An Efficient Black-Box Support of Advanced Coverage Criteria for Klee

Published in SAC'23, Software Verification and Testing track

Klee workshop, April 15-16, 2024, Lisbon

Nicolas Berthier
Steven De Oliveira

OCaml PRO

Nikolai Kosmatov
Delphine Longuet
Romain Soulat

THALES

Motivation

Only coverage criterion targeted by Klee: **all-path**

But may be:

- Too strong if target is instructions or decisions e.g.
- Too weak if target is a criterion incomparable with all-path (mutations, limits...)

Limitations of Klee all-path coverage

Generated tests for t of size 2

	t	n	v
Test 1	[0,0]	0	0
Test 2	[0,0]	1	0
Test 3	[0,0]	1	167
Test 4	[1,0]	2	0
Test 5	[0,0]	2	167

```
int search (int *t, int n, int v) {
  int res = 0, i = 0;
  while (!res && i < n) {
    if (t[i] == v)
      res = 1;
    i++;
  }
  return res;
}
```

Preconditions
size of $t \geq 0$
$0 \leq n \leq$ size of t

Covered: instructions, decisions and conditions

But with more tests than necessary

➔ improve Klee efficiency on simple criteria

Not covered: multicondition **`(res && i < n)`**

(finding v before the end of a non-empty array)

Limitations of Klee all-path coverage

Generated tests for t of size 2

	t	n	v
Test 1	[0,0]	0	0
Test 2	[0,0]	1	0
Test 3	[0,0]	1	167
Test 4	[1,0]	2	0
Test 5	[0,0]	2	167
Test 6	**[0,0]**	**2**	**0**

```
int search (int *t, int n, int v) {
  int res = 0, i = 0;
  while (!res && i < n) {
    if (t[i] == v)
      res = 1;
    i++;
    assert(!(res && i < n));
  }
  return res;
}
```

Preconditions
size of $t \geq 0$
$0 \leq n \leq$ size of t

Covered: instructions, decisions and conditions

But with more tests than necessary

➔ improve Klee efficiency on simple criteria

Covered: multiconditions

But with a complementary assertion

➔ improve Klee coverage on criteria incomparable to all-path

Motivation

Only coverage criterion targeted by Klee: **all-path**

But may be:

- Too strong if target is instructions or decisions e.g.
- Too weak if target is a criterion incomparable with all-path (mutations, limits...)

How can we make Klee efficiently support other coverage criteria without modifying the tool itself?

Coverage labels [Bardin et al. ICST'14]

Generic approach to represent coverage criteria as source code annotations by test objectives to be targeted by tools

For a test suite, covering all labels for a criterion = satisfying the criterion

```
statement1;
if (x < y)
  {...}
statement2;
```

→

```
statement1;
// l1: x < y
// l2: !(x < y)
if (x < y)
  {...}
statement2;
```

Decision coverage (DC)

```
statement1;

// l1: x<y && -1<=x-y && x-y<=1
if (x < y)
  {...}
statement2;
```

Boundary coverage (LIMIT)

