Towards Evaluating Size Reduction Techniques for Software Model Checking

Gyula Sallai[1], Ákos Hajdu[1,2], Tamás Tóth[1], Zoltán Micskei[1]

[1]*Department of Measurement and Information Systems, Budapest University of Technology and Economics*

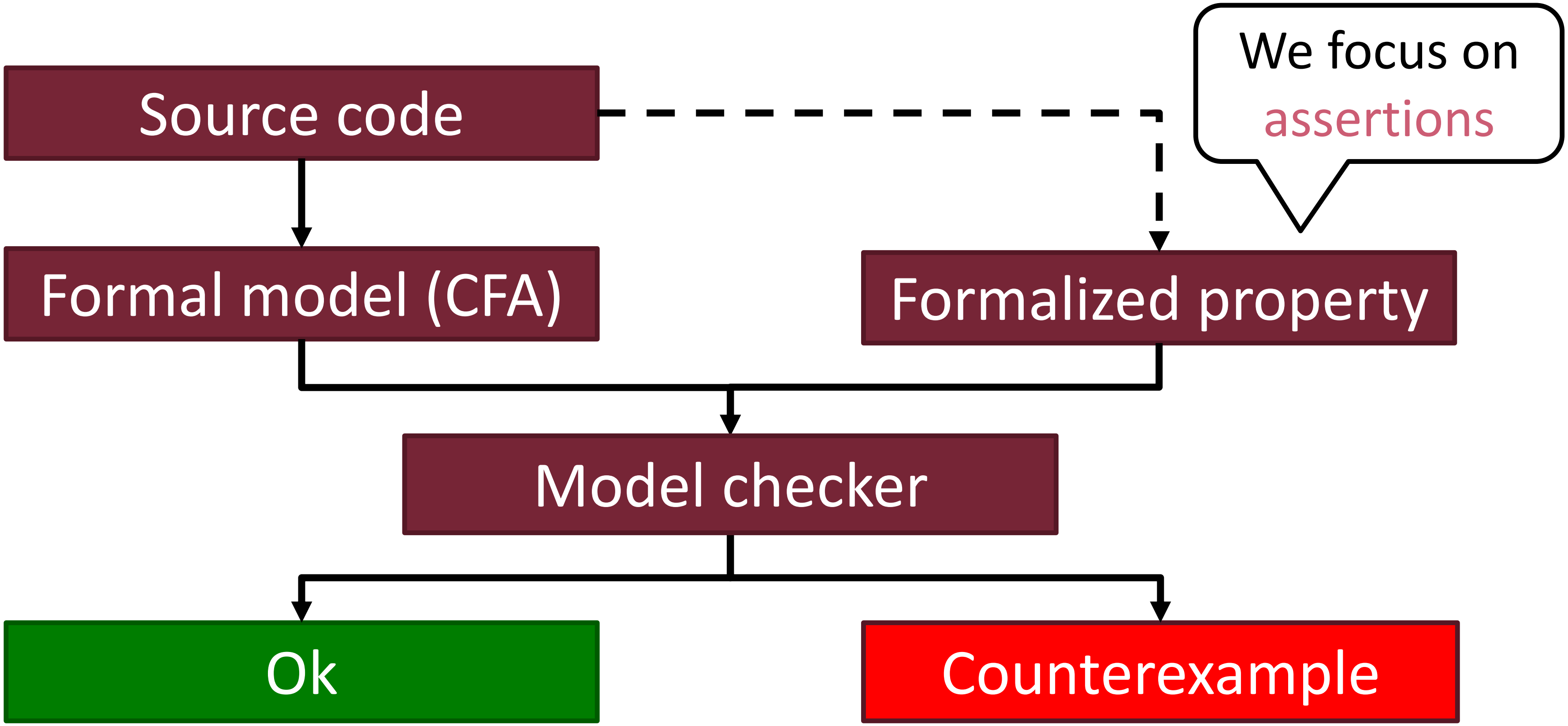
[2]*MTA-BME Lendület Cyber-Physical Systems Research Group, Budapest, Hungary*

VPT 2017, Uppsala, Sweden, 29.04.2017.

