DEPARTMENT OF INFORMATICS

LMU MÜNCHEN

Bachelor's Thesis in Informatics

# Converting Test Goals to Condition Automata

Frederic Schönberger

DEPARTMENT OF INFORMATICS

LMU MÜNCHEN

Bachelor's Thesis in Informatics

# Converting Test Goals to Condition Automata

# Konvertierung von Testgoals zu Conditions

| | |
|---|---|
| Author: | Frederic Schönberger |
| Supervisor: | Prof. Dr. Dirk Beyer |
| Mentor: | Thomas Lemberger |
| Submission Date: | 28. November 2020 |

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 28. November 2020 Frederic Schönberger

# Acknowledgments

I wish to express my sincere thanks to Mr. Thomas Lemberger, my mentor, for sharing valuable guidance and stellar support during the writing of this thesis.

I would like to thank the SoSy Lab for letting me use their cluster infrastructure to run my experiments.

I take this opportunity to express gratitude to my family and my friends, especially Mr. Matthias Kettl, for their help and support.

# Abstract

Testing is an important part of the software development process. Unfortunately it is expensive to test. That's why one can employ test generators, software that automatically generates test cases. Each tester has different strengths that we can combine using conditional testing. This way we can achieve higher branch coverage.

Conditional testing works by letting a tester run for a certain amount of time and then applying a reducer, a program that removes all paths from the program that are covered by the generated test suite. One such reducer is implemented by CondTest. Another reducer already exists in CPAchecker. It requires a condition automaton as input.

We designed two algorithms that can generate such a condition automaton from a list of covered test goals. They work by generating $\top$ assumptions for covered leaf goals and $\bot$ assumptions for uncovered goals. We implemented these algorithms: One naive implementation and another optimized version using propagation.

They were evaluated using the benchmark set of Test-Comp 2020 and its testers. We found that our algorithms can show significant improvements in branch coverage when we run a tester, reduce the program and then run the tester again. A combination of two testers showed no improvements for the considered combinations. We measured the amount of code our algorithms could reduce in terms of cyclomatic complexity and found that there is no correlation to branch coverage. We discovered our algorithms to have an equal performance in terms of branch coverage. We recorded resource usage and found that our algorithms had comparable performance to CondTest and outperformed it in some cases. Both CondTest and our algorithms always outperformed our baseline.

# Contents

# 1 Introduction

Software testing is an increasingly important part of the modern software development process. Methodologies such as test-driven development, extreme programming and some implementations of SCRUM rely on excessive testing [2, 3]. However, it is expensive to test: Up to 50% of software system development go into testing [21].

A possible solution are *test case generators*. Test case generators are programs that automatically generate test cases. Different approaches to test case generation have different strengths [10]. In order to achieve stronger tools one approach that has come up in recent years is conditional testing. Conditional testing combines various testers: First, we let one tester generate test cases. Then, we remove the paths covered by the generated test cases from the program using a *reducer*. Next, we let a second tester run on the reduced program. That way we can combine the strengths of both testers without the need of information exchange between testers (i.e. not needing to modify exiting testers) [8, 11, 12].

There already are some reducers available: CondTests's reducer and the one implemented in CPAchecker. The first one extracts a list of covered goals as *test goal labels* and uses them to prune the program [12]. The second one expects a condition automaton to reduce the program [11].

In this thesis we present an algorithm that constructs condition automata from a list of covered test goal labels. We then evaluate it against the benchmark set of Test-Comp 2020 using its candidates and compare it with CondTest.

# 2 Related Work

Our approach works by combining testers. There is no information exchange between the testers.

Verifiers can be used together with test generators. CondTest [12] uses formal verifiers to find assertion violations or program locations of interest. Blast [5], FShell [18], CoVeriTest [10] and CPA/Tiger [9] use the verifier for reachability analyses.

There's also the option of combining approaches with information exchange between the individual components. CoVeriTest [10] extracts information from ARGs generated during preceding analysis runs. SYNERGY [17] and DASH [4] alternate generating tests and constructing proofs [10]. SMASH [16] combines under- and over-approximation. Concolic testing [15, 16, 23] generates test inputs by interleaving random testing with symbolic execution. When random testing is stuck, symbolic execution starts at that point. As soon as a new goal is covered random testing starts again and is provided the values used to cover the goal. Badger [29] uses fuzzing with concolic execution in a similar manner. Test suite augmentation [20, 32, 33] can be used to combine one arbitrary tester with another specific tester that reuses information from the first's run.

The Electronic Tools Integration platform (ETI) [24, 31] and the Evidential Tool Bus (ETB) [13, 30] provide further approaches for combination of testers and verifiers.

# 3 Background

## 3.1 Control Flow Automata (CFA)

We can represent programs as *control flow automata* (CFA) [1, 7, 28]. We use program lines as nodes and operations as edges.

**Definition 1** (CFA)**.** A CFA is an automaton $C = (L, l_0, G)$, with a set of locations $L$, an initial location $l_0 \in L$ and a set of transitions $G \subseteq L \times \text{ops} \times L$, where *ops* is the set of operations.

```
1  void main() {
2      int i = nondet_int();
3      int j;
4      if(i < 0) {
5          j = 1;
6      } else {
7          j = 2;
8      }
9
10     assert(j <= 1);
11 }
```

Figure 3.1: An example program.

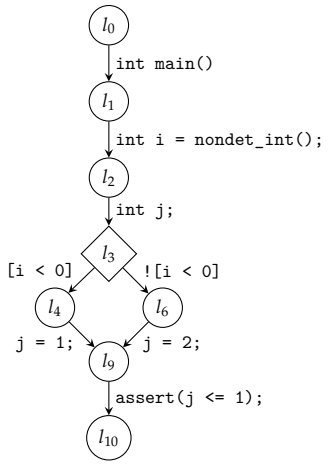



Figure 3.2: The corresponding CFA.

**Definition 2** (Concrete data state)**.** Let $X$ be the set of all variables occurring in *ops*. A concrete data state [11] $c$ is a mapping of each variable in $X$ to its concrete value.

Additionally there is a special variable called *pc*, the program counter, that points to the current position in the program.

**Definition 3** (Concrete program path)**.** A *concrete program path* [11] of a CFA $C = (L, l_0, G)$ is a sequence

$$\pi = (c_0, l_0) \xrightarrow{g_0} \dots \xrightarrow{g_n} (c_n, l_n)$$

such that the concrete data state $c_0$ initializes all variables with zero, $g_i = (l_{i-1}, \mathrm{op}_i, l_i) \in G$ ($g_i$ is a valid transition in the CFA) and $c_{i-1} \xrightarrow{\mathrm{op}_i} c_i$. We define $\mathrm{path}(C)$ as the set of all concrete paths of $C$.

**Definition 4** (Execution)**.** We define an *execution* [11] of a concrete program path $\pi$ as $\mathrm{ex}(\pi) = c_0 c_1 \dots c_n$ with $c_i$ being a concrete data state. We define $\mathrm{ex}(C)$ to be the set of all executions of a CFA $C$.

## 3.2 Condition Automata

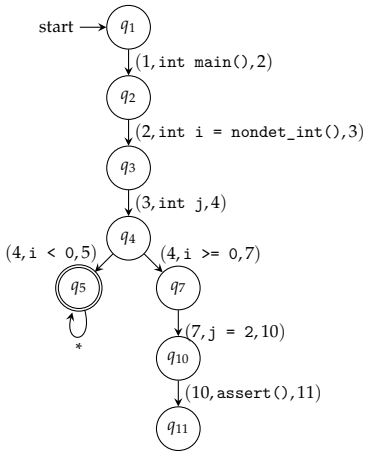



Figure 3.3: An example condition for Figure 3.1.

```
int main() {                              1
    int i = nondet_int();                 2
    int j;                                3
                                          4
    //Notice the different operator       5
    if(i >= 0) {                          6
        j = 2;                            7
    }                                     8
                                          9
    assert(j <= 1);                       10
}                                         11
```

Figure 3.4: The example program reduced using the condition on the left.

Conditions [8] are automata that can describe the paths taken by a verifier. They can additionally introduce *assumptions* that describe under which conditions a path has been explored. Assumptions are conditions on the concrete program state from a set $\Phi$. If a concrete program state $c$ satisfies a state condition $\varphi$ we write $c \vDash \varphi$.

**Definition 5** (Condition)**.** A condition automaton $A = (Q, \Sigma, \delta, q_0, F)$ (short: condition) is an automaton

- with a finite set of states $Q$ and an initial state $q_0 \in Q$,
- an alphabet $\Sigma \subseteq \mathcal{P}(Q) \times \Phi$,
- a transition relation $\delta \subseteq Q \times \Sigma \times Q$
- and a set $F \subseteq Q$ of accepting states.