label = (location, property)

```
statement1;
x = a+b;
statement2;
```

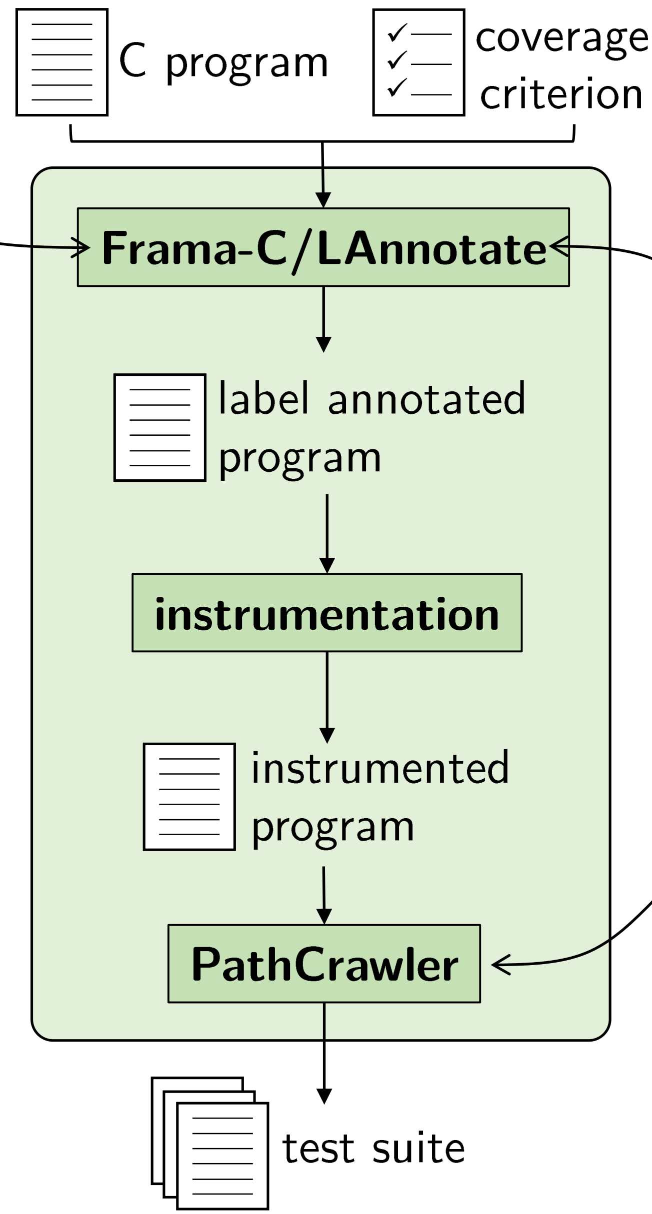
→

```
statement1;
// l1: a != abs(a)  (ABS)
// l2: b != abs(b)  (ABS)
// l3: a+b != a-b   (AOR)
// l4: a+b != a*b   (AOR)
