CUTE: A Concolic Unit Testing Engine for C

European Software Engineering Conference-
Foundations of Software Engineering (ESEC-FSE)
(2005)
Koushik Sen, Darko Marinov, and Gul Agha

2013-08-08
Yoo Jin Lim

© KAIST SE LAB 2013
Contents

- Introduction
- Related Work
- CUTE
  - Overall Approach
  - Example
  - Procedure
- Case Study
- Conclusion
- Discussion

© KAIST SE LAB 2013 2/22
Unit testing

- Applied to individual units of software independently
- Requires test inputs that explore all behaviors of unit
  - Aim to cover as much of the code as possible
- Execute unit on generated inputs to detect errors
  - Concrete Execution
  - Symbolic Execution
    - Concolic Execution
Concrete execution (a.k.a. Random testing)
- Execute unit *concretely* on randomly generated input values

Symbolic execution
- Execute unit *symbolically* on symbolic input values
- Goal: Generate concrete values that take different paths
  - Collect and solve symbolic constraints

```c
int bad_abs (int x){
    if (x < 0)
        return -x;
    if (x == 9)
        abort();  // ERROR
    return x;
}
```
Concolic Execution

- **Concolic** = Concrete + Symbolic
- Used in Directed Automated Random Testing (DART)

- Benefit over symbolic execution
  - Given unsolvable constraints, substitute random concrete value

During concrete execution, collect symbolic constraints
- Modify and solve constraints to direct next input to new path

\[ x = 9 \]
\[ x < 0 \]
\[ X \geq 0 \cap X = 9 \]
\[ X \geq 0 \cap X \neq 9 \]
DART’s limitations

- Simple strategy to generate structures using pointer input
  - Each pointer is either NULL or points to a NEW memory cell
- NO constraint solver for pointers

```c
typedef struct cell {
    int v;
    struct cell *next;
} cell;

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (p->next == p)
                ...
```

DART cannot test complex dynamic data structures using pointers
Motivations

- Programs which have dynamic data structures using pointers must be checked for correctness
- Computational cost of symbolic execution is high

Goals

- To improve DART,
  - Enable testing of programs with pointers and data structures
  - Speed up the constraint solving process
## Related Work

<table>
<thead>
<tr>
<th></th>
<th>DART</th>
<th>CUTE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Full name</strong></td>
<td>Directed Automated Random Testing</td>
<td>Concolic Unit Testing Engine</td>
</tr>
<tr>
<td><strong>Published date</strong></td>
<td>Jun. 2005</td>
<td>Sept. 2005</td>
</tr>
<tr>
<td><strong>Execution method</strong></td>
<td>Concolic Execution</td>
<td>Concolic Execution</td>
</tr>
<tr>
<td><strong>Subject program</strong></td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td><strong>Pointer input generation</strong></td>
<td>NULL, or new memory location</td>
<td>Appropriate location</td>
</tr>
<tr>
<td><strong>Basic constraint solving</strong></td>
<td>lp_solve*</td>
<td>Optimize before lp_solve*</td>
</tr>
<tr>
<td><strong>Pointer constraint solving</strong></td>
<td>X</td>
<td>O</td>
</tr>
</tbody>
</table>

*lp_solve : Linear Programming Solve

© KAIST SE LAB 2013
CUTE: Overall Approach

Represent

\[ p \rightarrow v = v_0, \]
\[ p \rightarrow \text{next} = n_0 \]

Track

\[ x \geq 1 \]
\[ p \rightarrow \text{NULL} \]
\[ n_0 = p_0 \]

Input Generation

Concolic Execution

Constraint Solving

DART

CUTE

© KAIST SE LAB 2013
typedef struct cell {
    int v;
    struct cell *next;
} cell;

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (p->next == p)
                abort();
    return 0;
}
```c
typedef struct cell {
    int v;
    struct cell *next;
} cell;

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (p->next == p)
                abort();
    return 0;
}
```

Concrete Execution

Symbolic Execution

Concrete state

symbolic state

constraints

CUTE: Example

© KAIST SE LAB 2013
typedef struct cell {
    int v;
    struct cell *next;
} cell;