Additionally it must well formed so that there is no transition such that an accepting state goes into an unaccepting state, i.e.

$$\neg\exists(q_f, \cdot, q) \in \delta \text{ with } q_f \in F \wedge q \notin F.$$

**Definition 6** (Coverage of a Path)**.** A condition $A = (Q, \Sigma, \delta, q_0, F)$ *covers* a path $\pi = (c_0, l_0) \xrightarrow{g_1} \ldots \xrightarrow{g_n} (c_n, l_n)$ if and only if there is a run $\rho = q_0 \xrightarrow{(G_1,\varphi_1)} \ldots \xrightarrow{(G_k,\varphi_k)} q_k$ in $A$ with $0 \leq k \leq n$ such that

1. $q_k$ is an accepting state, i.e. $q_k \in F$,
2. $\forall 1 \leq i \leq k : g_i \in G_i$ (for each $g_i$ such a transition exists in the program) and
3. all concrete program states $c_i$ satisfy the corresponding state condition $\varphi_i$ (i.e. $\forall 1 \leq i \leq k : c_i \vDash \varphi_i$).

## 3.3 Reducers

**Definition 7** (Program Reducer)**.** A *program reducer* [12] $\text{red}_G : P \to P'$ transforms a program $P$ to $P'$ such that for all test vectors $v$ both programs cover the same subset $G_v \subseteq G$ of all goals. We call this property *G-coverage-equivalent*. A program reducer must be both *sound* and *complete*.

*Soundness*. If for a test vector $v$ the program $P'$ covers a goal $g \in G$, then so must $P$.

*Completeness*. If for any test vector $v$ the original program $P$ covers a test goal $g \in G$, then so must $P'$.

**Definition 8** (Reducer)**.** Let $\mathcal{C}$ be the set of all CFAs and $\mathcal{A}$ the set of all Conditions. A *reducer* [11] is a mapping $\mathcal{C} \times \mathcal{A} \to \mathcal{C}$, that takes a CFA and a condition as an input and outputs another CFA that satisfies the *residual condition*

$$\forall C \in \mathcal{C} \forall A \in \mathcal{A} : \text{ex}(C) \setminus \{\text{ex}(\pi) : A \text{ covers } \pi\} \subseteq \text{ex}\left(\text{red}\left(C\right)\right) \subseteq \text{ex}(C),$$

i.e. a CFA with each path of the original program except those that are covered by the condition. One such reducer is implemented in CPAchecker [11].

## 3.4 Code Coverage

When unit testing code there is always the question of how much of the program has been tested ("covered"). There are multiple criterions we can use to measure which parts of our code have been covered [14, 26, 27]:

- *Statement Coverage* measures each statement that has been reached by our tests.
- *Branch Coverage* (Decision Coverage) dictates that each branch in a `if/else`, `switch` or `do-while` statement must be executed at least once. This is equivalent to the condition equating to `true` and `false` at least once.
- *Condition Coverage* requires each part of the condition of an `if` statement to evaluate to `true` and `false` at least once. For instance, let's consider the condition $x \leq 5 \wedge y = 5$. Our test cases would need to account for $x \leq 5$, $\neg(x \leq 5)$, $y = 5$ and $\neg(y = 5)$.
- *Decision-Condition Coverage* combines branch and condition coverage: Sufficient test cases must be written such that each condition in an `if/else` statement takes on all possible outcomes at least once and each `if/else` block is executed at least once.

Please note that this list is not exhaustive and contains only the kinds of coverage that are going to be important later on.

For our approach we are going to use branch coverage since it can be computed easily[1], is granular enough and thus provides a good metric. However there are some known issues with branch coverage: Compilers use techniques such as short circuiting[2] and we can not guarantee that each operation in a condition is executed. Those issues might be solved by using other kinds of coverage criterions. Future work is needed to evaluate if conditions can be used for those as well (see chapter 6).

## 3.5 Instrumentation

To measure branch coverage in our program and identify which branches have been covered we make use of *test goal labels*. For each node in our CFA that we want covered we prepend a no-op (a label `GOAL_n`). This allows us to quickly identify all tets goal

[1] e.g. by running GCOV on an instrumented program.

[2] For instance, see the C programming language standard, where in section 6.5.13.4 it is stated: "*[...] If the first operand compares equal to 0, the second operand is not evaluated.*" [19]

locations in our program. We are using CondTest's instrumentor, which is based on a testability transform `addLabels()` [12].

## 3.6 Cyclomatic Complexity

To measure how the reducer changed our program we can use a variety of software measures. One that is particularly useful is called *Cyclomatic Complexity* [25].

**Definition 9** (Cyclomatic Complexity)**.** Let $C$ be a control-flow graph. Then, the *cyclomatic complexity* $\text{cycl}(C) := E - N + 2P$, where $E$ is the number of edges on the $C$, $N$ the number of nodes and $P$ the number of connected components[3].

McCabe showed that $\text{cycl}(C)$ provides an upper bound for the number of test cases that we need to generate for full branch coverage. Hence, we can measure the work of our reducer by comparing $\text{cycl}(C)$ to $\text{cycl}(\text{red}(C))$.

[3]Connected components are individual groups of vertexes in a graph that are not connected to anything else. In the case of cyclomatic complexity a connected component can be interpreted as a function (or subprogram).

# 4 Automaton-based Conditional Testing

We want to convert a list of test goals to a condition. This condition is then used by a reducer to prune our program. The implementation is based on the CPAchecker framework. It consists of two steps: First we perform a breadth-first search on the CFA to find all leaf test goals. By partitioning them into "covered" and "not-covered" we can then generate a condition that is used by the integrated reducer.

## 4.1 Phase 1: BFS

In this phase we want to identify the leaf test goals in our CFA and categorize them by their coverage state. For that we need a list of covered test goals. Now we perform a breadth-first search to identify the outermost goals. Using the list we divide them into "covered" and "not covered".

**Algorithm 1:** Breadth-first search (Phase 1)

**Input:** CFA $P = (L, \cdot, G)$, a list of covered goals *coveredGoals*
**Result:** A list of test goals, partioned in covered/ not covered

```
1  waitList = {l ∈ L | ∀l' ∈ L : (l, ·, l') ∉ G} ;
2  visitedNodes = ∅ ;
3  leafGoals = ∅;
4  while waitlist ≠ ∅ do
5  |   pop l_i from waitlist;
6  |   visitedNodes = visitedNodes∪{l_i};
7  |   if l_i is a test goal label then
8  |   |   leafGoals = leafGoals∪{l_i}
9  |   else
10 |   |   waitList = waitList ∪{l' : l' ∉ visitedNodes ∧ ∃(l', ·, l) ∈ G};
11 |   end
12 end
13 return (leafGoals ∩ coveredGoals, leafGoals \ coveredGoals);
```

Our implementation identifies goals as labels. For that we use CondTests instrumentor. The program is instrumented using the `addLabels()` transformation [12].

In algorithm 1 we take a CFA and a list of covered goals as input. Our wait list is a queue that contains all elements from our CFA that don't have a transition to another state (line 1). We then iterate over this list until it is empty (line 4). We save all visited elements to avoid visiting one twice (line 5). If our element is a test goal we then add it to a list of found test goals (lines 7 and 8). Otherwise we add the parents (all elements for which there is a transition to this particular element) to our wait list (line 10). The algorithm then returns a tuple of all goals that are covered (the intersection of our leaf goals with our covered goals) and those that are not (our leaf goals without the ones in the "covered" list).

```
1  int main() {
2      if (nondet_int()) {
3          GOAL_0:; // covered
4          if (nondet_int()) {
5              GOAL_1:; // covered
6          } else {
7              GOAL_2:; // covered
8          }
9      } else {
10         GOAL_3:; // covered
11         if (nondet_int()) {
12             GOAL_4:; // covered
13         } else {
14             GOAL_5:; // not covered
15         }
16     }
17
18     return 0;
19 }
```

Figure 4.1: An example C program instrumented with goal labels.

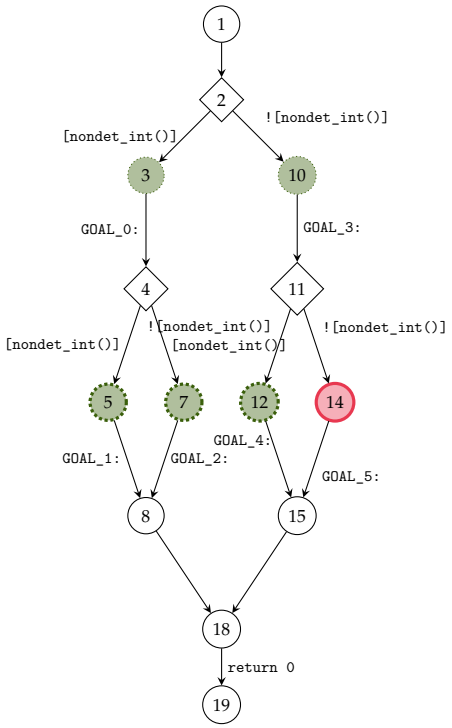



Figure 4.2: The corresponding CFA with test goals highlighted.