// l5: a+b != a/b   (AOR)
x = a+b;
statement2;
```

Weak mutation coverage (WM)

Test generation for labels [Bardin et al., SCP'21]

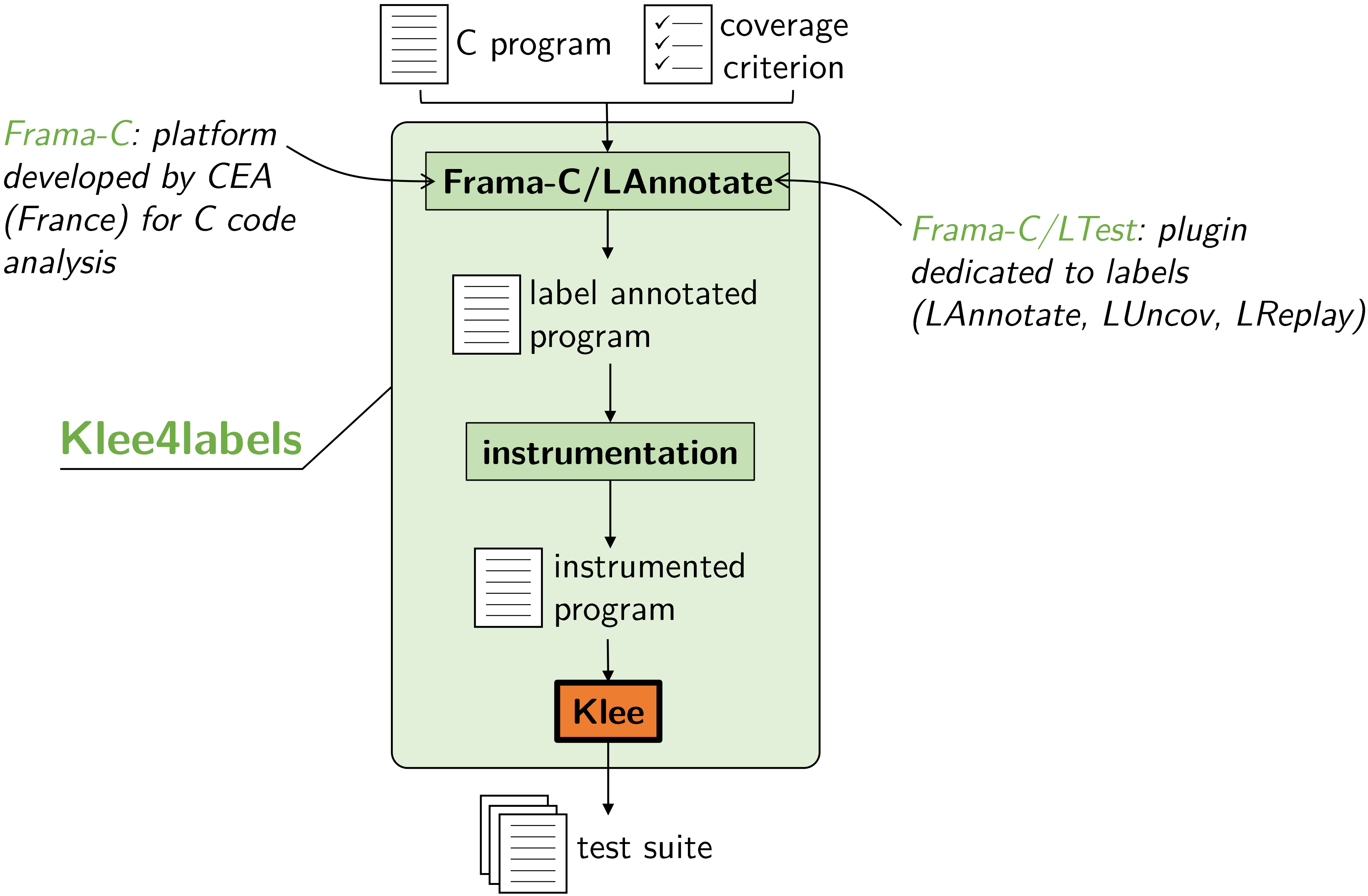
Frama-C: platform developed by CEA (France) for C code analysis