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (p->next == p)
                abort();
    return 0;
}
typedef struct cell {
  int v;
  struct cell *next;
} cell;

int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (p->next == p)
        abort();
  return 0;
}
```c
typedef struct cell {
    int v;
    struct cell *next;
} cell;

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (p->next == p)
                abort();
    return 0;
}
```

Concrete Execution

Symbolic Execution

Concrete state

Symbolic state

solve: \( x_0 > 0 \) and \( p_0 \neq \text{NULL} \)

Constraints

\( x_0 > 0 \)

\( !(p_0 \neq \text{NULL}) \)
```c
typedef struct cell {
    int v;
    struct cell *next;
} cell;

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (p->next == p)
                abort();
    return 0;
}
```

Concrete Execution: `testme` is called with `p = NULL` and `x = 236`.

Symbolic Execution:
- `x_0 > 0` and `p_0 != NULL`.
- `x_0 = 236`, `p_0 = NULL`.
- Solve: `!(p_0 != NULL)`.

Constraints:
- `x_0 > 0` and `p_0 != NULL`.
- `x = 236`.
- `p = p_0`, `x = x_0`.

Concrete State: `p = NULL`, `x = 236`.

Symbolic State: `p_0 = NULL`.

Concrete Execution: `testme` returns 0.
typedef struct cell {
    int v;
    struct cell *next;
} cell;

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (p->next == p)
                abort();
    return 0;
}
```c
typedef struct cell {
    int v;
    struct cell *next;
} cell;

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (p->next == p)
                abort();
    return 0;
}
```

Concrete Execution

Symbolic Execution

Concrete state:
- p = p0
- x = x0
- p->v = v0
- p->next = n0

Symbolic state:
- p = p0, x = x0, p->v = v0, p->next = n0

Constraints:
- x0 > 0

© KAIST SE LAB 2013
typedef struct cell {
    int v;
    struct cell *next;
} cell;

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (p->next == p)
                abort();
    return 0;
}
typedef struct cell {
    int v;
    struct cell *next;
} cell;

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (p->next == p)
                abort();
    return 0;
}
```c
typedef struct cell {
    int v;
    struct cell *next;
} cell;

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (p->next == p)
                abort();
    return 0;
}
```
typedef struct cell {
    int v;
    struct cell *next;
} cell;

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (p->next == p)
                abort();
    return 0;
}
```c
typedef struct cell {
    int v;
    struct cell *next;
} cell;

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (p->next == p)
                abort();
    return 0;
}
```

**CUTE: Example**

Concrete Execution

Symbolic Execution

**Concrete State**
- \( p = p_0 \)
- \( x = x_0 \)
- \( p \rightarrow v = v_0 \)
- \( p \rightarrow next = n_0 \)

**Symbolic State**
- \( x_0 > 0 \)
- \( p_0 \neq NULL \)
- \( n_0 = p_0 \)

**Constraints**
- \( x_0 > 0 \)
- \( p_0 \neq NULL \)
- \( n_0 \neq p_0 \)

**Solve:**
- \( x_0 > 0 \) and \( p_0 \neq NULL \) and \( n_0 = p_0 \)
typedef struct cell {
    int v;
    struct cell *next;
} cell;

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (p->next == p)
                abort();
    return 0;
}
typedef struct cell {
    int v;
    struct cell *next;
} cell;

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (p->next == p)
                abort();
    return 0;
}
```c
typedef struct cell {
    int v;
    struct cell *next;
} cell;

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (p->next == p)
                abort();
    return 0;
}
```

Concrete Execution
Symbolic Execution

Concrete state
Symbolic state
Constraints

CUTE: Example

Concrete state:
- `p = p_0`
- `x = x_0`
- `p->v = v_0`
- `p->next = n_0`

Symbolic state:
- `p = p_0, x = x_0, p->v = v_0, p->next = n_0`

Constraints:
- `x_0 > 0`
- `p_0 \neq NULL`
```c
typedef struct cell {
    int v;
    struct cell *next;
} cell;

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (p->next == p)
                abort();
    return 0;
}
```

CUTE: Example

Concrete Execution

Symbolic Execution

Concrete state

Program Error

symbolic state

Constraints

\[ x_0 > 0 \]

\[ p_0 \neq \text{NULL} \]

\[ n_0 = p_0 \]
```c
typedef struct cell {
    int v;
    struct cell *next;
} cell;

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (p->next == p)
                abort();
    return 0;
}
```