Let's quickly take a look at the example from Figure 4.1. Let's also say we ran a tester that covered all goals but `GOAL_5`. We can then run our algorithm. In Figure 4.2 we can see the corresponding CFA. Covered goals are colored in green with a dotted border,

uncovered goals are colored in red with a solid border. Leaf goals have a thick border. Node 19 is the only node without children so we put it in our wait list. It is not a test goal label, thus we continue our search by adding its parent to our queue. Node 18 is not a test goal label either, the same is true for node 15 and node 8. Hence we search their parents next. Now, our queue contains nodes 14, 12, 7 and 5. We identify labels `GOAL_1`, `GOAL_2` and `GOAL_4` as covered and `GOAL_5` as not covered. Since all our nodes are test goal labels we don't need to search any further and stop our search here.

## 4.2 Phase 2: Pruning the CFA

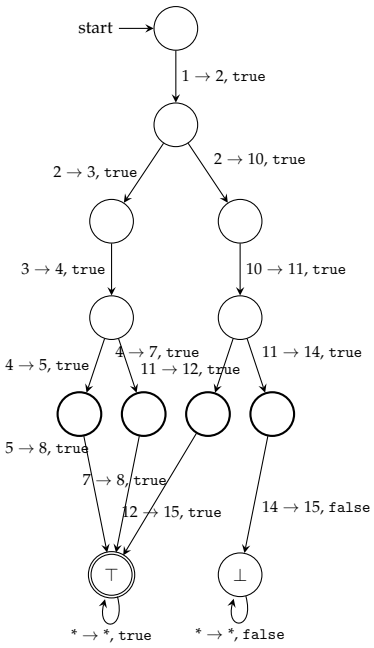



Figure 4.3: The generated condition.

```
int main() {                                1
    if (nondet_int()) {                     2
        GOAL_0:; // covered                 3
    } else {                                4
        GOAL_3:; // covered                 5
        if (!nondet_int()) {                6
            GOAL_5:; // not covered         7
        }                                   8
    }                                       9
                                            10
    return 0;                               11
}                                           12
```

Figure 4.4: The pruned program.

Using the list of leaf goals from phase 1 we can now construct our condition as follows: For each covered leaf goal we generate a `true` ($\top$) assumption. For all other uncovered *goals* (which are not necessarily leaf goals) the assumption is `false` ($\bot$). The latter requirement is necessary to avoid issues with non-linear program flows.

Continuing our example from the previous section we generated the condition in Figure 4.3. We can see that each transition has a `true` assumption by default. As soon as we encounter a leaf goal (highlighted by thick border) we look up whether it is

covered or not. In the first case we generate a final `false` assumption, otherwise a `true` assumption. In Figure 4.4 we can see a our original program that was pruned using this condition.

**Proposition**. We propose that we now can safely prune each leaf goal that has been covered. For that we need to show both *Soundness* and *Completeness* (i.e. $G$-coverage equivalency) of CPAchecker's reducer using our condition for the purpose of Definition 7 [12].

*Proof.* Let $C = (L, \cdot, \cdot)$ be the CFA of our original program and $C' = \operatorname{red} C$. Let $G \subseteq L$ be the set of all uncovered goals.

**Soundness** We know that CPAcheckers reducer algorithm is a reducer (i.e. it satisfies Definition 8) [11]. From $\operatorname{ex} \operatorname{red} C \subseteq \operatorname{ex} C$ follows soundness for any condition.

**Completeness** We can observe that by construction $\forall g \in G : A$ doesn't cover any paths that include $g$. Hence $\operatorname{ex} C \setminus \{\operatorname{ex} \pi : A \text{ covers } \pi\}$ contains all elements from $G$. Thus completeness follows from $\operatorname{ex} C \setminus \{\operatorname{ex} \pi : A \text{ covers } \pi\} \subseteq \operatorname{ex} \operatorname{red} C$. □

## 4.3 Possible optimizations

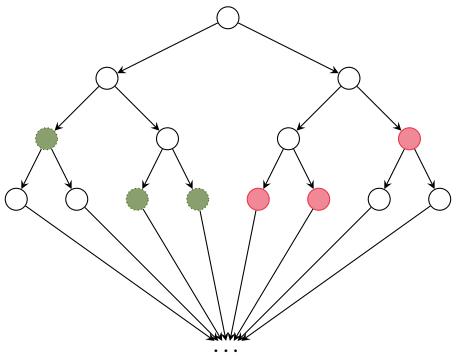


Figure 4.5: A CFA with many branches and its test goals highlighted.

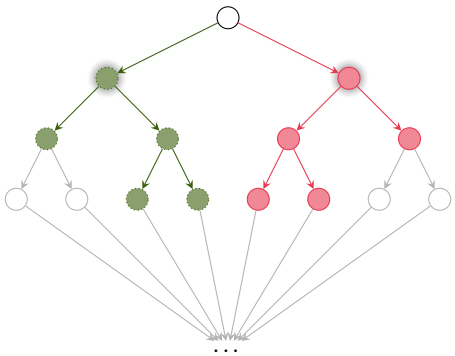


Figure 4.6: The same CFA, after applying our optimization.

A possible optimization technique for our approach consists of merging sibling nodes that are both covered. Illustrated in Figure 4.6 we see that by recursively merging nodes we can generate a much more optimal condition (highlighted by a drop shadow).

We implement our optimization by extending algorithm 1. We start a breadth-first search from the bottom of the CFA. All initial nodes are marked as "virgin". Later, when goal labels are discovered, we use the list of covered goals to mark our respective nodes as "covered" or "uncovered".

**Algorithm 2:** Breadth-first search with propagation (Phase 1)

**Input:** CFA $P = (L, \cdot, G)$, a list of covered goals *coveredGoals*
**Result:** A list of test goals, partioned in covered/ not covered

1 *waitList* = $\{l \in L : \forall l' \in L : (l, \cdot, l') \notin G\}$;
2 *visitedNodes* = $\emptyset$, *nodes* = $\emptyset$, *removableNodes* = $\emptyset$;
3 **while** *waitlist* $\neq \emptyset$ **do**
4 pop $l_i$ from *waitlist*;
5 *visitedNodes* = *visitedNodes*$\cup\{l_i\}$;
6 *children* = $\{l \in L : \exists(l_i, \cdot, l) \in G\}$;
7 **if** $l_i$ *is a test goal label and all children are* `virgin` **then**
8 **if** $l_i \in$ *coveredGoals* **then**
9 *nodes* = *nodes*$\cup\{(\texttt{covered}, l_i)\}$;
10 **else**
11 *nodes* = *nodes*$\cup\{(\texttt{uncovered}, l_i)\}$;
12 **end**
13 **else if** *all children are* `virgin` *or node has no children* **then**
14 *nodes* = *nodes*$\cup\{(\texttt{virgin}, l_i)\}$;
15 **else if** *all children are* `covered` **then**
16 *nodes* = *nodes*$\cup\{(\texttt{covered}, l_i)\}$;
17 **else if** *all children are* `uncovered` **then**
18 *nodes* = *nodes*$\cup\{(\texttt{uncovered}, l_i)\}$;
19 **else**
20 continue;
21 **end**
22 *removableNodes* = *removableNodes* $\cup$ *children*;
23 *waitList* = *waitList* $\cup\{l' : l' \notin$ *visitedNodes* $\wedge \exists(l', \cdot, l) \in G\}$;
24 **end**
25 **return** $(\{l \in L : (\texttt{covered}, l) \in nodes\} \setminus removableNodes, \{l \in L : (\texttt{uncovered}, l) \in nodes\} \setminus removableNodes)$;

Our algorithm keeps track of a map that assigns all elements to their respective state: `virgin` means no (leaf) goals have previously been found. `Covered` and `uncovered` are states that are either leaf goals themselves or whose children all have the same state.

Let's illustrate the difference to the naïve algorithm by once again looking at our example program's CFA representation from Figure 3.2. The steps are listed in Table 4.1. The column *nodes* represents the set `nodes` in algorithm 2. The first element of the tuple is abbreviated as "*V*" for virgin, "*U*" for uncovered and "*C*" for covered. The second element refers to the node's number. We underline the nodes that are *not* in the set `removableNodes`.

