From Symbolic Execution to Concolic Testing

Daniel Paqué
TU Kaiserslautern
d_paque12@cs.uni-kl.de

ABSTRACT. Creating software tests with high coverage can be quite challenging. One technique to automatically do so is symbolic execution. At first I will summarize and explain the procedure of symbolic execution. As pure symbolic execution has some serious weaknesses in runtime I then will present concolic testing and execution generated testing. Both are modern approaches to solve the runtime problem by mixing symbolic execution with concrete execution. Finally I will explain how concolic testing deals with the challenge of concurrent programs. There I will especially point out how the necessary race conditions are found with the aid of vector clocks.

1 Introduction

The development of software with a certain quality forces software developers to spend a serious amount of their resources for testing. Since today's software consists of more and more lines of code the effort for testing constitutes quite a big part in the budget of software development*. To save money in testing and still keep quality high the idea of code coverage was introduced very soon. The idea of code coverage is that every line of code is executed in at least one test case. Surely this does not guarantee correctness, but having a line of code or a branch in the program, which is not tested at all involves an even greater risk.

Since programs can have a complex structure with a lot of branches it is not trivial to generate a small number of test cases with high code coverage. One great way to do so is symbolic execution which I explain in the next chapter. In chapter three I then present two new approaches extending the idea of symbolic execution. Chapter four deals with one of the main challenges of these approaches, concurrency, and chapter five closes with related work.

2 Symbolic Execution

As the name already implies, the idea of symbolic execution is to execute the program in symbolic domain. Thereby the input of the program is set on a symbolic value which can be everything. This allows the symbolic execution to explore all possible paths through the program. For each path the constraints of the branching points are collected. With these constraints a concrete input can be generate for each path, providing the possibility to build a test suite with high coverage.

Two inputs create different paths if there is at least one branch in the program where the evaluation of this branch condition produces distinct outcomes for both inputs.

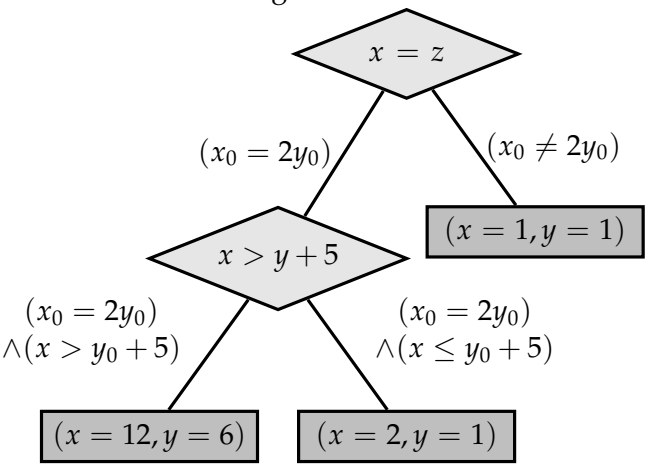
*It is not unusual that this amount reaches up to 50% of the total costs of the software. [12]

Figure 1:

```
1 foo(int x,int y){
2     z = 2*y;
3     if (x == z){
4         if (x > y + 5){
5             //some error
6         }
7     }
8 }
```

Description of Figure 2: *Possible execution paths of Figure 1 with path condition and satisfying assignments*

Figure 2:



For example the function $foo()$ in Figure 1 has three different execution paths generated by the inputs $\{x = 1, y = 1\}, \{x = 2, y = 1\}$ and $\{x = 12, y = 6\}$.

For implementing symbolic execution a symbolic domain is needed. This domain consists of a symbolic state σ which maps variables to symbolic expressions, and a path condition PC for each path. Initially σ is an empty map and the symbolic execution maintains only one PC which is set on true.

If the tested function needs some initial input for a variable, e.g. when the program receives a system input, the symbolic execution adds the mapping $var \mapsto s$ to σ, where s is a fresh symbolic variable. Testing $foo()$ in Figure 1, σ will be set to $\sigma = \{x \mapsto x_0, y \mapsto y_0\}$ since x and y are needed as input.

Symbolic execution interprets each instruction of the program in the symbolic domain. For instance if the next line is an assignment $v = e$, it has to evaluate the expression e symbolically and then store the result $\sigma(e)$ in the symbolic state σ by mapping v to $\sigma(e)$.The evaluation of the expression in line 2 in the example results in $z = 2y_0$, which will be stored in σ.