Introduction

Software model checking

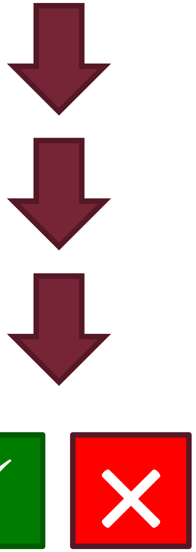
- Proving correctness formally
  - Problem: state space explosion



Motivation

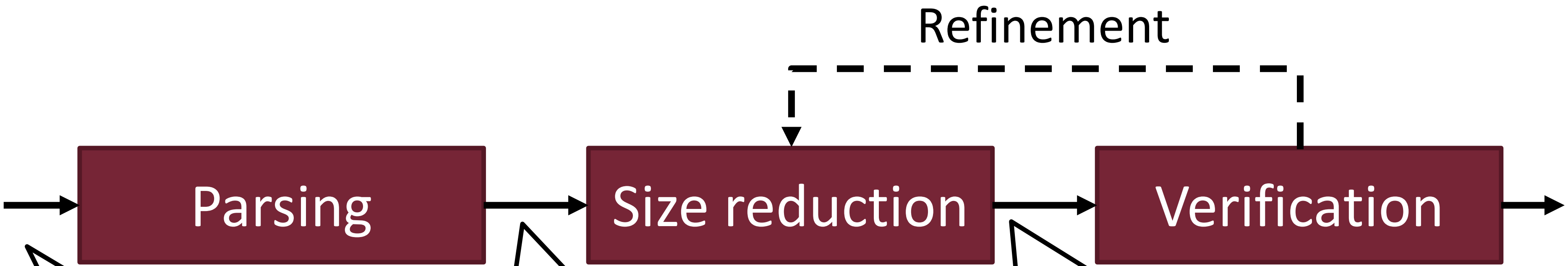
- Integrated, configurable workflow
  - From source code to verification results
  - Enhanced by size reduction techniques
    - Compiler techniques
    - Slicing
  - Supported by a verification framework
    - Based on abstraction and CEGAR
    - Highly configurable
- Evaluation
  - Impact of size reduction on verification

```c
#include <assert.h>

int main(void) {
    int i = 0;
    int sum = 0;

    while (i < 11) {
        sum = sum + i;
        i = i + 1;
    }

    assert(i == 11);

    return 0;
}
```



Workflow

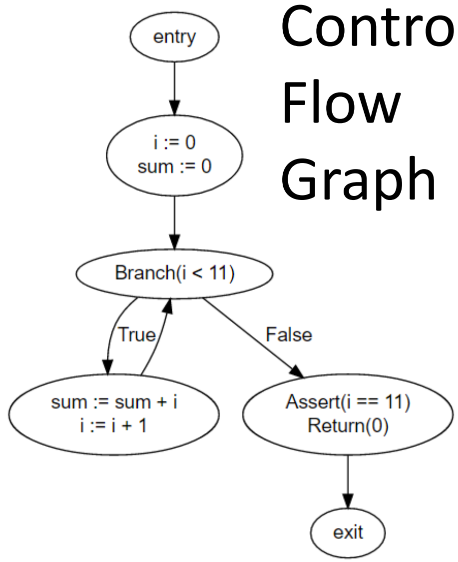
Workflow – Overview



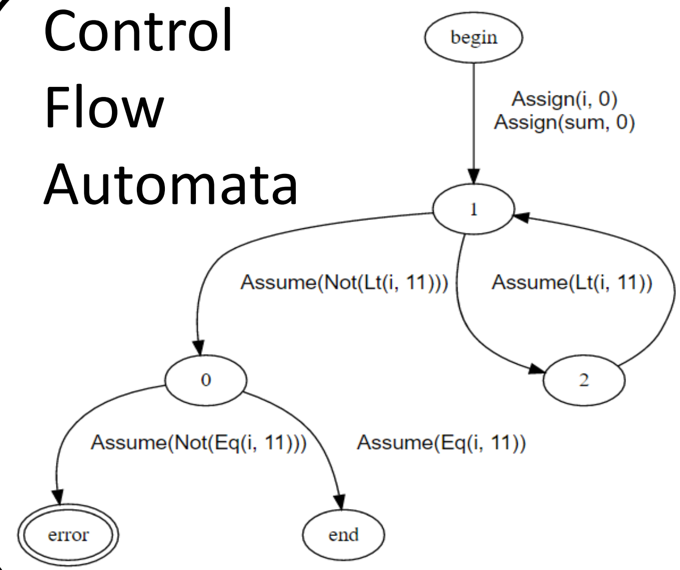
C code

```c
#include <assert.h>

int main(void) {
    int i = 0;
    int sum = 0;

    while (i < 11) {
        sum = sum + i;
        i = i + 1;
    }

    assert(i == 11);

    return 0;
}
```