Table 4.1: Overview over the steps in our improved algorithm.

| Step | `nodes` |
|---|---|
| 1 | (V, N19) |
| 2 | (V, N19), (V, N18) |
| 3 | (V, N19), (V, N18), (V, N15) |
| 4 | (V, N19), (V, N18), (V, N15), (V, N8) |
| 5 | (V, N19), (V, N18), (V, N15), (V, N8), (U, N14) |
| 6 | (V, N19), (V, N18), (V, N15), (V, N8), (U, N14), (C, N12) |
| 7 | (V, N19), (V, N18), (V, N15), (V, N8), (U, N14), (C, N12), (C, N7) |
| 8 | (V, N19), (V, N18), (V, N15), (V, N8), (U, N14), (C, N12), (C, N7), (C, N5) |
| 9 | (V, N19), (V, N18), (V, N15), (V, N8), (U, N14), (C, N12), (C, N7), (C, N5) |
| 10 | (V, N19), (V, N18), (V, N15), (V, N8), (U, N14), (C, N12), (C, N7), (C, N5), (C, N4) |
| 11 | (V, N19), (V, N18), (V, N15), (V, N8), (U, N14), (C, N12), (C, N7), (C, N5), (C, N4), (C, N3) |
| 12 | (V, N19), (V, N18), (V, N15), (V, N8), (U, N14), (C, N12), (C, N7), (C, N5), (C, N4), (C, N3) |

We start at node 19 because it is the only one without children (Step 1). It is not a test goal label, just like its successor node 18, so both are assigned the `virgin` state (Step 2). The same is true for their successors nodes 15 and 8. We assign them the `virgin` state, too (Steps 3 and 4). We add their successors to our `removableNodes` set. This allows us to only return *leaf nodes* at a later point.

Next, we look at node 14. It is a test goal label that is not covered, hence we assign the `uncovered` state (Step 5). Its sibling node 12 as well as nodes 7 and 5 are all covered and we assign them the corresponding state (Steps 6 to 8).

Next up is Node 11. Its children, nodes 14 and 12 aren't both either covered or uncovered so our exploration stops here (Step 9). Node 4 inherits the `covered` state from its children (Step 10). Node 3 gets its `covered` state from node 4 (Step 11). Lastly, the algorithm stops at node 2 because not all of its children are covered and the wait list is empty (Step 12).

Finally our algorithm returns nodes 3 and 12 as *covered leaf nodes* and node 15 as *uncovered leaf node*. Figure 4.7 and Figure 4.8 illustrate the generated condition and the resulting program.

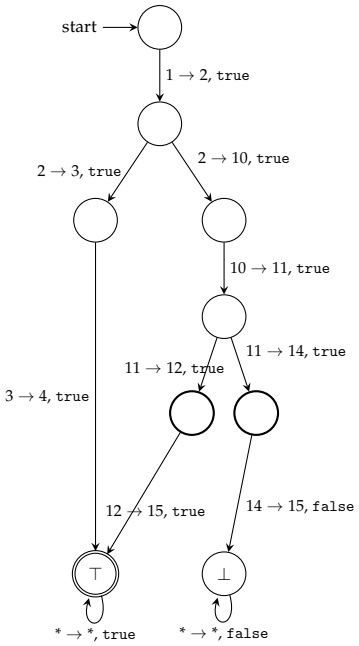



Figure 4.7: The generated condition using our optimized algorithm.

```
int main() {                                1
    if (!nondet_int()) {                    2
        GOAL_3:; // covered                 3
        if (!nondet_int()) {                4
            GOAL_5:; // not covered         5
        }                                   6
    }                                       7
                                            8
    return 0;                               9
}                                           10
```

Figure 4.8: The pruned program using out optimized algorithm.

# 5 Evaluation

We want to check whether conditional testing improves existing solutions. We want to know if our proposed solution improves *branch coverage*. Furthermore we look at the *resource consumption* (i.e. CPU time and memory). Lastly we compare reduction algorithms in regard to *cyclomatic complexity*.

## 5.1 Setup

We performed our experiments on a cluster of 168 with Intel Xeon E5–1320 v5 processors running at 3.40 GHz. The servers are running Ubuntu 20.04.1 LTS with Linux kernel 5.4.0–52–generic.

### 5.1.1 Software

We use CondTest (v3.1-dev, git commit 4f0005fb[1]) for instrumentation, pruning and test goal extraction. For running a sequence of testers and reducers we employ CoVeriTeam (v0.5, git commit 126a1bf8[2]). Branch coverage is measured by TestCov (v3.1-dev, git commit 74977001[3]). Cyclomatic Complexity is measured by pmccabe (installed from the Ubuntu 18.04 package repositories[4]).

### 5.1.2 Benchmarks

We used the candidates from Test-Comp 2020 [6]. They are state-of-the-art testers and their license allows for free replication and evaluation and does not pose any

[1] `https://gitlab.com/sosy-lab/software/conditional-testing/-/tree/4f0005fb`

[2] `https://gitlab.com/sosy-lab/software/coveriteam/-/tree/126a1bf8`

[3] `https://gitlab.com/sosy-lab/software/test-suite-validator/-/tree/74977001`

[4] `https://packages.ubuntu.com/de/bionic/pmccabe`. Note: It is not a mistake that we installed the Ubuntu 18.04 Version while the servers were running Ubuntu 20.04. Some cheap calculations were run locally. For more information see subsequent sections.

Table 5.1: All testers used in the evaluation.

| Name | Version | Repository | Git Tag/ Commit |
|---|---|---|---|
| CoVeriTest | CPAchecker 1.8-svn-3223 | Test-Comp Archives 2020 | `testcomp20` |
| HybridTiger | CPAchecker 1.8-svn-32283M | Test-Comp Archives 2020 | `testcomp20` |
| Klee | KLEE 2.1-pre-test-comp | Test-Comp Archives 2020 | `testcomp20` |
| PRTest | 2.1 | PRTest | `171b066` |
| Symbiotic | 7.0.0-dev | Test-Comp Archives 2020 | `testcomp20` |
| TracerX | Klee v1.2.0 | Test-Comp Archives 2020 | `testcomp20` |

restrictions on their outputs. We had to upgrade to a newer PRTest version due to incompatibilities with our environment. Table 5.1 gives an overview over the testers used[5]. There was an issue with Legion writing its test suites to a different output folder than was expected that forced us to exclude the tool from our benchmarks. We couldn't get LibKluzzer or VeriFuzz to run, that's why we didn't record any results for them.

Our testers ran on the tasks from the *cover branches* category of Test-Comp 2020. These benchmarks were chosen because it is a large benchmark set and they cover a wide range of real-life problems.


Figure 5.1: Evaluation setup for pairs of testers.

We measured four different setups:

- To get our baseline we let a tester run for 15 minutes. We then calculated the branch coverage.
- Three instances of the setup in Figure 5.1. First we instrument our program with test goal labels. Then we execute tester 1 with a time limit of seven minutes. Then we extract information about which goals are covered, and execute a pruner. Lastly we let tester 2 run on the reduced program with a time limit of 8 minutes. These results are used to calculate the cyclomatic complexity locally and join the test suites of testers 1 and 2 together. Branch coverage of the joined test suite is

[5]The repository *Test-Comp Archives 2020* can be found in `https://gitlab.com/sosy-lab/test-comp/archives-2020`, PRTest can be found in `https://gitlab.com/sosy-lab/software/prtest`.

then calculated on the cluster. We used CondTest's pruner, our naïve algorithm and the optimized algorithm with propagation as pruners. The whole sequence had a time limit of 20 minutes.

Our local analysis uses the output files of CoVeriTeam, mainly `execution_trace.xml`. From this file we extract:

- The location of the test suites testers 1 and 2 generated,
- Measurements about the run of testers 1 and 2, like used memory and CPU time,
- The location of the instrumented and reduced program.

We join both test suites together to upload them in the cloud for branch coverage calculation. The measurements and the cyclomatic complexity of both the instrumented (i.e. original) and the reducer program are written to a CSV file. If for some reason no `execution_trace.xml` is produced, our local analysis is not able to join the test suites and this particular run is discarded.

## 5.2 Results

### 5.2.1 One Tester

We tried executing testers sequentially one after another (i.e. we used the same tester for tester 1 and 2). This approach was described by Beyer et. al. [12] as "$\text{tester}_{\text{cycl}}$" (with one repetition).

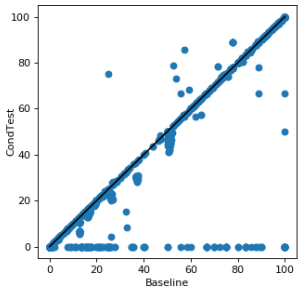
#### 5.2.1.1 CoVeriTest

Table 5.2: Status Codes for CoVeriTest. Similar status codes have been merged.