**CUTE: Example**

- **Input 1:**
  - `p` pointing to `NULL`, `x=236`

- **Input 2:**
  - `p` pointing to `634`, `x=236`

- **Input 3:**
  - `p` pointing to `634`, `x=236`
Logical Input Map

- Simple way to serialize inputs including pointers
  - Yet, precisely shows how data structures are connected
- Maps logical addresses to
  - Numerical values (regular input) or Logical addresses (pointer input)

```c
int testme(cell *p, int x)
```

Input 1:
```
NULL
p
```

Logical Address: 0 1 2
- Logical or Primitive: NULL 0 236

Input 3:
```
634
```

Logical Address: 0 1 2 3 4
- Logical or Primitive: NULL 3 236

More diagrams and inputs are shown, including values and structures for `p`, `x`, and pointers `p->v` and `p->next`. An example of an input generation process is also demonstrated.
**CUTE: Procedure**

**Concolic Execution: Instrumentation**
- To perform symbolic computation for each line of code

**Procedure**
- Use C Intermediate Language (CIL) to convert C code
- Insert functions throughout the simplified code
- Compile → executable file
- Run `cute`, which executes the instrumented program

---

<table>
<thead>
<tr>
<th>Before Instrumentation</th>
<th>After Instrumentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>// assignment</td>
<td>execute_symbolic(&amp;v, “e”);</td>
</tr>
<tr>
<td>v ← e;</td>
<td>v ← e;</td>
</tr>
<tr>
<td>// conditional</td>
<td>evaluate_predicate(“p”, p)</td>
</tr>
<tr>
<td>if (p) goto l</td>
<td>if (p) goto l</td>
</tr>
<tr>
<td>// normal termination</td>
<td>solve_constraint();</td>
</tr>
<tr>
<td>HALT</td>
<td>HALT;</td>
</tr>
</tbody>
</table>
Concolic Execution: In Action

- Concretely on the logical input map
- Symbolically on the symbolic states, through functions
  - `execute_symbolic (m, e)`: Map e to m in the appropriate symbolic state
  - `evaluate_predicate (p, b)`: Collect symbolic constraints from branch points
- When the program halts, solve constraints ➔ new input
CUTE: Procedure

- Constraint Solving

- Path constraints from two consecutive executions are similar
  - Previous solution more or less similar to current solution
- Optimize by reducing the number of constraints to solve
CUTE: Procedure

- Optimizations before Constraint Solving
  - (OPT 1) Fast unsatisfiability check
    ```
    if(x == 2)
    if(y > 0)
    if(x != 2);  // Negation of previous constraint → unreachable
    ```
  - (OPT 2) Common sub-constraints elimination
    ```
    for (int x = 0, x < 10, x++){
    if((y > 0) && (y == x));  // Constraint remains the same throughout
    }
    ```
  - (OPT 3) Incremental solving
    ```
    if(x > 2)  // Does not share variable with the last constraint
    if(y > 0)
    if (y < 5);  // Look at the last constraint
    ```
CUTE: Procedure

Constraint Solving

- Arithmetic
  - Use `lp_solve`, linear arithmetic constraint solver

- Pointer
  - Use a graph

© KAIST SE LAB 2013
CUTE: Procedure

- Constraint Solving - Pointers
  - Construct an undirected graph
    - Node: pointer location
    - Edge: inequality ≠ in the path condition
  - Use the graph to solve (in)equality constraints

```c
typedef struct cell {
    int v;
    struct cell *next;
} cell;

int testme(cell *p, int x) {
    ...
    if (p != NULL) {
        if (p->next == p) ...
    }
```

- Solve p₀ ≠ NULL
  - Add an edge
- Solve n₀ = p₀
  - Merge nodes
Case Study (1/2)

❖ Goal
  ▪ Show how CUTE can detect errors
    • memory leaks in addition to standard errors

❖ Method
  ▪ Run CUTE on CUTE’s non-standard data-structures
    • Ex) cu_linear (linear symbolic expressions)
    • Ex) cu_pointer (pointer symbolic expressions)
  ▪ Use in combination with Valgrind to detect memory leaks