Control Flow Graph



Control Flow Automata



Size reduction techniques

- Compiler optimizations
  - Constant folding and propagation

```
int x = 5 * 2;
int y = x + 2;
```

→

```
int x = 10;
int y = 12;
```

  - Dead branch elimination

```
x = false;
if (x) {
    ...
}
```

→

```
x = false;
```

  - Function inlining

```
int add(int x, int y) { return x + y; }

x = add(y, z);
```

→

```
x = y + z;
```

Size reduction techniques

- Program slicing
  - Slice: subprogram that produces the same output and assigns the same values to a set of variables

```
0: int i = 0;
1: int x = 0;
2: while (i < 11) {
3:    x = x + i;
4:    i = i + 1;
   }
5: assert(i != 0);
```

```
0: int i = 0;
1: int x = 0;
2: while (i < 11) {
3:    x = x + i;
4:    i = i + 1;
   }
5: assert(i != 0);
```

Criterion: *value of **i** at statement 5*

Size reduction techniques

- Backward slicing
  - Retain all instructions crucial to criterion
    - Data flow and control dependencies
  - Accurate slices
- Thin slicing
  - Retain data flow dependency only
    - Replace control dependencies with abstract predicates
  - Spurious counterexample → refinement of slice
- Value slicing
  - Middle ground between backward and thin
    - Retain variables determining control criterions

Size reduction techniques

Original

```
int u = 0;
int t = 0;
int x = 0;
while (t < 1000) {
  int s = nondet();
  int y;
  if (s == 1) {
    y = x * 2;
  } else {
    y = x - 1;
  }
  assert(y != 0);
  x = x + y;
  t = t + 1;
  u = u + t;
}
printf("u=%d", u);
```

Backward

```
int u = 0;
int t = 0;
int x = 0;
while (t < 1000) {
  int s = nondet();
  int y;
  if (s == 1) {
    y = x * 2;
  } else {
    y = x - 1;
  }
  assert(y != 0);
  x = x + y;
  t = t + 1;
  u = u + t;
}
printf("u=%d", u);
```

Thin

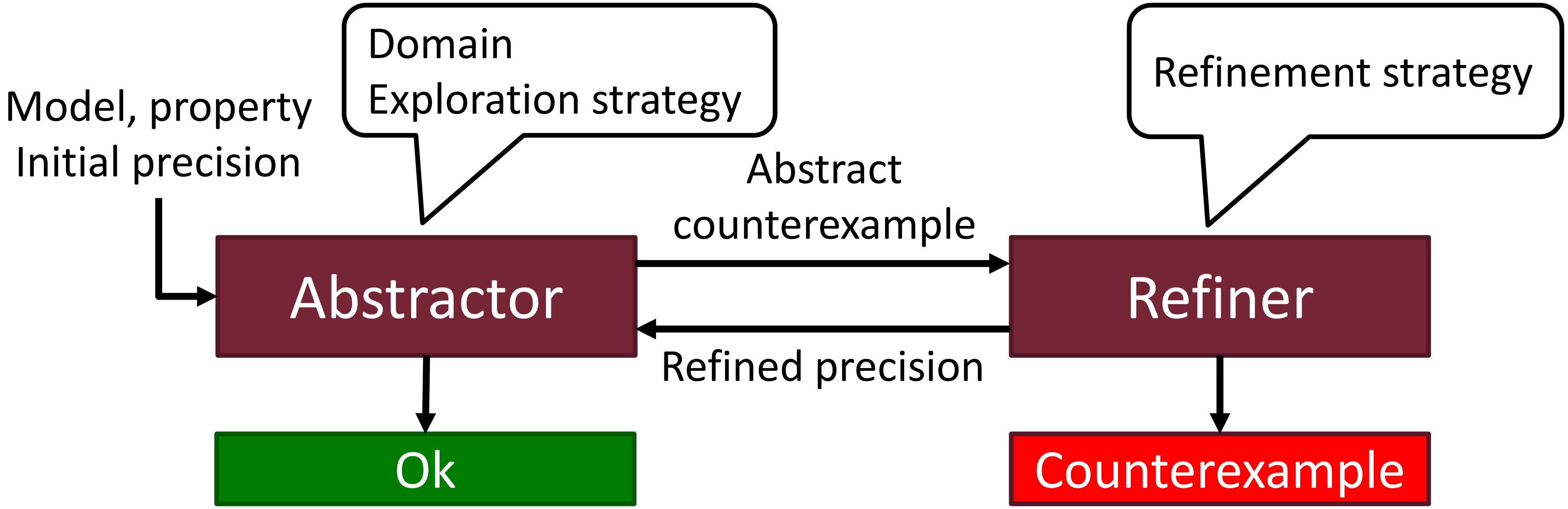
```
int u = 0;
int t = 0;
int x = 0;
while (ϕ1) {
  int s = nondet();
  int y;
  if (ϕ2) {
    y = x * 2;
  } else {
    y = x - 1;
  }
  assert(y != 0);
  x = x + y;
  t = t + 1;
  u = u + t;
}
printf("u=%d", u);
```