When reaching a conditional statement `if (e) then P1 else P2` they symbolic execution forks by creating an instance for each branch. In detail it does the following: Evaluate `(e)` under σ, update PC to $PC \wedge \sigma(e)$ for the then-branch and create another path condition $PC' = PC \wedge \neg\sigma(e)$ for the else-branch. Since it is possible that there is an infeasible path the symbolic execution has to check PC and PC' for satisfiability. If PC is satisfiable the symbolic execution continues with P_1. Respectively if P_2 is satisfiable it creates a new instance and continues with P_2. If any condition is unsatisfiable this instance terminates.

Additionally an instance of the symbolic execution terminates if it hits an exit statement or an error, or if the program violates some assertions or crashes. Since these are the cases we are interested in, symbolic execution uses the related path constraint to generate a satisfying assignment and returns it. As this assignment fulfills all constraints which led to this point it will always trigger the error.

Applied to Figure 1 symbolic execution creates three instances with $(x_0 \neq 2y_0)$,

$(x_0 = 2y_0 \land x_0 \leq y_0 + 5)$ and $(x_0 = 2y_0 \land x_0 > y_0 + 5)$ as path conditions (see also Fig. 2). According test inputs are for example $\{x = 1, y = 1\}, \{x = 2, y = 1\}$ and $\{x = 12, y = 6\}$.

Symbolic execution is able to give us high-coverage test suites with concrete inputs. However this is only possible if the resulting path constraints are solvable (within a reasonable amount of time). Formulas that cannot be solved efficiently, like non-linear constraints make proving satisfiability very difficult. The same holds for calls to source code which is not available (e.g. operating system procedures). Regarding the example imagine that we replace line 2 with a call to a function $bar()$ with unknown source code (see Figure 3). As a consequence the constraints in line 3 $x_0 = bar(y_0)$ cannot be solved and the symbolic execution fails to generate any input for this program.

```
1   foo2(int x,int y){
2       z = bar(y)
3       if (x == z){
4           ...
5       }
6   }
7
8   bar(int w){
9       return 2*w;
10  }
```

Figure 3: *Modified version of foo()*

```
1   foo3(int x){
2     y = 2;
3     z = 3*y;
4     if (x == z){
5         //...
6     } else {
7         //...
8     }
9   }
```

Figure 4: *EGT version of foo(). Line two and three do not need symbolic execution*

3 Modern Approaches

Since symbolic execution cannot handle unsolvable or almost unsolvable constraints on its own an idea of modern approaches is to mix concrete execution with symbolic execution.

3.1 Concolic Testing

The key idea of concolic testing is to use some given or random input, execute the program with it and collect the constraints of the branches which have been taken during its execution. Now a constraint solver can infer a new input for the next execution that forces the program to take an unexplored path. Hence concolic testing maintains besides the symbolic state another concrete state where the mapping of variables to their concrete values is saved. This state is set at the beginning of a concolic testing, usually with some random values.
For the example in Figure 1 we generate the random input $\{x = 29, y = 4\}$. The symbolic execution is now done simultaneously with the concrete one. Based on the generated input the concrete execution takes the else-branch in line 3 and the symbolic execution returns $x_0 \neq 2y_0$ as constraint. Solving the negation of this constraint leads to the next execution with $\{x = 8, y = 4\}$ as input. This time the execution continues with the then-branch and takes the second else-branch (l.8). The resulting constraint is $(x_0 = 2y_0) \land (x_0 \leq y_0 + 5)$.

Consecutively we get $\{x = 10, y = 5\}$ as third and last input. This input leads to the error-statement. Since this execution did not reveal any further constraints and all possible branches have been taken, the concolic testing reports that all feasible paths have been taken and terminates.