Frama-C/LTest: plugin dedicated to labels (LAnnotate, LUncov, LReplay)

PathCrawler: dynamic symbolic execution tool developed by CEA (France)

Test generation for labels **with Klee?**



Label instrumentation

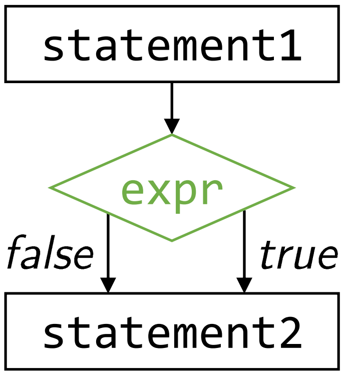
Naive instrumentation: addition of a branching condition for each label

Drawbacks:

- Exponential growth of the path space
- Multiple visits of the same label

```
statement1;
// l: expr
statement2;
```

→



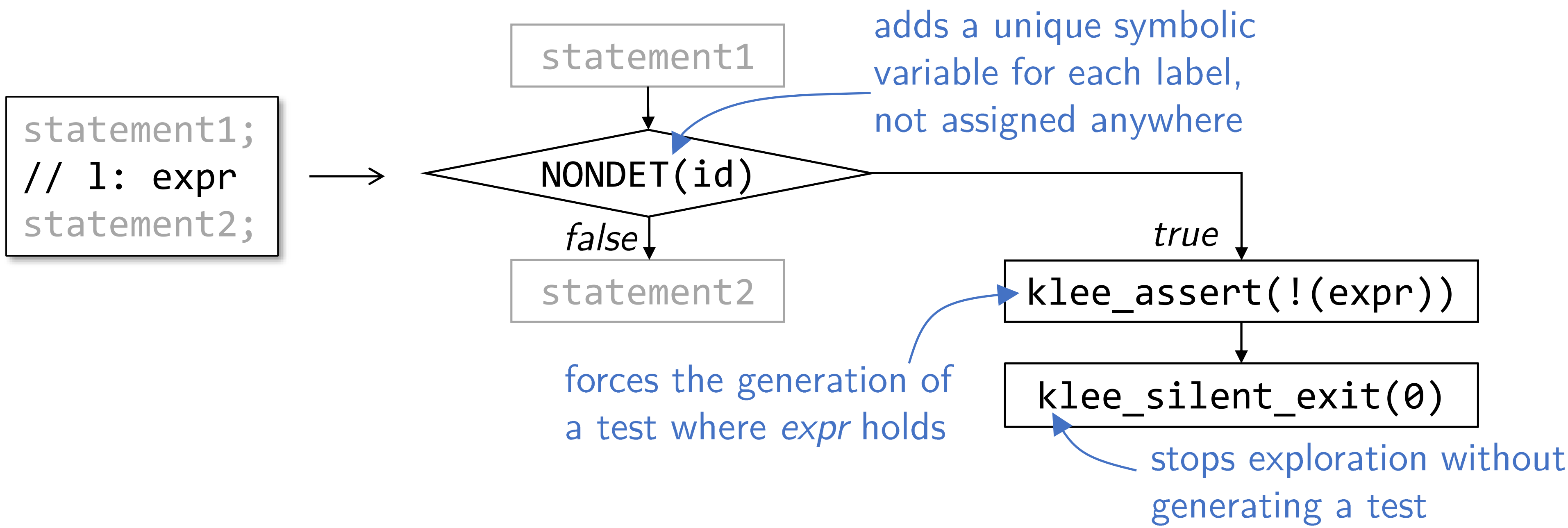
Optimized instrumentation

- Tight instrumentation: path ends after visiting a label
- Iterative label deletion: replay of each generated test to delete all covered labels along the execution path

Tight instrumentation

Aim of tight instrumentation for Klee

- Add the minimum of paths needed for labels
- Stop exploration as soon as a label is reached



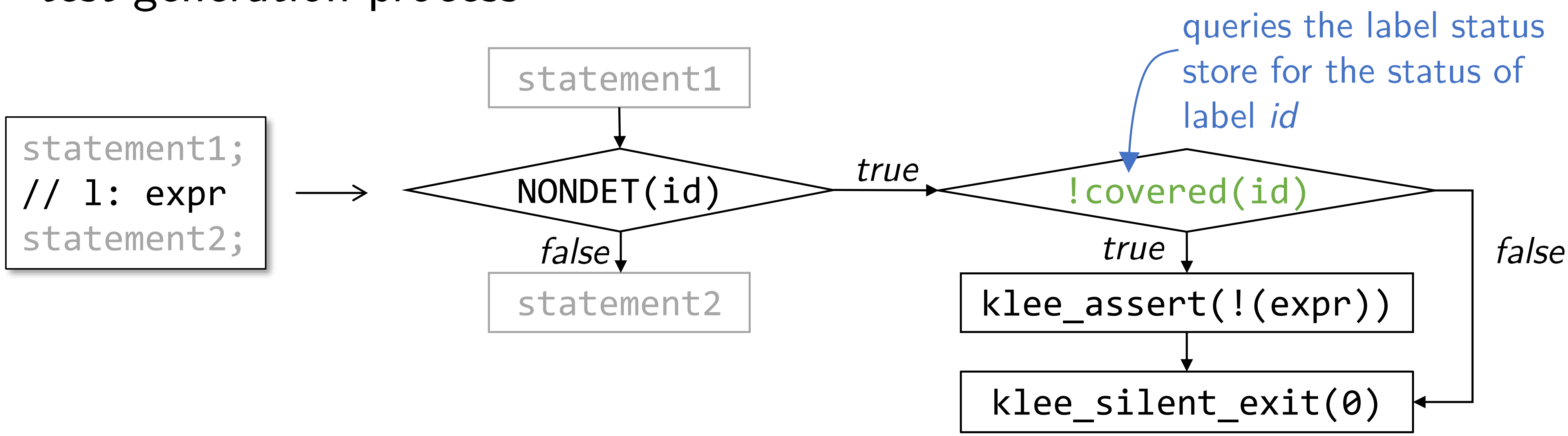
Benefit: only keep test cases generated for `klee_assert` (`.assert.err`)