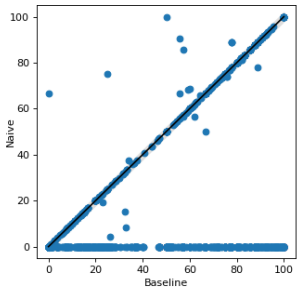
| Status | Baseline | CondTest | Naïve Algorithm | Propagation |
|---|---|---|---|---|
| Done | 793 | 2105 | 683 | 683 |
| Error | 50 | 144 | 1743 | 1747 |
| Out Of Memory | 47 | 43 | 43 | 44 |
| Timeout | 1641 | 239 | 62 | 57 |

We recorded the status codes of our testers in Table 5.2. Both our naïve algorithm and our optimized algorithm produced a lot of `Error` states. This has to do with the issues described in subsection 5.3.3; CoVeriTest did in some cases not produce a test suite when it should have.

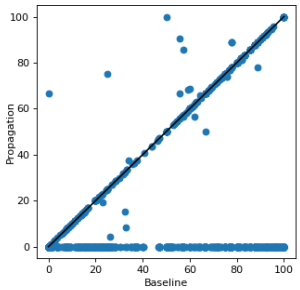
Analyzing the results from Figure 5.2 presents a mixed picture. CondTest in Figure 5.2a saw some improvements and some declines in branch coverage compared to just running CoVeriTest. There are some dips at around 50%. The fact that these can't be found in Figure 5.2b or Figure 5.2c is an artifact of the issue with CoVeriTeam not running properly in some cases. Generally speaking it does seem like our approach produces test suites with a higher branch coverage than CondTest or our baseline. We can also see that there is virtually no difference between our optimized algorithm and the naïve approach.



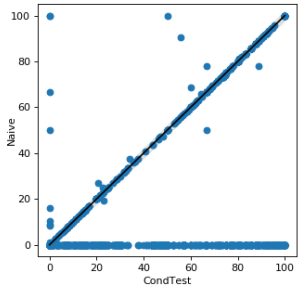

(a) Baseline/ CondTest



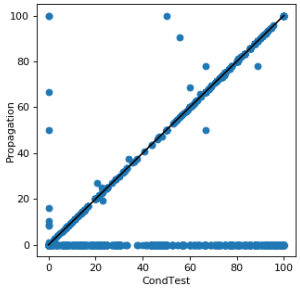

(b) Baseline/ Naïve



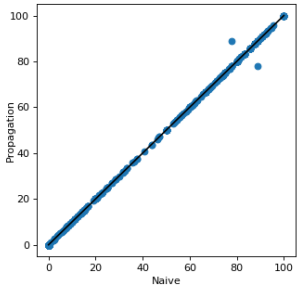

(c) Baseline/ Propagation




(d) CondTest/ Naïve




(e) CondTest/ Propagation




(f) Naïve/ Propagation

Figure 5.2: Branch Coverage (in %) for CoVeriTest.

Let's look at the outliers (i.e. tasks where $|\text{CondTest} - \text{naïve}| > 2$)[6]. This limit is somewhat arbitrary but works in this case. Table 5.3 and Table 5.4 give an overview. We can compare the cyclomatic complexity of each row to see how each pruner fared.

In some cases our reducer did not reduce the program but instead made it more complex: `randesum20.c`'s and `rangesum60.c`'s cyclomatic complexity increased by 36.36%. This however did not result in a worse branch coverage. Our naïve approach had significantly more branch coverage than our baseline. Somewhat surprisingly CondTest's pruner does not change the cyclomatic complexity. In fact this is not only true for this sample but for the complete data set. We made sure to check our set-up for any mistakes even though the fact that it was used for our other algorithms where differing values for the cyclomatic complexity were calculated makes an error unlikely. We can see that in both cases reduced complexity does not necessarily influence the branch coverage. This is not surprising: If a tester fails to generate a test case for a particular branch because of issues like non-linear math it wouldn't help if the rest of the program is reduced; the obstacle is still there.

There is an overlap between the two outlier sets: 8 out of 10 tasks from the CondTest table are also in our baseline table. The two exceptions are `cs_fib_longer-1.c` and `cs_queue-1.c`.

In Figure 5.2f we can see that there is virtually no difference between our naive algorithm and the optimized approach. The two outliers are `openbsd_cmemchr-alloca-1.yml` (77.78% and 88.89% coverage respectively) and `openbsd_cmemrchr-alloca-1.c` (88.89% / 77.78% coverage). Both results do not change the general picture, that's why we're not analyzing the outliers of Baseline/ Propagation and CondTest/ Propagation.

[6] We ignore runs where either is equal to zero.

Table 5.3: Outliers for CoVeriTest (Baseline/ Naïve Algorithm). Correlation Coefficient $r = 0.3174$. Threshold = 2.

| | Branch Coverage | | | Cyclomatic Complexity | | | |
|---|---|---|---|---|---|---|---|
| Task | CondTest | Naïve | Δ | Initial | Naïve | Δ | % red. |
| termination-memory-alloca/b.16-alloca.c | 50.00% | 100.00% | -50.00 | 6 | 5 | 1 | 16.67% |
| termination-memory-alloca/c.01_assume-alloca.c | 25.00% | 75.00% | -50.00 | 8 | 2 | 6 | 75.00% |
| bitvector/s3_clnt_2.BV.c.cil-2a.c | 55.80% | 90.58% | -34.78 | 93 | 87 | 6 | 6.45% |
| termination-memory-alloca/openbsd_cstrcmp-alloca-2.c | 57.14% | 85.71% | -28.57 | 8 | 3 | 5 | 62.50% |
| reducercommutativity/rangesum20.c | 77.78% | 88.89% | -11.11 | 11 | 15 | -4 | -36.36% |
| reducercommutativity/rangesum60.c | 77.78% | 88.89% | -11.11 | 11 | 15 | -4 | -36.36% |
| list-ext-properties/list-ext_flag.c | 55.56% | 66.67% | -11.11 | 19 | 2 | 17 | 89.47% |
| termination-memory-alloca/openbsd_cmemrchr-alloca-1.c | 77.78% | 88.89% | -11.11 | 20 | 2 | 18 | 90.00% |
| termination-libowfat/strtoull.c | 59.09% | 68.18% | -9.09 | 26 | 20 | 6 | 23.08% |
| bitvector/s3_clnt_2.BV.c.cil-1a.c | 60.14% | 68.84% | -8.70 | 93 | 87 | 6 | 6.45% |
| ssh-simplified/s3_clnt_3.cil-1.c | 34.06% | 37.68% | -3.62 | 92 | 86 | 6 | 6.52% |
| busybox-1.22.0/hostid.c | 23.08% | 19.23% | 3.85 | 39 | 2 | 37 | 94.87% |
| bitvector/s3_clnt_3.BV.c.cil-1a.c | 62.14% | 56.43% | 5.71 | 93 | 3 | 90 | 96.77% |
| loop-industry-pattern/mod3.c.v+cfa-reducer.c | 88.89% | 77.78% | 11.11 | 8 | 7 | 1 | 12.50% |
| termination-libowfat/atoll.c | 66.67% | 50.00% | 16.67 | 13 | 8 | 5 | 38.46% |
| busybox-1.22.0/dirname-1.c | 32.61% | 15.22% | 17.39 | 55 | 2 | 53 | 96.36% |
| busybox-1.22.0/basename-1.c | 26.03% | 4.11% | 21.92 | 74 | 2 | 72 | 97.30% |
| busybox-1.22.0/basename-2.c | 32.88% | 8.22% | 24.66 | 74 | 2 | 72 | 97.30% |

Table 5.4: Outliers for CoVeriTest (CondTest/ Naïve Algorithm). Correlation Coefficient $r = 0.3861$. Threshold = 2.