❖ Result
  ▪ Discovered a few memory leaks and seg faults
  ▪ These errors did not show up in other uses of CUTE
Case Study (2/2)

❖ Goal
  ▪ To measure the efficiency of CUTE

❖ Method
  ▪ Run CUTE on SGLIB (Simple Generic Library for C data structures)

❖ Result

<table>
<thead>
<tr>
<th>Name</th>
<th>Runtime in sec</th>
<th>% Branch Coverage</th>
<th>OPT 1 in %</th>
<th>OPT 2 &amp; 3 in %</th>
<th># of Bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array Quick Sort</td>
<td>2</td>
<td>97.73</td>
<td>67.80</td>
<td>49.13</td>
<td>0</td>
</tr>
<tr>
<td>Array Heap Sort</td>
<td>4</td>
<td>100.00</td>
<td>71.10</td>
<td>46.38</td>
<td>0</td>
</tr>
<tr>
<td>Doubly Linked List</td>
<td>3</td>
<td>99.12</td>
<td>86.95</td>
<td>79.38</td>
<td>1</td>
</tr>
<tr>
<td>Hash Table</td>
<td>1</td>
<td>85.19</td>
<td>97.01</td>
<td>52.94</td>
<td>1</td>
</tr>
<tr>
<td>Red Black Tree</td>
<td>2629</td>
<td>71.18</td>
<td>89.65</td>
<td>64.93</td>
<td>0</td>
</tr>
</tbody>
</table>

- Found two bugs
- Optimization → constraint solving time reduced considerably
  - High % of executions unsatisfiability check was successful
  - High average % of constraints eliminated in each execution

ⓒ KAIST SE LAB 2013
Conclusion

❖ Contributions

▪ Provided a method for representing and solving pointer constraints to generate test inputs
  • Now applicable to a broader class of C programs

▪ Proposed an efficient constraint solver
  • Facilitated input generation for unit testing

❖ Future Work

▪ Apply to a broader class of programs in general
  • Programs with concurrency and in other languages
**Discussion**

**Pros**
- Developed practical tools for the proposed approaches
- Discovered bugs in widely used software

**Cons**
- Bounded DFS lacks completeness
- Correctness vs. Performance trade-off
- Small samples and no statistical analysis in case study
Thank You.
Benefit of Concolic Execution

- Given unsolvable constraints, substitute random concrete value

<table>
<thead>
<tr>
<th>Example</th>
<th>Concrete Values</th>
<th>Path Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>void testme(int a, int b) { if (a * b &gt; 100) //target }</td>
<td>a = -10, b = -11</td>
<td>(-10 * b) &gt; 100</td>
</tr>
</tbody>
</table>
Appendix

- Execution paths of a program
  - Node: “if then else” statement
  - Edge: non-conditional statements
  - Path: equivalent class of inputs

- Traverse all execution paths one by one to detect errors
  - Achieving 100% branch coverage
  - Check for assertion violation and program crash
CIL

- Convert complex statements into simplified form
  - Three Address Code

\[
\begin{align*}
V &= x + y + z; \\
V &\leftarrow x + y \\
V &\leftarrow V + z
\end{align*}
\]
Case Study 1 Result

- Discovered a few memory leaks and seg faults
  - Ex) `cu_linear_add (cu_linear *c1, cu_linear *c2)`
    If adding two linear expressions that add up to a constant, NULL is returned without freeing the memory.

```c
cu_linear *
cu_linear_add(cu_linear *c1, cu_linear *c2, int add) {
    int i, j, k, flag;
    cu_linear* ret=(cu_linear*)malloc(sizeof(cu_linear));
    ... // skipped 18 lines of code
    if(ret->count==0) return NULL;
```
Appendix

Case Study 2 Result

- Found **two bugs** in SGLIB1.0.1
  1) **doubly-linked list library**
     - seg fault when a non-zero length list is concatenated with a zero-length list
     - discovered in 140 iterations (< 1 second)
  2) **hash-table**
     - an infinite loop in hash table is_member function
     - 193 iterations (1 second)
Pointer Approximation Example

```c
*p = 0;
*q = 1;
if (*p == 1)
    ERROR
...```

### Diagram

```
 P → 1
 Q → 1
```

```
 P → 0
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```

```
 P → 1
 Q → 1
```