At a first glance there is no big difference to pure symbolic execution apart from the fact that it uses a sequential depth search to explore the paths. Only when the symbolic execution cannot solve the constraint (efficiently) the benefit of the concrete execution unveils. Figure 3 shows a modified version of $foo()$ where the assignment on z is replaced with a call to function $bar()$. Imagine now that the source code of $bar()$ is not available. This results in the equation $x_0 = bar(y_0)$ for the first branching point, which is unsolvable since *bar()* is an uninterpreted function. Consequently a pure symbolic approach could neither continue its execution as it does not know which path to take, nor could it generate a concrete input for this path. Though the concolic execution uses the concrete values to evaluate the condition in the concrete domain. Thereby it at least finds out which of the condition and its negation is satisfiable since one of them has to evaluate to true. The execution then continues with the corresponding path. Applied to Figure 3 with $\{x = 1, y = 1\}$ as input, the execution evaluates $(x_0 == bar(y_0))$ with the concrete values. As this returns false, the execution continues with the else branch without reasoning about the symbolical constraint.

Note that this did not solve the constraint, so the concolic execution is not able to infer any input for the alternative branch. However there are cases where concolic execution can solve the constraint by replacing some of the symbolic values with their concrete values. In the example it is enough to replace $bar(y_0)$ with the return value of the concrete execution to get $x_0 \neq 2$. With this simplification the input $\{x = 2, y = 1\}$ follows immediately. Nevertheless one has to be aware that the concolic execution cannot guarantee completeness since it is still possible that it generates unsolvable constraints which cannot be simplified.

3.2 Execution-Generated Testing

Like Concolic Testing the EGT (execution-generated testing) approach is based on the idea to mix the symbolic execution with the concrete execution of a program. Whereas concolic testing is guided by the concrete execution, EGT usually works on the symbolic domain. Only in some cases, where the symbolic execution gets stuck or when all operands of the instruction are concrete it switches to concrete execution. As a consequence the concrete values are not initialized immediately but generated when needed. In contrast to concolic testing the EGT approach does not explore the paths sequentially. When hitting a branch EGT forks the execution and creates a new instance, similar to the pure symbolic execution.

Applying the EGT approach to Figure 1 the execution will only start with symbolic values x_0 and y_0. As this example does not contain any special cases the execution of the EGT approach works like the pure symbolic execution [see section 2]. Again the execution ends with three constraints for the three paths which are solved to get the input values.

To illustrate the differences consider $foo3$ in Figure 4. EGT dynamically checks if all operands of the instruction are concrete before every instruction. If so the operation can be executed concretely. Respectively a condition will be evaluated concretely if all operands are concrete. Accordingly line 2 and 3 are both executed concretely. However as soon as there is

at least one symbolic operand, EGT will execute the instruction symbolically. In the example this happens at the condition `if (x == z)` in line 4 since it contains the symbolic value x_0. With the knowledge $z = 6$ from the concrete execution EGT can simplify the condition to $(x = 6)$. Similar to concolic testing the constraint of this condition and its negation are checked for satisfiability. Since both branches have satisfying assignments ($x = 6$ for the then branch and $x = 5$ for the else branch) the execution forks and each instance continues as usual.

3.3 Properties of Concolic and Execution Generated Testing

Since both approaches work with the same principles the following reasoning holds for both approaches: The big advantage of mixing concrete with symbolic execution is that by executing a single path the tool tests all possible values of this path, rather than a single set of concrete values [3, p.2]. So it is not necessary to hit the exact values that trigger the error but it is sufficient to find the path it lies on.

Moreover it holds that any detected execution leading to an error is guaranteed to be sound, meaning the error is not a false alarm. Even when hitting unsolvable constraints, switching to concrete execution ensures that these approaches explore only feasible paths [6, p.8]. Consequently for each explored path a test case can be generated which will always take this path and trigger detected errors.

Another big advantage of these approaches is that they can be done automatically. Tools implementing this technique, like CUTE [5] or KLEE [2] do not need any input besides the tested code [2, p.2], [5, p.1].

Finally, unlike other program analysis techniques, approaches with symbolic execution are not limited to finding generic errors such as buffer overflows [1, p.1]: As the path condition describes the possible range for each variable in each position of the program, these approaches can reason about complex program assertions and are only limited by the capabilities of the constraint solver.

4 Concurrency in Concolic Testing

Today's real-world software is seldom totally sequential. Accordingly one big challenge of concolic testing is to adopt its method for testing concurrent programs. These programs consist of several threads or actors whose execution can be interleaved arbitrarily. Consequently the number of possible execution paths, which is already huge in sequential environment, is now exploding due to the number of possible interleavings of the concurrent events.