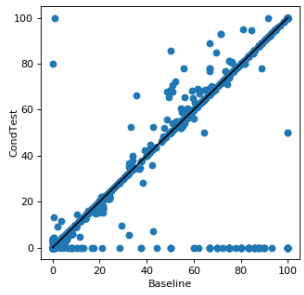
| Task | Branch Coverage | | | Cyclomatic Complexity | | | | |
|---|---|---|---|---|---|---|---|---|
| | CondTest | Naïve | Δ | Initial | CondTest | Naïve | Δ | % red. |
| termination-memory-alloca/b.16-alloca.c | 50.00% | 100.00% | -50.00 | 6 | 6 | 5 | 1 | 16.66% |
| bitvector/s3_clnt_2.BV.c.cil-2a.c | 55.80% | 90.58% | -34.78 | 93 | 93 | 86 | 7 | 7.52% |
| loop-industry-pattern/mod3.c.v+cfa-reducer.c | 66.67% | 77.78% | -11.11 | 8 | 8 | 7 | 1 | 12.50% |
| seq-pthread/cs_fib_longer-1.c | 20.51% | 26.92% | -9.11 | 75 | 75 | 3 | 72 | 96.00% |
| bitvector/s3_clnt_2.BV.c.cil-1a.c | 60.14% | 68.84% | -8.70 | 93 | 93 | 87 | 6 | 6.45% |
| ssh-simplified/s3_clnt_3.cil-1.c | 34.06% | 37.68% | -3.62 | 92 | 92 | 86 | 6 | 93.47% |
| seq-pthread/cs_queue-1.c | 22.46% | 25.13% | -2.67 | 147 | 147 | 5 | 142 | 3.40% |
| busybox-1.22.0/hostid.c | 23.08% | 19.23% | 3.85 | 39 | 39 | 2 | 37 | 94.87% |
| termination-memory-alloca/openbsd_cmemchr-alloca-1.c | 88.89% | 77.78% | 11.11 | 9 | 9 | 4 | 5 | 55.55% |
| termination-libowfat/atoll.c | 66.67% | 50.00% | 16.67 | 13 | 13 | 8 | 5 | 34.46% |

#### 5.2.1.2 HybridTiger

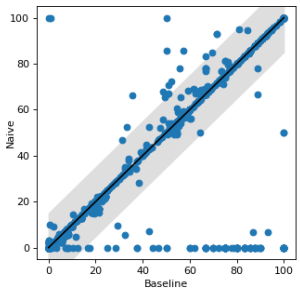
Table 5.5: Status Codes for HybridTiger. Similar states have been merged.

| Status | Baseline | CondTest | Naïve Algorithm | Propagation |
|---|---|---|---|---|
| Done | 807 | 2146 | 2275 | 2272 |
| Error (1) | 9 | 107 | 168 | 175 |
| Out Of Memory | 93 | 76 | 75 | 76 |
| Timeout | 1622 | 202 | 13 | 8 |

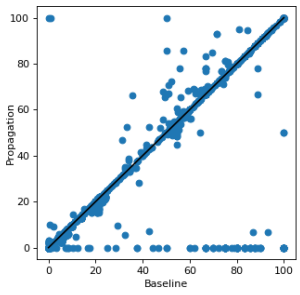
In Table 5.5 we can see HybridTiger's status codes. Our baseline generated a lot of `Timeouts`. This is not a problem, a timeout just forces the algorithm to stop, it is still a valid result.



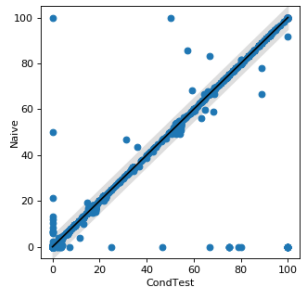

(a) Baseline/ CondTest



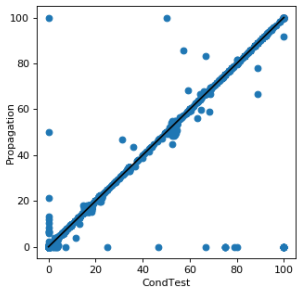

(b) Baseline/ Naïve



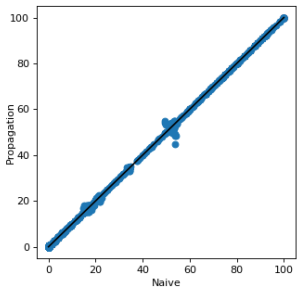

(c) Baseline/ Propagation




(d) CondTest/ Naïve




(e) CondTest/ Propagation




(f) Naïve/ Propagation

Figure 5.3: Branch Coverage (in %) for HybridTiger.

The data in Figure 5.3 shows us that HybridTiger generally benefits from a conditional testing approach. Independent of the reducer most tasks saw a small to medium

improvement in branch coverage. Our improved algorithm and the naïve version were once again virtually identical. Note that there are some tasks where the propagation performed worse than the simple approach. Additionally our optimized version returned more errors than the naive one.

In our data set we can find a lot of outliers if we followed the previous definition. That's why we decided to íncrease our threshold. For this particular dataset we identified outliers for the baseline by $|\text{baseline} - \text{naïve}| > 15$ and for CondTest by $|\text{condtest} - \text{naïve}| > 5$. Again, these values are arbitrary but allow us to find pieces of data where one approach was clearly better or worse than the other without drowning in noise. Table 5.6 and Table 5.7 give an overview.

This time despite the higher threshold we have a lot more outliers. Our algorithm is still better than the baseline in all but 5 of the 27 cases. Our correlation is even weaker than with CoVeriTest. Unlike last time our algorithm did not add more complexity.

When we compare our naive approach to `CondTest` we see that there isn't much overlap between the outliers. Only 5 out of 13 outliers are the same task: loop-industry-pattern/mod3.yml, `termination-15/count_up_and_down_alloca.yml` , `termination-memory-alloca/b.16-alloca.yml`, `loop-invgen/apache-get-tag.i.p+lhb-reducer.yml` and `termination-memory-alloca/gcd1-alloca.yml`.

Table 5.6: Outliers for HybridTiger (Baseline/ Naïve Algorithm). Correlation Coefficient $r = 0.106$. Threshold = 15.

| Task | Branch Coverage | | | Cyclomatic Complexity | | | |
|---|---|---|---|---|---|---|---|
| | CondTest | Naïve | Δ | Initial | Naïve | Δ | % red. |
| ldv-regression/sizeofparameters_test.c | 1.00 | 100.00 | -99.00 | 5 | 1 | 4 | 80.00% |
| termination-memory-alloca/b.16-alloca.c | 50.00 | 100.00 | -50.00 | 6 | 5 | 1 | 16.67% |
| termination-memory-alloca/openbsd_cstrncat-alloca-1.c | 50.00 | 85.71 | -35.71 | 13 | 4 | 9 | 69.23% |
| heap-manipulation/bubble_sort_linux-2.c | 35.59 | 66.10 | -30.51 | 58 | 1 | 57 | 98.28% |
| termination-15/count_up_and_down_alloca.c | 57.14 | 85.71 | -28.57 | 6 | 2 | 4 | 66.67% |
| list-ext-properties/simple-ext.c | 55.56 | 77.78 | -22.22 | 15 | 2 | 13 | 86.67% |
| forester-heap/dll-optional-2.c | 71.43 | 92.86 | -21.43 | 13 | 1 | 12 | 92.31% |
| forester-heap/sll-optional-2.c | 71.43 | 92.86 | -21.43 | 13 | 1 | 12 | 92.31% |
| seq-mthreaded/pals_floodmax.4.1.ufo.BOUNDED-8.pals.c | 52.35 | 72.41 | -20.06 | 262 | 1 | 261 | 99.62% |
| forester-heap/dll-01-1.c | 50.98 | 70.59 | -19.61 | 38 | 1 | 37 | 97.37% |
| seq-mthreaded/pals_floodmax.4.2.ufo.BOUNDED-8.pals.c | 48.64 | 67.98 | -19.34 | 262 | 1 | 261 | 99.62% |
| forester-heap/sll-token-1.c | 33.33 | 52.38 | -19.05 | 17 | 1 | 16 | 94.12% |
| termination-memory-alloca/gcd1-alloca.c | 66.67 | 83.33 | -16.66 | 8 | 8 | 0 | 0.00% |
| seq-mthreaded/pals_floodmax.4.ufo.BOUNDED-8.pals.c | 50.15 | 66.47 | -16.32 | 262 | 1 | 261 | 99.62% |
| seq-mthreaded/pals_floodmax.4.4.ufo.BOUNDED-8.pals.c | 51.06 | 67.07 | -16.01 | 262 | 1 | 261 | 99.62% |
| seq-mthreaded/pals_floodmax.4.3.ufo.BOUNDED-8.pals.c | 49.55 | 65.26 | -15.71 | 262 | 1 | 261 | 99.62% |
| loop-invgen/apache-get-tag.i.p+lhb-reducer.c | 31.41 | 46.79 | -15.38 | 80 | 23 | 57 | 71.25% |
| seq-mthreaded/pals_STARTPALS_ActiveStandby.1.ufo.BOUNDED-10.pals.c | 69.60 | 84.80 | -15.20 | 121 | 1 | 120 | 99.17% |
| heap-manipulation/sll_to_dll_rev-1.c | 29.27 | 9.76 | 19.51 | 44 | 1 | 43 | 97.73% |
| loop-industry-pattern/mod3.c | 88.89 | 66.67 | 22.22 | 9 | 1 | 8 | 88.89% |
| loops/bubble_sort-2.c | 32.39 | 5.63 | 26.76 | 62 | 43 | 19 | 30.65% |
| heap-manipulation/merge_sort-1.c | 42.86 | 7.14 | 35.72 | 51 | 1 | 50 | 98.04% |
| floats-cdfpl/newton_3_6.c | 100.00 | 50.00 | 50.00 | 6 | 1 | 5 | 83.33% |
| floats-cdfpl/sine_1.c | 100.00 | 50.00 | 50.00 | 5 | 1 | 4 | 80.00% |
| loops/string-1.c | 86.67 | 6.67 | 80.00 | 15 | 2 | 13 | 86.67% |
| loops/string-2.c | 93.33 | 6.67 | 86.66 | 15 | 2 | 13 | 86.67% |

Table 5.7: Outliers for HybridTiger (CondTest/ Naïve Algorithm). Correlation Coefficient $r = 0.3693$. Threshold = 5.