Iterative label deletion

Aim of iterative label deletion for Klee

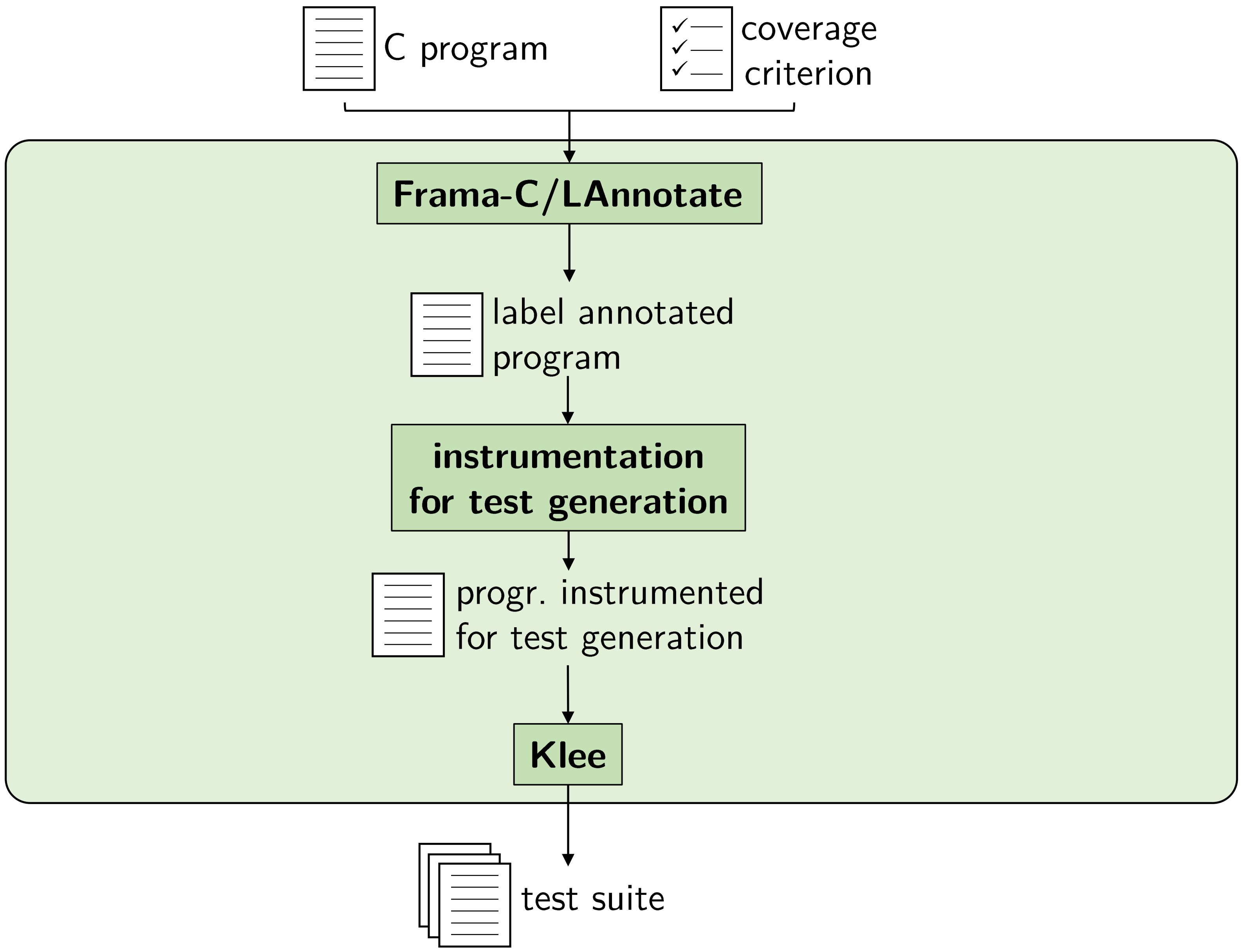
- Avoid targetting a label already covered by a previous test

Replay of a test case immediately after its generation, in parallel of the test generation process