To cope with this challenge Koushik Sen and Gul Agha developed a new technique to adapt concolic testing to concurrent programs. They called their technique "race detection and flipping algorithm" [6] - [9] and its main contribution is to minimize redundant executions in concurrent programs.

The basic idea of this reduction is that a lot of these interleavings share the same structure and thus are equivalent with respect to finding a bug. In order to detect such equivalent interleavings the algorithm computes the race condition between certain events addition-

Thread t_0

```
1    x = 3;
```

Thread t_1

```
1    y = 0;
2    x = 4;
3    z = x + 12;
```

Figure 5: *Simple multi-threaded code.*

Def.: *Following I represent events in the execution of a multi-threaded program as tuple* (t, l)*, where t is the thread of the executed instruction on label l.*

Figure 6: *Some possible executions of Fig.5.*
Ex.1: $(t_0, l.1), (t_1, l.1), (t_1, l.2), (t_1, l.3)$
Ex.2: $(t_1, l.1), (t_0, l.1), (t_1, l.2), (t_1, l.3)$
Ex.3: $(t_1, l.1), (t_1, l.2), (t_0, l.1), (t_1, l.3)$

ally to the path constraints. Similar to the path constraint the race conditions are collected during a concrete execution. The input for the next execution is then computed not only by the path condition but also in consideration of the race conditions. Thereby execution paths with a different interleaving but the same structure of races are avoided. Techniques for avoiding such redundant executions are called *partial order reduction* [10].

A race occurs between two events if they stem from different threads, if both access the same memory location without holding a lock and if the order of the happening of these events can be permuted by changing the schedule of the threads. Figure 5 shows a code example with race condition (the accesses on the shared variable x). Figure 6 illustrates that the detection of the error in Figure 5 does not depend on the exact interleaving but on the order of the instruction with race condition. It shows three possible executions of the code, generated by different interleavings of thread 0 and thread 1. However the order of the accesses on x and therefore the structure of the race conditions in Ex.1 and Ex.2 is the same. Thus the algorithm should only consider one of these executions and then try to generate a distinct structure of the race condition for the next execution (e.g. Ex.3).

4.1 Execution Model

Before going into detail it has to be mentioned that Sen and Agha constructed their execution model using some assumptions. Important assumptions are:

- the execution of a statement by a thread can perform at most one shared-memory operation [9, p.5],
- the set of memory locations, needed to implement locking (like semaphores) is disjoint from the set of memory locations that can be read or written [9, p.6], and
- sequential consistent memory model [†]is used. [9, p.6]

With the second assumption lock and unlock do not need an extra treatment although they technically manipulate memory locations. This keeps the algorithm simple.

In the following section I sum up some of the important aspects of the execution model from [6] and [9], necessary for the "flipping and race detection" algorithm: The execution of a program P is considered as a sequence of events where an event is the execution of a statement from P. An event is represented as a 3-tuple (t, l, a), where l is the label of the

[†]In a s.c. memory model write operation become instantly visible and are not delayed due to buffering.

statement executed by thread t and a is the type of the shared memory access in the statement (w for write, r for read, l for lock and u for unlock). Such a sequence of events is called execution path. The set of all feasible execution paths exhibited by the program P on all possible inputs and all possible choices by the scheduler is called $Ex(P)$.

As race detection is the basis of identifying equivalent paths, it has to be defined exactly. In order to do so Sen and Agha introduced in [9] four relations: sequential, shared-memory access precedence, causal and race relation. The first two relations constitute the order of events and memory accesses. The causal relation is the union of the first two relations and is needed to define the race relation. In detail the relations are defined as follows [9, p.7-8]:

In an execution path $\tau \in Ex(P)$, any two events $e = (t_i, l_i, a_i)$ and $e' = (t_j, l_j, a_j)$ appearing in τ are *sequentially related* (denoted by $e \lhd e'$) iff:

1. $e = e'$, or
2. $t_i = t_j$ and e appears before e' in τ, or
3. $t_i \neq t_j, t_i$ created the thread t_j, and e appears before e'' in τ, where e'' is the fork event on t_i creating the thread t_j, or
4. there exists an event e'' in such that $e \lhd e''$ and $e'' \lhd e'$

We say $e \Updownarrow e'$ iff $e \not\lhd e'$ and $e' \not\lhd e$.

