VeriTrace : Verifying k-Trace Equivalence

1 An Overview

VeriTrace is a tool to test k-trace equivalence between states in finite labeled transition systems (LTSs). Given an LTS (in the form of multiple transitions), the basic and primary functionalities of our tool are:

k-trace equivalence validation Decides whether two given states are k-trace equivalent for an arbitrary number $k(k \geq 1)$. In the case they are not, a counterexample trace can be produced to help the user to identify the causes.

Executions Assessment Evaluate the size (states number $\times$ transitions number) of an execution, the number of threads it contains, and how many methods are invoked by each thread.

sub-LTS extraction Given a LTS, extract a sub LTS from it.

Loop detection Whether a given LTS contains any loop, if it does, how many states are there leading to a loop, and which are they.

To implement these functionalities, we had to handle a lot of operations involving regular matching and text processing. Thus we chose to implement this tool with Perl 5 (version 5.10 or higher) programming language. Readers can find on Github the source codes of our tool.

1.1 1-trace equivalence testing

Deciding whether a pair of states in a finite LTS is 1-trace equivalent is relatively easier, we only need to investigate whether they possess the same set of traces (with τ-transitions omitted). Our tool achieves this by repeatly performing breadth first search (BFS) that returns a set of traces with length n for each state, and then comparing whether the two sets are identical.

If the two trace sets are different for a length $n \geq 1$, it means the two states are not 1-trace equivalent. Then our tool terminates and returns a counterexample trace from the difference set. In the otherwise case, the algorithm sets n to $n + 1$ and continues the previous procedure. We assume there is no circle [1] in the given LTS, the algorithm will

[1] A circle is an execution with a state to appear more than once

eventually terminate when n reaches the maximum length of all possible traces started from the two states. A `true` value is then returned to represent the 1-trace equivalence of the two states. Pseudocode in Algorithm 1 provides an illustration of the above testing procedure.

Input: a pair of τ-state $\langle s, t\rangle$
Result: whether s and t are 1-trace equivalent
$n = 0$;
N_{max} = the maximum length of traces started from s and t;
while $n < N_{max}$ **do**
  T(s) $\leftarrow \{\sigma : \sigma_0 = s \wedge |\sigma| = n\}$;
  T(t) $\leftarrow \{\sigma : \sigma_0 = t \wedge |\sigma| = n\}$;
  if *T(s)* $\neq$ *T(t)* **then**
    choose $\sigma \in \{T(s) \cup T(t)\} \backslash \{T(s) \cap T(t)\}$;
    return (σ and `false`) ;
  else
    $n = n + 1$ and continue the loop;
  end
end
return `true`;

Algorithm 1: Pseudocode for testing 1-trace equivalence. The τ-transitions are omitted in all traces.

1.2 k-trace equivalence

As m-trace inequivalence implies n-trace inequivalence for any $1 \leq m \leq n$. To test whether a pair of τ-states $\langle s, t\rangle$ is k-trace ($k > 1$) equivalent, we need to make sure that $\langle s, t\rangle$ are m-trace equivalent for all $m < k$. Since all k-traces of t are also k-traces of s, then our tool only checks whether there is a m-trace ($1 \leq m < k$) that belongs to s but not to t. This is done by conducting a m-trace equivalence check for $m = 1, 2, ..., k-1$. When the above procedure completes, we examine for each k-trace generated from s if it is possible for t to

produce the same k-trace. The following Algorithm shows the idea.

Input: a pair of τ-state $\langle s, t \rangle$, and an integer k
Result: whether s and t are k-trace equivalent
$m = 2$;
if *s and t are not 1-trace equivalent* **then**
  return `false`;
else
  while $m \leq k$ **do**
    $\Sigma(s) \leftarrow \{ \sigma : \sigma$ is a $m-1$ trace of $s\}$;
    $\Sigma(t) \leftarrow \{ \sigma : \sigma$ is a $m-1$ trace of $t\}$;
    if $\Sigma(s) \backslash \Sigma(t) \neq \emptyset$ **then**
      return `false` ;
    else
      $m = m + 1$ and continue;
    end
  end
end
return `true`;

Algorithm 2: Pseudocode for testing k-trace equivalence

2 Implementation & Usage

To use this tool, the user must provide at least one file that is readable by the standard IO handlers of the programming language. The input files, usually ending with `.aut, .dat`, or `.txt`, should contain all the transitions required to form a complete LTS (a snapshot of an LTS is shown in Fig 1), from which we first extract the transition graph. We use an hash table `trans` to store the transitions such that `trans[a][b]` returns the action from state `a` to `b`.

```
des (0, 72900, 18520)
(0, "CALL !CCAS !2", 18515)
(0, "CALL !SETFLAG !4", 18516)
(0, "CALL !SETFLAG !5", 18517)
(0, "CALL !CCAS !3", 18518)
(0, "CALL !CCAS !1", 18519)
(1, i, 6372)
(1, i, 6378)
...
```

Figure 1: A snapshot of an LTS. Each parenthesis represents a transition in which the numbers represent state labels and quoted contents and i represents actions.

2.1 Usage

Our tool can be used directly under Linux and macOS systems, where the Perl version is required to be at least `v5.10`. For Windows users, we recommend cygwin as a platform to provide Unix-like environment and command-like interface. The main executable file of our tool is `vtrace`, whose usage is:

./vtrace [args] [(files)] [(states)]

`args` stands for arguments (shown in Table 1), the choice would depend on the functionality a user requires (several examples will be given in later parts of this document).

Table 1: Arguments for vtrace tool

`-e`	`-equiv`	Default argument for trace equivalence testing.
`-a`	`-assess`	Assess input LTS.
`-h`	`-hide`	Hide actions that are not method calls or returns.
`-c`	`-circle`	Circle detection in a given LTS.
`-r`	`-restore`	Restore hidden actions in an quotient LTS. A class file and original LTS are required.