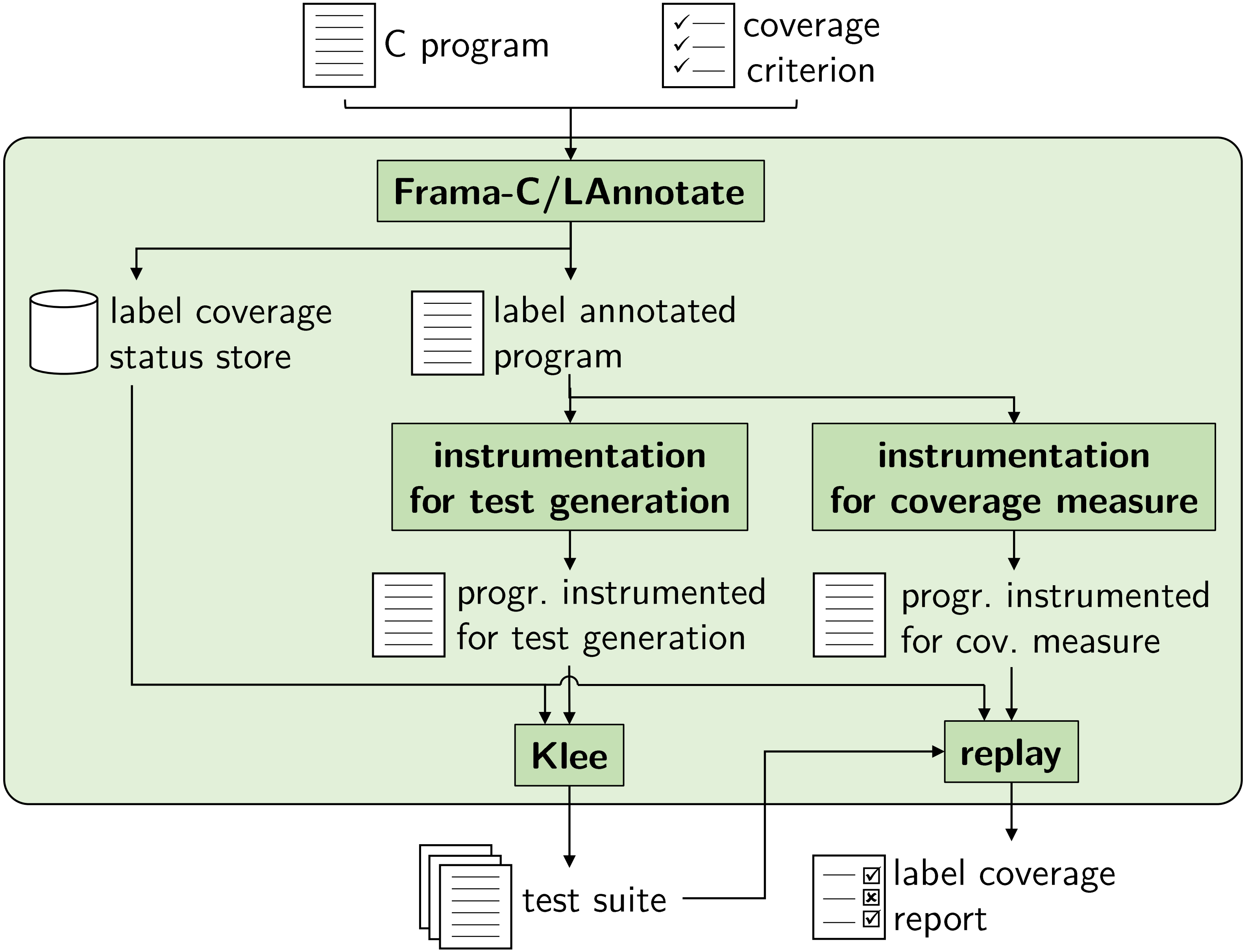


Benefit: condition of a label considered only when necessary (at most once on a program path and only if the label is not yet covered)

Klee4labels



Klee4labels



Klee4labels prototype for evaluation

Publicly available prototype: github.com/OCamlPro/klee4labels

- 700 lines of OCaml code
- 300 lines of C for instrumentation macros and library of external functions

Proprietary optimized version with more advanced implementation of the label coverage store

Results of the evaluation of the optimized version of Klee4labels

1. Higher coverage of labels
2. Reasonable size of generated test suites
3. Reasonable time overhead of test generation

Evaluation results

Program (nb loc)	Cov. criterion (nb labels)	Klee (cov., nb tests, time)			Opt. Klee4labels (cov., nb tests, time)		
power (18)	decisions (4)	100%	3	8.3 s	100%	2	1.2 s
	mutations (25)	12%	3	8.4 s	84%	7	27 s
tritype (22)	multicond. (38)	71%	14	0.8 s	100%	24	1.7 s
	mutations (101)	58%	14	0.5 s	91%	22	1.3 s
modulus (25)	decisions (8)	100%	5	timeout	100%	3	1.2 s
checkutf8 (74)	mutations (178)	45%	23	2.4 s	80%	44	18.5 s
	limits (25)	56%	23	2.0 s	100%	25	3.7 s
tcas (110)	multicond. (66)	77%	23	0.4 s	80%	13	2.2 s
	mutations (87)	44%	18	0.6 s	60%	18	3.6 s
gd_full_bad (156)	limits (19)	32%	33	3.4 s	84%	16	5.4 s