In an execution path $\tau \in Ex(P)$, any two events $e = (t_i, l_i, a_i)$ and $e' = (t_j, l_j, a_j)$ appearing in τ are *shared-memory access precedence related* (denoted by $e <_m e'$) iff:

1. e appears before e' in τ, and
2. e and e' both access the same memory location m, and
3. one of them is an update (not a read) of m.

In an execution path $\tau \in Ex(P)$, any two events $e = (t_i, l_i, a_i)$ and $e' = (t_j, l_j, a_j)$ appearing in τ are *causally related* (denoted by $e \preccurlyeq e'$) iff:

1. $e \lhd e'$, or
2. $e <_m e'$ for some shared-memory location m, or
3. there exists an event e'' in such that $e \preccurlyeq e''$ and $e'' \preccurlyeq e'$

The causal relation is a partial-order relation. We say that $e \parallel e'$ iff $e \not\preccurlyeq e'$ and $e' \not\preccurlyeq e$.

Any two events $e = (t_i, l_i, a_i)$ and $e' = (t_j, l_j, a_j)$ are *race related* (denoted by $e \lessdot e'$) iff:

1. $e \Updownarrow e'$, and
2. if e is lock event and e'' is the corresponding unlock event, then $e'' <_m e'$ and there exists no e_1 such that $e_1 \neq e'', e_1 \neq e'$, $e'' \preccurlyeq e_1$ and $e_1 \preccurlyeq e'$, and
3. if e is read or write event, then $e <_m e'$ and there exists no e_1 such that $e_1 \neq e$, $e_1 \neq e', e \preccurlyeq e_1$ and $e_1 \preccurlyeq e'$

Roughly said, if two accesses on the same memory location stem from different threads, if they are not protected by a lock and if there is no other such event between those two, they are in a race condition. Based on this definition the concolic testing is extended by the "race-detection and flipping" algorithm.

4.2 The race-detection and flipping algorithm

At first the algorithm generates an initial schedule specifying the interleaving of the threads and sets all input variables on random concrete values. Starting with this input the algorithm does the following in a loop: Execute the given code with the generated input and schedule. Collect and extend the path condition at each branching point (like concolic testing) and compute the race condition between the events. Dependent on the path condition and the computed races the algorithm generates a new input or a new schedule and executes the program again. This is repeated until all possible, distinct execution paths have

Figure 7:

Thread t_0

```
1    x = 3;
```

Thread t_1 (with z as input)

```
1    x = 2;
2    if (x == 2*z+1)
3        error;
4    ...
```

Figure 8:

Ex.1: $[z = 8,\ sched_0]$
$(t_0, l.1), (t_1, l.1), (t_1, l.2), (t_1, l.4)$

Ex.2: $[z = 8,\ sched_1]$
$(t_1, l.1), (t_1, l.2), (t_1, l.4), (t_0, l.1)$

Ex.3: $[z = 8,\ sched_2]$
$(t_1, l.1), (t_0, l.1), (t_1, l.2), (t_1, l.4)$

Ex.4: $[z = 1,\ sched_2]$
$(t_1, l.1), (t_0, l.1), (t_1, l.2), (t_1, l.3), (t_1, l.4)$

been explored.

Since it is possible that several paths belong to the same schedule the algorithm first tries to generate a new input for the same schedule. This is done like in concolic testing by picking a constraint, negating it and finding some satisfying values. Only when all distinct paths within that schedule have been explored a new schedule is generated. To do so the algorithm picks two events e and e' which are in a race condition. Let e be the first of those events and let t be the related thread. The beginning of the new schedule is a copy of the former schedule, up to the point where e has happened. There t is delayed as much as possible. Since $e \Updownarrow e'$ holds for a race condition this will definitively permute e and e'. The new schedule is used for the next execution.

Example

Figure 7 contains a short multi-threaded program with a possible race condition. The program consists of two threads t_0 and t_1, a shared integer-variable x and a local integer variable z, which is set at the beginning of the program as input. The algorithm first generates a random input for z and executes P with a default schedule. Assume that $\{z = 8\}$ is picked as input value, and that t_0 is executed first. The resulting execution has $(2 \neq 2z_0 + 1)$ as path condition and can be seen in Figure 8 - Ex.1. Besides the algorithm detects a race condition between the first and the second event because both of them write the same variable x in different threads without having a lock.