Value

```
int u = 0;
int t = 0;
int x = 0;
while (ϕ1) {
  int s = nondet();
  int y;
  if (s == 1) {
    y = x * 2;
  } else {
    y = x - 1;
  }
  assert(y != 0);
  x = x + y;
  t = t + 1;
  u = u + t;
}
printf("u=%d", u);
```

Verification

- CEGAR
  - Counterexample-Guided Abstraction Refinement
  - Configurable framework



Evaluation

Objects

- Models: SV-COMP examples
  - Locks: locking mechanisms
    - 100-150 LOC, many smaller slices
  - ECA: event-driven systems
    - 500-600 LOC, one slice
  - SSH-simplified: server-client systems
    - 500-600 LOC, one slice

- Requirement: reachability of assertion violation

Environment

- Algorithms
  - Slicing: None / Backward / Value / Thin
  - Compiler optimizations: True / False
  - Domain: Predicate abstraction
  - Refinement: Sequence interpolation
  - Exploration strategy: BFS / DFS
- A configuration
  - Slicing + optimizations + exploration strategy
  - E.g.: BTD → Backward, True, DFS

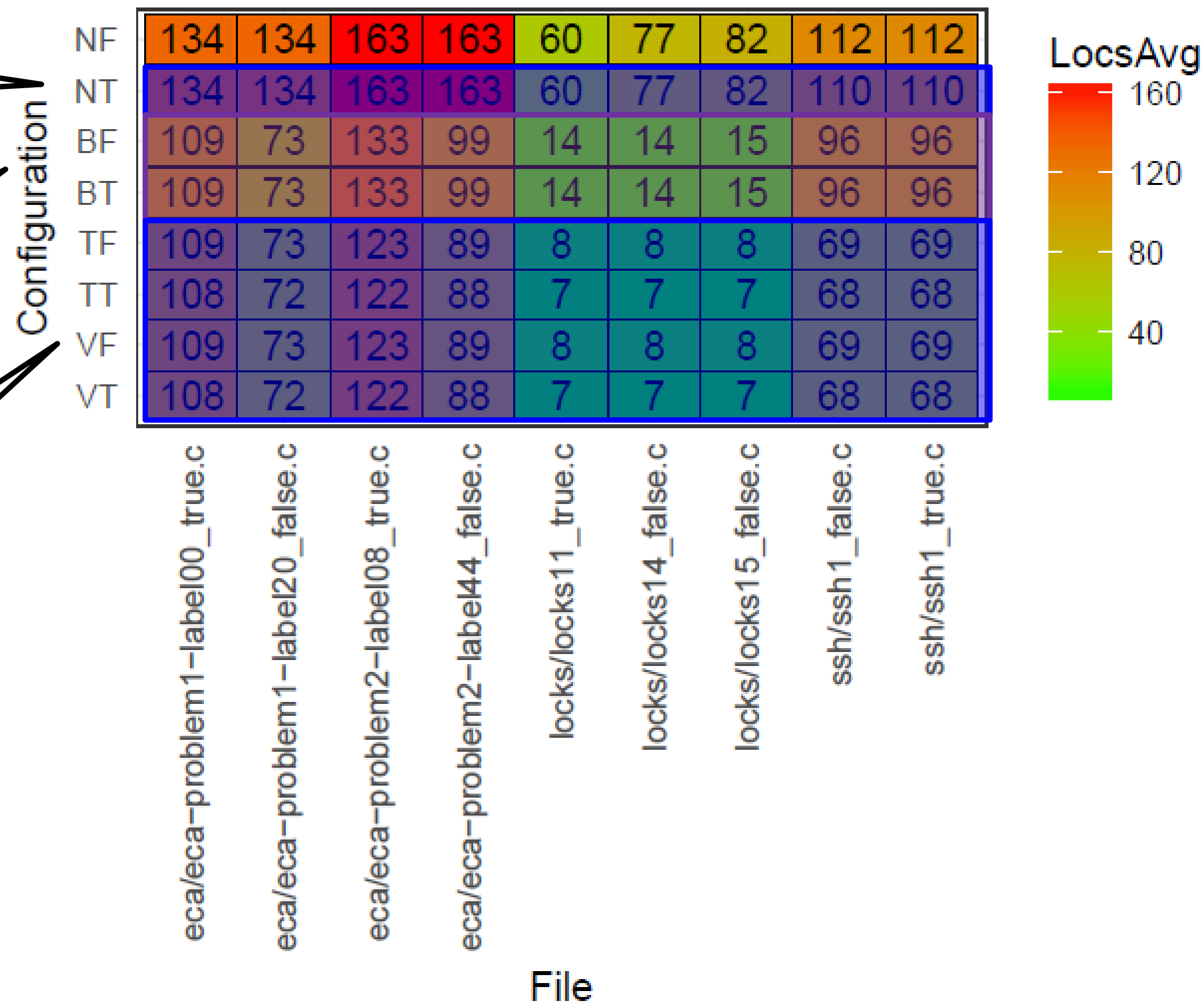
Results

- Initial CFA size with different slicing / optimization configurations

Optimizations do not give large reductions

Backward slicing may yield large reductions

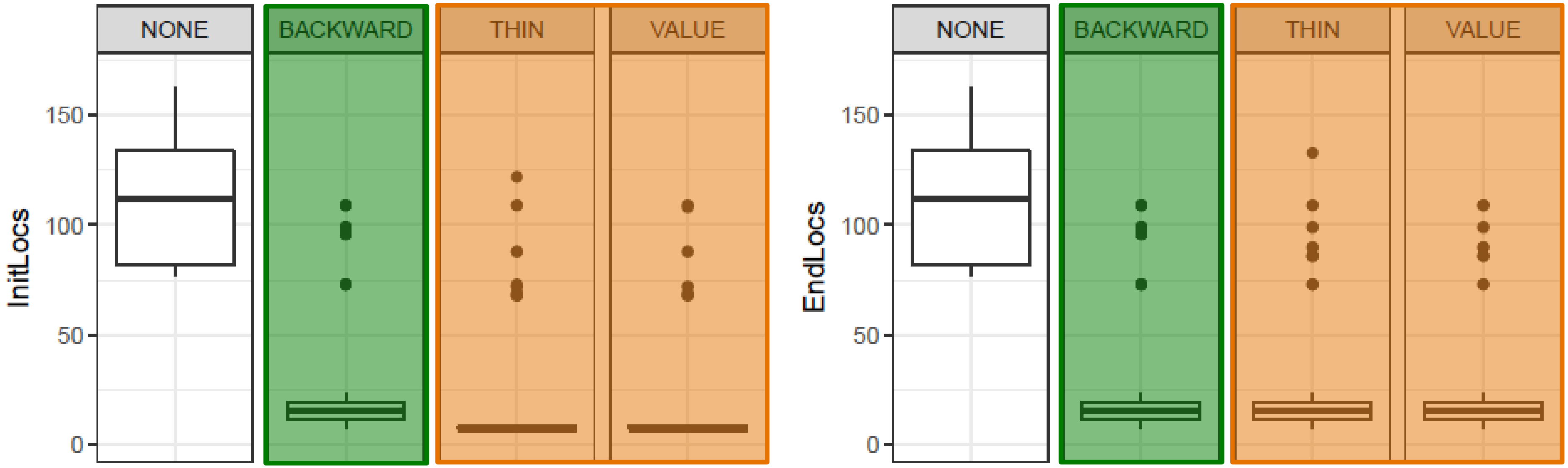
Thin and value slicing allow even more reductions



