \Sigma, \Gamma, \Delta, I, F)$  
**Output:** A set of paths fulfilling the coverage criterion.

1. $C = Q \times Q$
2. Choose an arbitrarily successful path $\pi$
3. $S = \{\pi\}$
4. Remove from $C$ all the pairs visited by $\pi$.
5. **while** $C \neq \emptyset$ **do**
6. Choose arbitrarily $(p, q)$ in $C$
7. **if** $(p, q)$ cannot be visited **then**
8. $C = C \setminus \{(p, q)\}$
9. **else**
10. Let $\pi$ be a successful path visiting $(p, q)$
11. Remove from $C$ all the pairs visited by $\pi$.
12. $S = S \cup \{\pi\}$
13. **end if**
14. **end while**
15. **return** $S$

**A. Push-down Automata for Shunting Yard Algorithm**

A shunting yard algorithm\(^1\) is proposed by Dijkstra for converting mathematical expressions from the usual infix notation to the reverse Polish notation. For example, the expression $3 + 4 \times (2 - 1)$ becomes $3 4 2 1 - + \ast$ in the reverse Polish notation. A shunting yard algorithm is modelled by a NPDA in [14]. The tested C-implementation of the shunting yard algorithm is also given in [14] and comes from wikipedia. Figure 6 illustrates this NPDA that takes into account only the “+” and “*” operators. The stack labels are $\{Z, X_+, X_-, X_0\}$. The read transitions model what is read from the input, while the write transitions model what is written in the output. The label read $x$ or write $x$ mean the input or the output of a digit in $\{0, 1, \ldots, 9\}$. The label EOI denotes that there is nothing more to be read on the input.

**B. Oracle and Concretization**

After test generation, it remains to execute them on the implementation. If a NPDA are abstractions of systems as in our application, then, it may exist some test cases that do not correspond to any concrete execution of the system under test. The powerful advantage of our example is that all generated test cases are concretizable. The input and the output of the program can be computed from a given test case: the first step consists in extracting the labels of input and output transitions for each test case. Then, for each read $x$, write $x$, the value of $x$ is replaced by a digit in $\{0, \ldots, 9\}$. This value is randomly chosen.

**Example 4.**

Let $\pi = (q_{init}, \text{push}(Z), q_0)(q_0, \text{read } x, q_0, \text{write } x, q_0) (q_0, \text{read } *, q_*) (q_*, \text{pop}(Z), q_4) (q_4, \text{push}(Z), q\text{send}) (q\text{send}, \text{push}(X_0), q_0)(q_0, \text{read } x, q_0, \text{write } x, q_0) (q_0, \text{read } +, q_+) (q_+, \text{pop}(X_0), q_{+*}) (q_{+*}, \text{write } +, q_+)$

\(^1\)http://en.wikipedia.org/wiki/Shunting-yard_algorithm
(q+, pop(Z), q1)(q1, push(Z), q+end)(q+end, push(XX+), q0) (q0, read x, qd)(qd, write x, q0)(q0, EOI, q8) (q8, pop(X+), q0)(q6, write +, q8)(q8, pop(Z), qf) be a path, the sequence of the input and output transitions is read x write x read * read x write x read + write * read x write x write + by replacing the value of x randomly. Thus, the input mathematical expression is 6 * 3 + 5 and the output is 6 3 * 5 + .

For each test case, an input and an output are computed (in the automaton). The input is run on the implementation and the output (of the program) is compared to the output on the NPDA. If they equals, the test is successful, else, the implementation is not conform to the model.

A Java prototype has been implemented. Following the described approach we obtained 47 test cases that are all concretizable. It takes about 28.12 seconds on windows 7 64 bit, it covers 100% of the reachable transitions of the NPDA.

Secondly, we have generated 100 modifications (mutations) of the code, introducing bugs in order to evaluate whether their were detected by the test suit. These mutations have been generated using a freely available tool developed by Arun Babu and called Mutate2. Using the generated test suits, all the mutants have been detected.

IV. CONCLUSION

In this paper we proposed a new coverage criterion for testing systems modelled by push-down automata. The abstract test cases can be generated in polynomial time. Experimental results are encouraging. In the future, we plan to test the approach on larger systems or program, as parsing programs.

REFERENCES


2A copy of this code is available at http://members.femto-st.fr/pierre-cyrille-heim/mutatepy