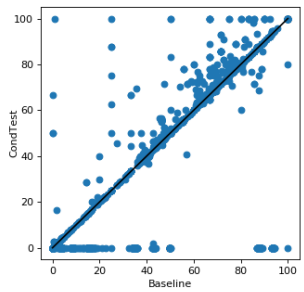
| Task | Branch Coverage | | | Cyclomatic Complexity | | | | |
|---|---|---|---|---|---|---|---|---|
| | CondTest | Naïve | Δ | Initial | CondTest | Naïve | Δ % red. | |
| termination-memory-alloca/b.16-alloca.c | 50 | 100 | -50 | 6 | 6 | 5 | 1 | 16.67% |
| termination-15/count_up_and_down_alloca.c | 57.14 | 85.71 | -28.57 | 6 | 6 | 2 | 4 | 66.67% |
| termination-memory-alloca/gcd1-alloca.c | 66.67 | 83.33 | -16.66 | 8 | 8 | 8 | 0 | 0.00% |
| loop-invgen/apache-get-tag.i.p+lhb-reducer.c | 31.41 | 46.79 | -15.38 | 80 | 80 | 23 | 57 | 71.25% |
| termination-libowfat/strtoul.c | 59.09 | 68.18 | -9.09 | 26 | 26 | 20 | 6 | 23.08% |
| seq-pthread/cs_stack-2.c | 35.92 | 43.66 | -7.74 | 121 | 121 | 1 | 120 | 99.17% |
| seq-mthreaded/pals_opt-floodmax.5.4.ufo.BOUNDED-10.pals.c | 64.85 | 59.74 | 5.11 | 651 | 651 | 1 | 650 | 99.85% |
| seq-mthreaded/pals_opt-floodmax.5.ufo.BOUNDED-10.pals.c | 63.11 | 55.92 | 7.19 | 651 | 651 | 1 | 650 | 99.85% |
| busybox-1.22.0/hostid.c | 11.54 | 3.85 | 7.69 | 39 | 39 | 2 | 37 | 94.87% |
| termination-15/cstrncat_reverse_alloca.c | 100 | 91.67 | 8.33 | 11 | 11 | 4 | 7 | 63.64% |
| termination-libowfat/strtoull.c | 68.18 | 59.09 | 9.09 | 26 | 26 | 20 | 6 | 23.08% |
| loop-industry-pattern/mod3.c.v+cfa-reducer.c | 88.89 | 77.78 | 11.11 | 8 | 8 | 7 | 1 | 12.50% |
| loop-industry-pattern/mod3.c | 88.89 | 66.67 | 22.22 | 9 | 9 | 1 | 8 | 88.89% |

#### 5.2.1.3 Klee

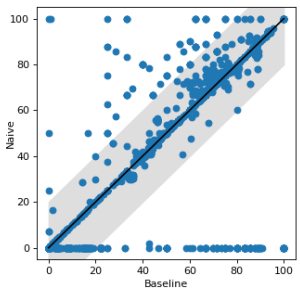
Table 5.8: Status Codes for KLEE.

| Status | Baseline | CONDTEST | Naïve Algorithm | Propagation |
|---|---|---|---|---|
| Done | 457 | 1801 | 1882 | 1881 |
| Error (1) | 156 | 105 | 356 | 356 |
| Out of Memory | 8 | 8 | 6 | 6 |
| Timeout | 1651 | 617 | 287 | 287 |
| Unknown | 167 | | | |

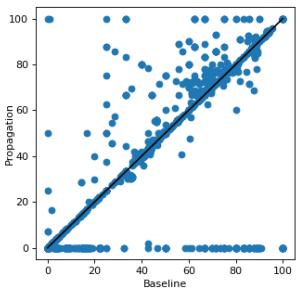
We can see that KLEE alone runs into a lot of timeouts. Conditional testing helps with that and our algorithms allowed even more runs to complete successfully.



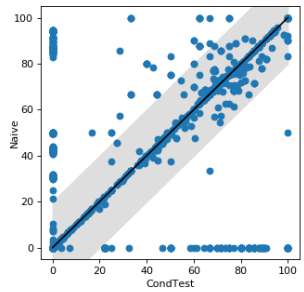

(a) Baseline/ CONDTEST



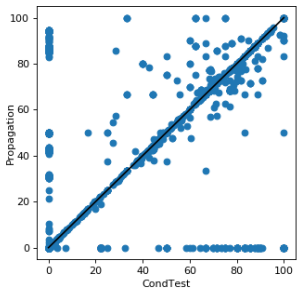

(b) Baseline/ Naïve



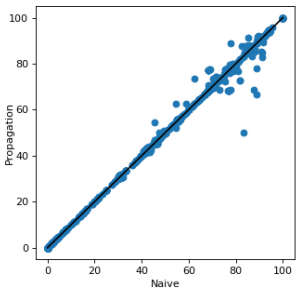

(c) Baseline/ Propagation




(d) CONDTEST/ Naïve




(e) CONDTEST/ Propagation




(f) Naïve/ Propagation

Figure 5.4: Branch Coverage (in %) for KLEE.

We can see that our approach compared to our baseline seems to improve branch coverage, especially where our baseline already has test suites with a high branch

coverage. Our approaches seem to be on par with CondTest. In some cases it is much worse, in others much better. Propagation does not enhance our basic algorithm by a lot and in some case even drastically worsens the results.

Analyzing outliers is not very fruitful. To get a analyzable amount of data points we'd have to increase our threshold to as much as 25 (see the intervals indicated in light gray in Figure 5.4b and Figure 5.4d).

#### 5.2.1.4 PRTest

Table 5.9: Status Codes for PRTest.

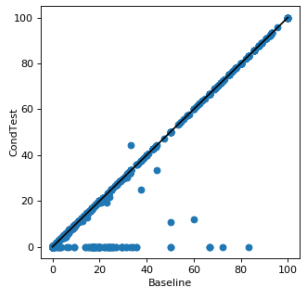
| Status | Baseline | CondTest | Naïve Algorithm | Propagation |
|---|---|---|---|---|
| Done | – | 2184 | 1699 | 1698 |
| Error (1) | 521 | 256 | 688 | 689 |
| Out of Memory | 8 | 7 | 7 | 7 |
| Timeout | 2002 | 82 | 137 | 137 |

Most runs of PRTest in combination with conditional testing did execute successfully, which gives us plenty of data to analyze. Note how the tester itself either runs into a timeout or an error, that's because it never stops testing unless it is killed (i.e. by signal SIGTERM). It does however produce valid test suites for each timeout.

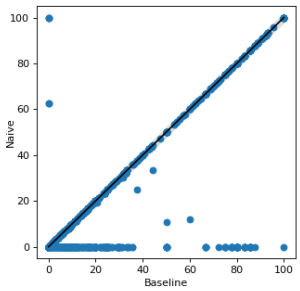
We can see that PRTest does not benefit from conditional testing. Neither CondTest nor our algorithm had any kind of improvement compared to our baseline. Usually the results are about the same, in some rare cases the results are worse. This might be due to the way PRTest works [22]: It executes a program and generates a random value for each input. A reduced program might be smaller and PRTest might be able to create tests more quickly, but the technique doesn't benefit as much from program reduction as other more sophisticated test generators.

Unsurprisingly, we don't see as many outliers[7]: Table 5.10 shows us four outliers. Each tells us that our algorithm performed worse than the baseline. There is a negative correlation, however with that small of a sample size it is mostly meaningless. There Table 5.11 has only one outlier. Here our algorithm performed worse, too.

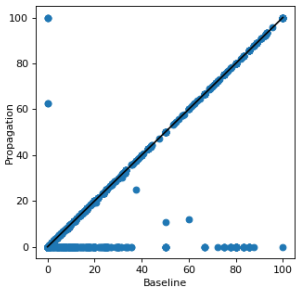
[7] To get our outliers we used a threshold of 2.