timeout = 60 s

1. Higher coverage of labels

Program (nb loc)	Cov. criterion (nb labels)	Klee (cov., nb tests, time)			Opt. Klee4labels (cov., nb tests, time)			Diff. cov.
power (18)	decisions (4)	100%	3	8.3 s	100%	2	1.2 s	
	mutations (25)	12%	3	8.4 s	**84%**	7	27 s	**+72**
tritype (22)	multicond. (38)	71%	14	0.8 s	**100%**	24	1.7 s	**+29**
	mutations (101)	58%	14	0.5 s	**91%**	22	1.3 s	**+33**
modulus (25)	decisions (8)	100%	5	timeout	100%	3	1.2 s	
checkutf8 (74)	mutations (178)	45%	23	2.4 s	**80%**	44	18.5 s	**+35**
	limits (25)	56%	23	2.0 s	**100%**	25	3.7 s	**+44**
tcas (110)	multicond. (66)	77%	23	0.4 s	**80%**	13	2.2 s	**+3**
	mutations (87)	44%	18	0.6 s	**60%**	18	3.6 s	**+16**
gd_full_bad (156)	limits (19)	32%	33	3.4 s	**84%**	16	5.4 s	**+53**

timeout = 60 s

Better to far better coverage for criteria stronger than all-path
All feasible labels are covered

2. Reasonable size of test suites

Program (nb loc)	Cov. criterion (nb labels)	Klee (cov., nb tests, time)			Opt. Klee4labels (cov., nb tests, time)			Diff. #tests
power (18)	decisions (4)	100%	3	8.3 s	100%	2	1.2 s	**×0.7**
	mutations (25)	12%	3	8.4 s	84%	7	27 s	×2.3
tritype (22)	multicond. (38)	71%	14	0.8 s	100%	24	1.7 s	×1.7
	mutations (101)	58%	14	0.5 s	91%	22	1.3 s	×1.6
modulus (25)	decisions (8)	100%	5	timeout	100%	3	1.2 s	**×0.6**
checkutf8 (74)	mutations (178)	45%	23	2.4 s	80%	44	18.5 s	×1.9
	limits (25)	56%	23	2.0 s	100%	25	3.7 s	×1.1
tcas (110)	multicond. (66)	77%	23	0.4 s	80%	13	2.2 s	**×0.6**
	mutations (87)	44%	18	0.6 s	60%	18	3.6 s	×1.0
gd_full_bad (156)	limits (19)	32%	33	3.4 s	84%	16	5.4 s	**×0.5**

timeout = 60 s

More accurate tests, sometimes even fewer tests to achieve better coverage

3. Reasonable time overhead of generation

Program (nb loc)	Cov. criterion (nb labels)	Klee (cov., nb tests, time)			Opt. Klee4labels (cov., nb tests, time)			Diff. time
power (18)	decisions (4)	100%	3	8.3 s	100%	2	1.2 s	**×0.1**
	mutations (25)	12%	3	8.4 s	84%	7	27 s	×3.2
tritype (22)	multicond. (38)	71%	14	0.8 s	100%	24	1.7 s	**×2.1**
	mutations (101)	58%	14	0.5 s	91%	22	1.3 s	×2.6
modulus (25)	decisions (8)	100%	5	**timeout**	100%	3	**1.2 s**	
checkutf8 (74)	mutations (178)	45%	23	2.4 s	80%	44	18.5 s	×7.7
	limits (25)	56%	23	2.0 s	100%	25	3.7 s	**×1.9**
tcas (110)	multicond. (66)	77%	23	0.4 s	80%	13	2.2 s	×5.5
	mutations (87)	44%	18	0.6 s	60%	18	3.6 s	×6.0
gd_full_bad (156)	limits (19)	32%	33	3.4 s	84%	16	5.4 s	×1.6

timeout = 60 s

Small time overhead for fully satisfiable criteria
Otherwise, time lost on uncoverable labels

Conclusions and future work

Lightweight black-box integration of labels for Klee

- No need to modify the underlying test generation strategy
- Direct benefit of the various strategies and optimizations of the tool

Main results

- Efficient coverage of basic criteria with fewer and more targeted test cases than when Klee is used directly
- High coverage of more advanced criteria with a reasonable overhead

Future work

- Industrial evaluation on real-life code
- Detecting infeasible objectives prior to test generation
- Support of hyperlabels
- Integration of labels in other white- or gray-box test generation tools