As described the algorithm now tries to generate a new input which would follow another path without changing the schedule. Therefore the only condition $(2 \neq 2z_0 + 1)$ is picked, negated and a constraint solver tries to find a satisfying assignment. However $(2 = 2z_0 + 1)$ has no satisfying solution (as z is an integer). Hence the routine for generating a new schedule starts. In the new schedule $(t_0, l.1)$ will be postponed as much as possible, resulting in execution Ex.2 (Figure 8).

Running the second schedule with the old input, $(2 \neq 2z_0 + 1)$ is collected again as path condition. In this execution the second event hinders the first and the third event to be in a race since it holds that $e_1 \preccurlyeq e_2 \wedge e_2 \preccurlyeq e_3$ (see section 4.1). Therefore the algorithm computes only one race condition between $(t_1, l.2)$ and $(t_0, l.1)$. As the negated path constraint still cannot be solved a third schedule is generated. This time the second event is delayed which

results in execution Ex.3.

Also in the third run the else-branch is taken. However due to the changed interleaving the path condition becomes $3 \neq 2z_0 + 1$. This time the negation of the constraint is solvable and $z = 1$ is obtained as new input for the fourth execution. Consequently the program is executed with $z = 1$ and the old schedule, which results in finding the error.

Execution 4 again contains several race conditions. Though re-ordering them would generate the same order as in the first or in the third run. This means all possibilities to generate distinct paths have been taken. Consequently the algorithm stops since now all different execution paths have been explored.

Vector Clocks

In [6] and [9] Sen and Agha present a tool called jCute implementing the "race-detection and flipping" algorithm. In order to compute the race conditions jCute uses vector clocks. Vector clocks are basically integer vectors which assign to each thread a time stamp, representing the last known action of this thread. With the aid of such vector clocks the relation of the events can be tracked. More precisely a vector clock V is a mapping $V : T \to \mathbb{N}$, where T is the set of threads created by the program. Vector clocks (VC) can be compared: $V \leq V'$ iff $V(t) \leq V'(t)$ for all $t \in T$. $V \neq V'$ iff $V \not\leq V'$ and $V' \not\leq V$. Besides $max\{V, V'\}$ is defined as a new VC with the component-by-component maximum of both vector clocks.

To detect a race condition all three conditions of the race relation (section 4.1) have to be checked. The first condition is checked by building a **sequential vector clock** SVC. Since in concurrent systems actors can have different knowledge each thread t is assigned its own sequential vector clock, denoted by VS_t. All vector clocks are initialized with 0 and will be updated at runtime. Whenever an event e of a thread t happens the following is done[‡]:

1. If e is not a fork event or a new thread event, then $VS_t(t) = VS_t(t) + 1$
2. If e is a fork event and if t' is the newly created thread then
   $VS_{t'} = VS_t,\ VS_t(t) = VS_t(t) + 1$ and $VS_{t'}(t) = VS_{t'}(t) + 1$

A **dynamic vector clock** DVC is build to check the second and the third condition of the race relation. Again each threads gets their own vector clock. Besides two additional vector clocks, V_m^a and V_m^w are associated with each shared memory location m, V_m^a for accesses on m, V_m^w for write, lock or unlock on m.
At the beginning of an execution all vector clocks are empty. Whenever a thread t with vector clock V_t generates an event e, the following algorithm is executed [§]:

1. If e is not a fork event or a new thread event, then $V_t(t) = V_t(t) + 1$
2. If e is a read of a shared memory location m then
   $V_t = max\{V_t, V_m^w\}$ and $V_m^a = max\{V_m^a, V_t\}$
3. If e is a write, lock or unlock of a shared memory location m then
   $V_m^w = V_m^a = V_t = max\{V_m^a, V_t\}$
4. If e is a fork event and if t' is the newly created thread then
   $V_{t'} = V_t,\ V_t(t) = V_t(t) + 1$ and $V_{t'} = V_{t'} + 1$