The `[(states)]` args are optional. For example, there is no need to specify it in the case of assessing an given LTS.

The main functionalities are wrapped into the script file: `ktrace_equiv.pl`. For users who want to extend the script and implement their own functionalities, we briefly introduce several subroutines that play important roles in that script. For better readability, the detailed codes are omitted in the following table, where `param` represents parameters of the subroutine, s, t represent states and d an integer representing depth.

```
my (
# A hash storing all the transitions of the given LTS
% trans,
# A hash recording the maximal k such that two states are k-trace equivalent
% equivalent,
# A hash storing prefix for pure traces
% prefexzero,
)
sub trans_process { ... }
# param: <.aut> file,
# return: a hash trans caching the transition graph specified by the input
    file
sub prefix_trace { ... }
# param: <s, d>
# return: all 1-traces started from s with lengthes no more than d
```

```
sub one_trace_equiv { ... }
# param: a pair of states <s, t>
# return: true if s and t are 1-trace equivalent, false otherwise
sub prefix_k_trace { ... }
# param: <s, d>
# return: all k-traces started from s with lengthes no more than d
sub k_trace_equiv { ... }
# param: a pair of states <s, t> and an integer k
# return: true if s and t are k-trace equivalent, false otherwise
```

3 A Few Examples

3.1 Example 1: Assessing trace equivalence of any τ-pairs

By τ-pair we mean a pair of states $\langle s, t\rangle$ whose transition contains an action that is neither a method call nor a method return. In the following figure, the input file `input/lfs23.aut` is a quotient LTS extracted from an execution of `lock free stack`. The number 2 means there are two processors, while 3 represents each processor randomly invokes methods (push or pop) for three times.

The last line (`(1, 2, 3, 4)_equiv = ...`) concludes the results, showing that there are ten 1-trace equivalent τ-pairs and two 2-trace equivalent τ-pairs. There is no τ-pair that is k-trace equivalent for any $k \geq 3$.

```
$ ./vtrace -equiv input/lfs23.aut
1-trace equivalent pairs:
                1310        231
                1332        249
                1587        508
                1602        537
                1919        1193
                1920        1240
                ...

2-trace equivalent pairs:
                1961        1959
                ...

(1, 2, 3, 4)_equiv = (10, 2, 0, 0)
```

Figure 2: Assessing trace equivalence

3.2 Example 2: Print Counterexample Trace

Given a τ-pair, our tool can detect the maximal value k such that the two states are k-trace equivalent. Fig 3 considers two states: $\langle s_8, s_{517} \rangle$, a τ-pair in a quotient LTS extracted from an execution of MS queue. This τ-pair is not 1-trace equivalent. The 4-th line:

(8) `RET !DEQ_E !2 (..)`

presents a trace that belongs to state s_8 but not to s_{517}. The complete LTS and path that leads to the trace is printed out to help the users to identify the cause.

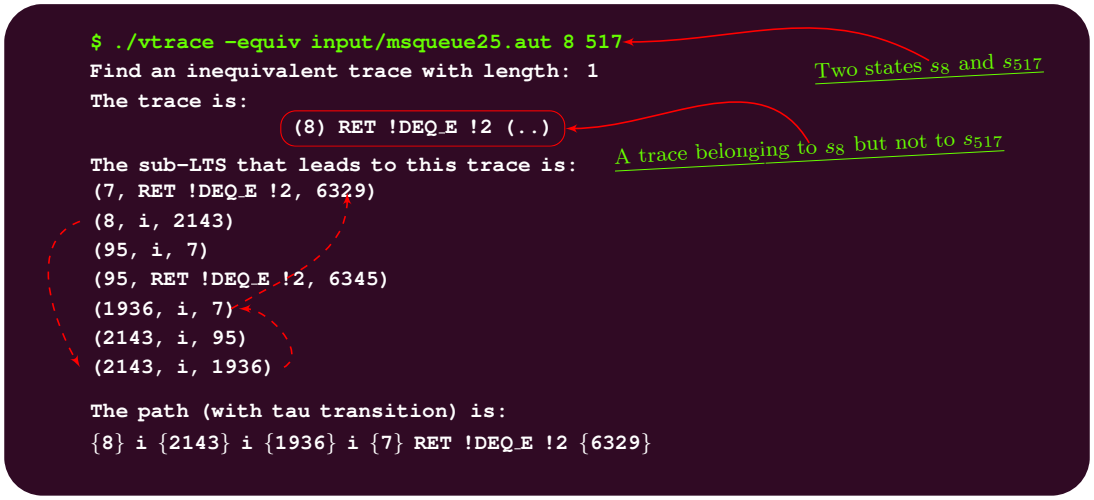

Figure 3: An example of msqueue

3.3 Example 3: Assessing LTS

The input file `lfs23.aut` actually corresponds to an LTS. By running `./vtrace -a [file]`, the user can obtain information on the number of threads, states and transitions. The pair $(129 : 17)$ in the last line means the final state is s_{129}, and it occurs for 16 times in the aut file.

```
$ ./vtrace -a input/lfs23.aut
The number of threads: 2
Invoked methods by each threads: 3

There are 1963 states, 4380 transitions
The number of final state(s): 1
The final state(s): <129: 16>
```

3.4 Example 4: Detecting Loops

```
$ ./vtrace -c input/hwbig.aut
9 leads to a circle
4 leads to a circle
72 leads to a circle
...
48 circles were found
```

There are 48 loops detected in the given LTS. The states from which a loop can be found are: $s_9, s_4,$ and $s_{72}, \cdots$.