Results

- Effect of slice refinement: initial and final CFA size

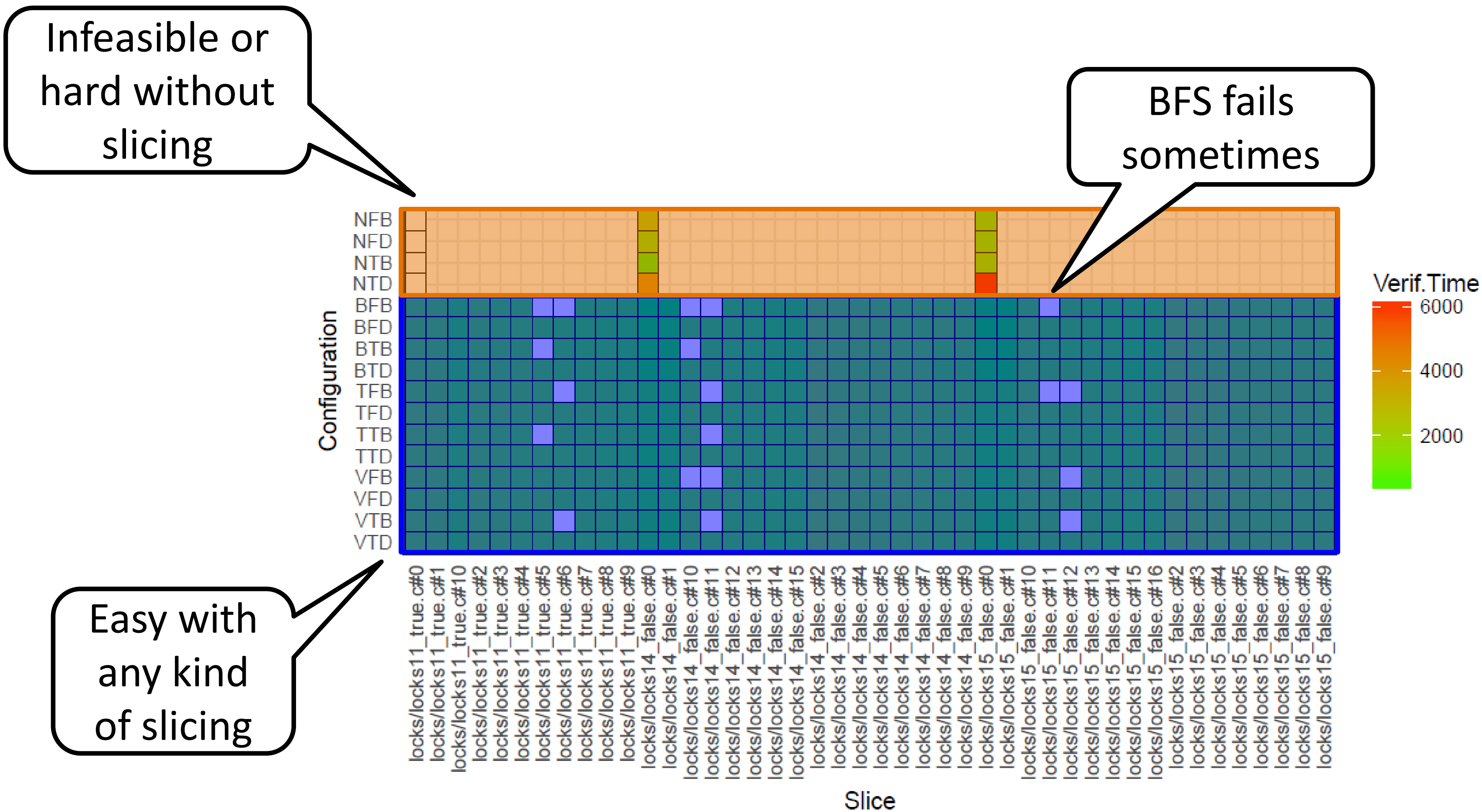
No refinement is needed