[‡]algorithm from [6], p.30
[§]algorithm from [6], p.23

	V_{t_0}		V_{t_1}		V_x^a		V_x^w	
	t_0	t_1	t_0	t_1	t_0	t_1	t_0	t_1
init_0	0	0	0	0	0	0	0	0
fork_1	1	0	0	1	0	0	0	0
$(t_0, rd, x)_2$	**2**	**0**	0	1	2	0	0	0
$(t_1, rd, x)_3$	2	0	**0**	**2**	2	2	0	0
$(t_1, wr, x)_4$	2	0	**2**	**3**	2	3	2	3
$(t_0, rd, x)_5$	**3**	**3**	2	3	3	3	2	3

Figure 9: *Tracking of the dynamic vector clocks on an execution with two threads t_0 and t_1. For the last four rows $V\{e\}$ is shown bold.*

To understand how this algorithm works consider the following scenario: A program with two threads t_0 and t_1. At first thread t_0 creates t_2 and then reads the shared variable x twice. Thread t_1 writes on x and reads it afterwards. Figure 9 shows a possible execution of this code with the tracking of all vector clocks.

If an event e belongs to thread t, $V\{e\}$ denotes the vector clock V_t after event e. If e is an event of thread t, then the event in thread t that happened right before e is denoted by $prev(e)$. Respectively $next(e)$ denotes the event right after e. With this definition and the previously defined vector clocks the following theorem is used to identify race relations ([6], p.30):

THEOREM 1. *For any two event e and e', if the following holds:*

1. $V\{e\} \neq V\{prev(e')\}$ *given that* $prev(e')$ *exists, and*
2. $V\{next(e)\} \neq V\{e'\}$ *given that* $next(e)$ *exists, and*
3. $V\{e\} \leq V\{e'\}$, *and*
4. $VS_e \neq VS_{e'}$

then

- *if each of e and e' is a read or a write event, then $e \lessdot e'$,*
- *if e is an unlock event and e' is a lock event, then $e'' \lessdot e'$, where e'' is the lock event corresponding to e.*

A formal proof of this theorem can be found on [6] (p.24-30). Due to space limitations I will in the following only explain the intuition behind this proof, starting with the fourth condition of the theorem:
This condition reflects the non-sequentiality ($e \Updownarrow e'$) between both events, which is needed for the race relation. For a SVC holds by construction that two events e and e' are sequentially related if $VS\{e\} \leq VS\{e'\}$ or $VS\{e'\} \leq VS\{e\}$. Hence for checking the condition ($e \Updownarrow e'$) it is sufficient to check if $VS\{e\} \not\leq VS\{e'\}$ and $VS\{e'\} \not\leq VS\{e\}$. As events from the same thread are trivially sequential I will only consider events from different threads in the following.

The key idea behind the third condition is the following: As long as two threads t_0 and t_1 do not access the same memory location they will only increase their own counter $V_{t_0}(t_0)$ and $V_{t_1}(t_1)$ respectively. Consequently for two events e and e' (from t_0 and t_1 respectively)

holds $V\{e\}(t_0) > V\{e'\}(t_0)$ and $V\{e\}(t_1) < V\{e'\}(t_1)$, resulting in $V\{e\} \neq V\{e'\}$. However if both events access to the same memory location the vector clocks of this location (V_m^a and V_m^w) works like a mediator, propagating the values of the accessing events: The first event e saves its time stamp (after max-calculation) in the vector clock of the memory location. There the second event e' then reads this time stamp and update its own to the maximum of both, resulting in $V\{e\} \leq V\{e'\}$.

Indeed it does not work exactly like this, because two consecutive reads on the same location without a write beforehand are not in a race. Therefore the vector clock of each memory location is split up into the two vector clocks V_m^a and V_m^w. Since a read event will only update V_m^a without reading it and since it only reads the values in V_m^w, whereas a write event will update both and read from V_m^a, this problem is avoided.

Checking only $V\{e\} \leq V\{e'\}$ is not sufficient for a race condition, since this holds for almost all accesses on the same memory location. However only consecutive accesses can be in a race. This is reflected by the first and the second condition of the theorem:

Consider the following scenario: $(t_0, wr, x), (t_1, wr, x), (t_1, wr, x), (t_1, rd, y)$. Of course there is a race between the first and the second event and therefore $V\{e_1\} \leq V\{e_2\}$ holds. However vector clocks are increased monotonically. This means also $V\{e_1\} \leq V\{e_3\}$ and even $V\{e_1\} \leq V\{e_4\}$ hold transitively. Only checking the predecessor of e_4 exposes that e_3 or a prior event is in a race with e_1. Therefore e_1 and e_4 are not in a race since this would be a violation of the third race condition (see section 4.1).