Final CFA size increases due to refinements

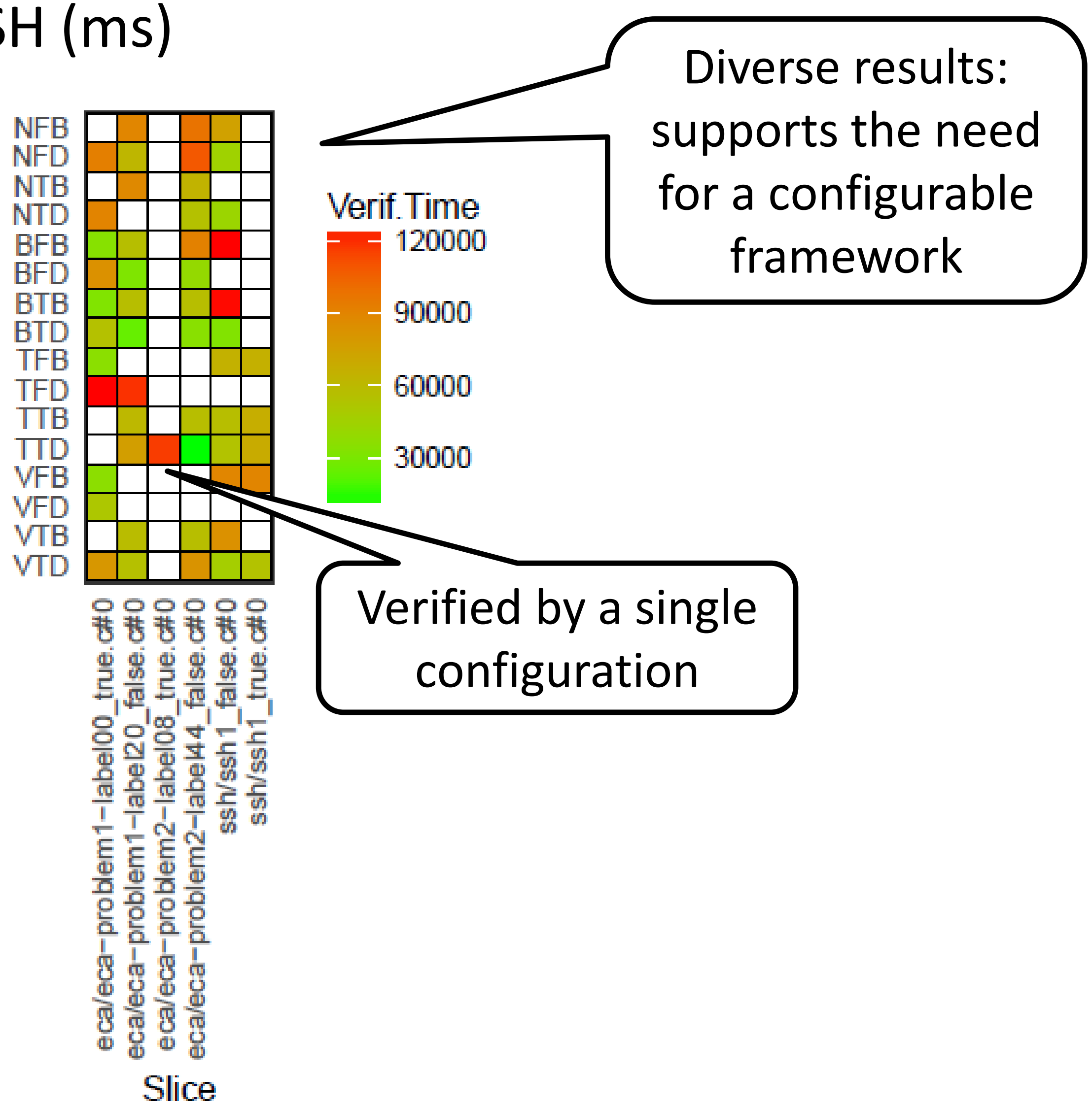
Results

- Verification time – locks (ms)



Results

- Verification time – ECA/SSH (ms)