5 Related Work

Being not familiar with symbolic execution, [1] provides a perfect introduction to the topic. It gives a compact and up-to-date overview about symbolic execution, concolic testing and execution generated testing without overwhelming the reader with technical details. Besides current main challenges and possible solutions are discussed.

Going historically, J.C. King was one of the first proposing the idea of symbolic execution already in 1976 [11]. He also came up with a tool called EFFIGY which provides symbolic execution for program testing. However, back then the constraint solvers were quite slow, which made this approach not very attractive. Almost 30 years passed until the progress in constraint satisfaction motivated a new approach. Around 2005 the dynamic technique of combining symbolic with concrete execution made this symbolic execution practical.

An elementary work on this topic is Godefroid et al.'s directed automated random testing (DART) [4]. Their tool uses a mixture of symbolic and concrete execution to generate directed test inputs. Moreover DART is able to automatically extract the unit interfaces from source code. However Dart can only handle integer constraints and falls back to random testing when pointer constraints appear.

Christian Cadar et al. introduced in [3] with their tool EXE the EGT method. This is a slightly different approach of mixing symbolic with concrete execution (see section 3.2). Like DART, EXE is able to automatically generate test inputs. However they admit that the handling of non-linear operations is very slow (p.9). Though they give a detailed description of the attached constraint solver.

KLEE [2] is a redesign of EXE from 2008. It improves the ability to handle interactions with the outside environment. They solve the problem of unavailable source code by providing a way to model the behaviour of the environment.

Returning to concolic testing, back in 2005 the CUTE project [5] extends DART by tracking symbolic pointer constraints in a simplified version. CUTE was written for C code and it uses memory graphs to represent the reachable memory addressed by a pointer. CUTE does not provide automated extraction of interfaces but leaves it up to the user to specify necessary preconditions [5, p.9].

jCUTE [6] - [9] is a extension of CUTE, written for Java, in order to handle concurrent programs. In his Ph.D. thesis Sen gives some very detailed insights in the theoretical basics of the race detection technique.

References

[1] C. CADAR, K. SEN. Symbolic Execution for Software Testing: Three Decades Later *Communications of the ACM* (2013).

[2] C. CADAR, D. DUNBAR AND D. ENGLER. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. *Proceedings of OSDI'08* (Dec. 2008)

[3] C. CADAR, V. GANESH, P. PAWLOWSKI, D. DILL AND D. AND ENGLER. EXE: Automatically generating inputs of death. *Proceedings of CCS'06* (Oct - Nov 2006). An extended version appeared in ACM TISSEC 12, 2 (2008).

[4] P. GODEFROID, N. KLARLUND AND K. SEN. DART : Directed Automated Random Testing. *Proceedings of PLDI'05* (June 2005)

[5] K. SEN, D. MARINOV, AND G. AGHA. CUTE: A concolic unit testing engine for C. *Proceedings of ESEC/FSE'05* (Sept. 2005)

[6] K. SEN. Scalable Automated Methods for Dynamic Program Analysis. Ph.D. thesis University of Illinois at Urbana-Champaign (June 2006)

[7] K. SEN AND G. AGHA. Automated systematic testing of open distributed programs. *Proceedings of FASE'06*

[8] K. SEN AND G. AGHA. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. *Proceedings of CAV'06*

[9] K. SEN AND G. AGHA. A race-detection and flipping algorithm for automated testing of multi-threaded programs. *Proceedings of HVC* (2006)

[10] P. GODEFROID. Partial-Order Methods for the Verification of Concurrent Systems An Approach to the State-Explosion Problem. *Volume 1032 of LNCS* Springer-Verlag (1996)

[11] J.C. KING. Symbolic execution and program testing. *Communications of the ACM* (July 1976), p. 385 - 394

[12] MYERS AND SANDLER AND BADGETT. The Art of Software Testing. *ITPro collection* Wiley (2011)