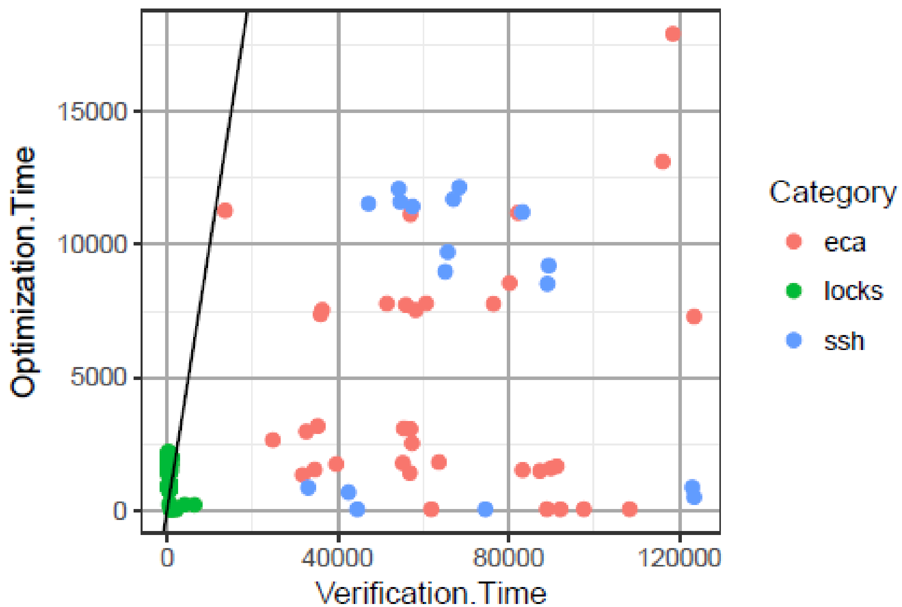


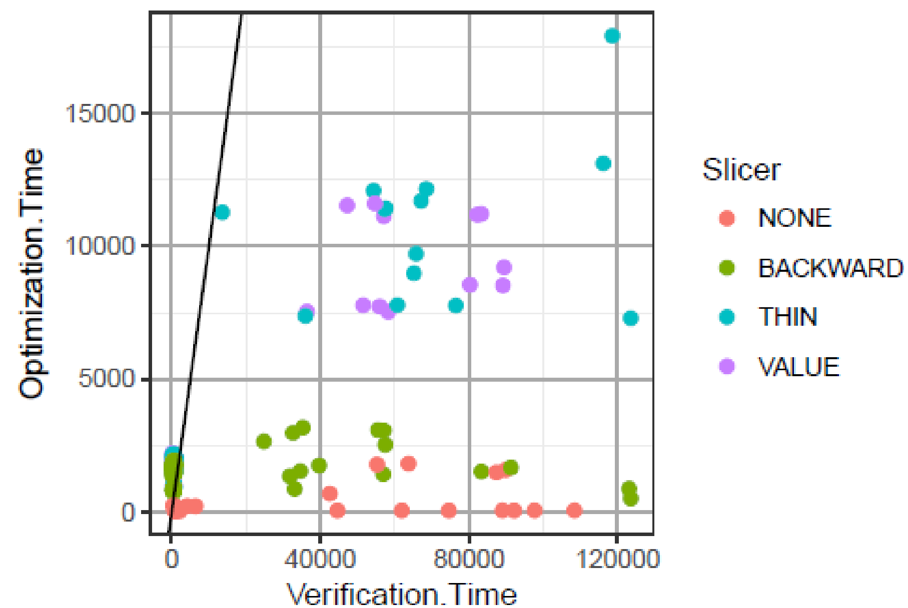
Results

- Comparison of verification and optimization time

Optimization time is negligible for larger programs

Backward slicing is quick, thin and value requires more time

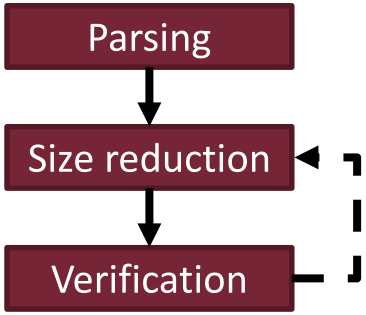




Conclusions

Conclusions

- Workflow for software verification
  - Enhanced by size reduction techniques
  - Supported by a configurable verification framework



- Experimental evaluation
  - Different configurations are more suitable for different tasks

```
0: int i = 0;
1: int x = 0;
2: while (i < 11) {
3:     x = x + i;
4:     i = i + 1;
   }
5: assert(i != 0);
```

- Future work
  - Extend supported elements of C
  - Interprocedural slicing
  - LLVM support

Configuration

NF	134	134	163	163	60	77	82	112	112
NT	134	134	163	163	60	77	82	110	110
BF	109	73	133	99	14	14	15	96	96
BT	109	73	133	99	14	14	15	96	96
TF	109	73	123	89	8	8	8	69	69
TT	108	72	122	88	7	7	7	68	68
VF	109	73	123	89	8	8	8	69	69
VT	108	72	122	88	7	7	7	68	68

hajdua@mit.bme.hu
inf.mit.bme.hu/en/members/hajdua