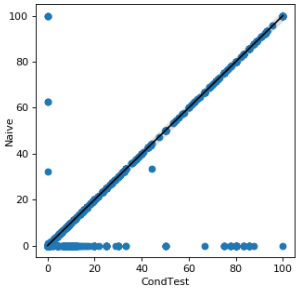

(a) Baseline/ CondTest



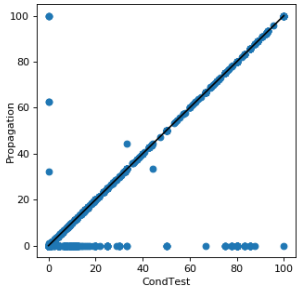

(b) Baseline/ Naïve



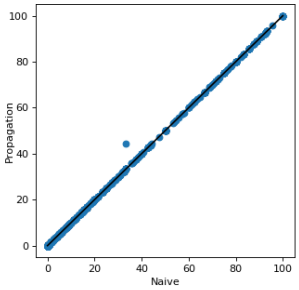

(c) Baseline/ Propagation




(d) CondTest/ Naïve




(e) CondTest/ Propagation




(f) Naïve/ Propagation

Figure 5.5: Branch Coverage (in %) for PRTest.

Table 5.10: Outliers for PRTest (Baseline/ Naïve Algorithm). Correlation Coefficient $r = -0.6969$.

| Task | Branch Coverage | | | Cyclomatic Complexity | | | |
|---|---|---|---|---|---|---|---|
| | CondTest | Naïve | Δ | Initial | Naïve | Δ | % red. |
| loop-industry-pattern/mod3.c.v+cfa-reducer.yml | 44.44 | 33.33 | 11.11 | 8 | 7 | 1 | 12.50% |
| array-industry-pattern/array_monotonic.yml | 37.5 | 25 | 12.5 | 8 | 5 | 3 | 37.50% |
| loop-crafted/simple_array_index_value_4.i.v+lhb-reducer.yml | 50 | 10.71 | 39.29 | 19 | 18 | 1 | 5.26% |
| loop-crafted/simple_array_index_value_4.i.v+nlh-reducer.yml | 60 | 12 | 48 | 16 | 15 | 1 | 6.25% |

Table 5.11: Outliers for PRTest (CondTest/ Naïve Algorithm).

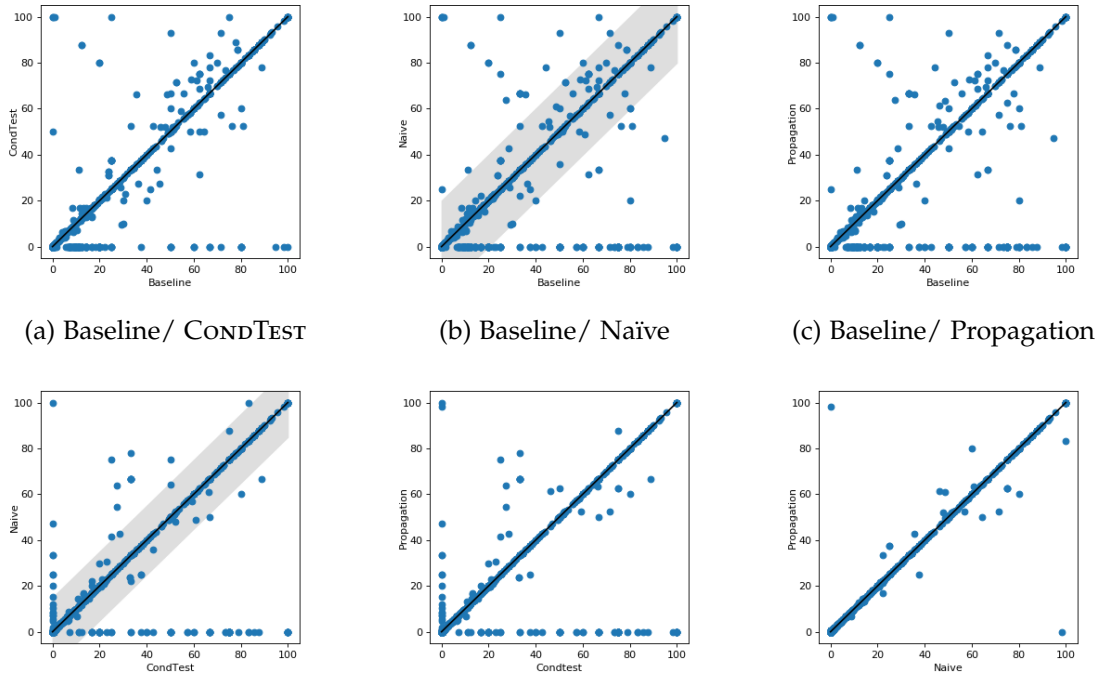
| Task | Branch Coverage | | | Cyclomatic Complexity | | | | |
|---|---|---|---|---|---|---|---|---|
| | CondTest | Naïve | Δ | Initial | CondTest | Naïve | Δ | % red. |
| loop-industry-pattern/mod3.yml | 44.44 | 33.33 | 11.11 | 9 | 9 | 1 | 8 | 88.89% |

#### 5.2.1.5 Symbiotic

Table 5.12: Status Codes for Symbiotic.

| Status | Baseline | CondTest | Naïve Algorithm | Propagation |
|---|---|---|---|---|
| Done | 2488 | 2402 | 2284 | 2283 |
| Error | 112 | 105 | 223 | 224 |
| Out of Memory | 27 | 24 | 24 | 24 |
| Timeout | 9 | – | – | – |

In Table 5.12 we can see Symbiotic's status codes. A lot of tasks completed successfully which allows us to get a differentiated view on the performance of our algorithms.




(a) Baseline/ CondTest (b) Baseline/ Naïve (c) Baseline/ Propagation

(d) CondTest/ Naïve (e) CondTest/ Propagation (f) Naïve/ Propagation

Figure 5.6: Branch Coverage (in %) for Symbiotic.

In Figure 5.6 we can see a mixed picture: In some tasks conditional testing (regardless of the implementation) performed better, in others worse. It's hard to determine outliers

in this case, because we would need a threshold as high as 50 (or even higher) to get a subset of data points that would be actually analyzable. We can however look at the outliers for CondTest/ our naïve algorithm (Table 5.14) using a threshold value of 15.

We can see that a lot of tasks (7 out of 15) didn't reduce the program at all but instead made it more complex. Although they were more complex our algorithm still had a higher branch coverage in all of them.

In Figure 5.6f we can see that this time there is a difference between propagation and non-propagation: For the most part propagation seems to improve our results by some margin, even though there were some significant dips.

Table 5.13: Outliers for Symbiotic (Baseline/ Naïve Algorithm). Correlation Coefficient $r = 0.1894$. Threshold = 20.

| Task | Branch Coverage | | | Cyclomatic Complexity | | | |
|---|---|---|---|---|---|---|---|
| | CondTest | Naïve | Δ | Initial | Naïve | Δ | % red. |
| ldv-regression/sizeofparameters_test.yml | 1.00 | 100.00 | -99.00 | 5 | 1 | 4 | 80.00% |
| array-tiling/rew.yml | 12.50 | 87.50 | -75.00 | 9 | 1 | 8 | 88.89% |
| array-tiling/rewrev.yml | 12.50 | 87.50 | -75.00 | 9 | 1 | 8 | 88.89% |
| termination-memory-alloca/Avery-2006FLOPS-Tabel1_true-alloca.yml | 25.00 | 100.00 | -75.00 | 8 | 1 | 7 | 87.50% |
| ldv-regression/test22-1.yml | 20.00 | 80.00 | -60.00 | 11 | 1 | 10 | 90.91% |
| ldv-regression/test22-2.yml | 20.00 | 80.00 | -60.00 | 11 | 1 | 10 | 90.91% |
| array-tiling/mlceu.yml | 25.00 | 75.00 | -50.00 | 8 | 9 | -1 | -12.50% |
| forester-heap/dll-optional-2.yml | 50.00 | 92.86 | -42.86 | 13 | 1 | 12 | 92.31% |
| termination-memory-alloca/openbsd_cstrncmp-alloca-1.yml | 27.27 | 63.64 | -36.37 | 10 | 12 | -2 | -20.00% |
| loop-industry-pattern/mod3.c.v+cfa-reducer.yml | 44.44 | 77.78 | -33.34 | 8 | 7 | 1 | 12.50% |
| termination-memory-alloca/HarrisLalNoriRajamani-2010SAS-Fig3-alloca.yml | 33.33 | 66.67 | -33.34 | 6 | 2 | 4 | 66.67% |
| termination-memory-alloca/cstrncmp-alloca-2.yml | 33.33 | 66.67 | -33.34 | 12 | 16 | -4 | -33.33% |
| termination-15/cstrncmp_diffterm_alloca.yml | 33.33 | 66.67 | -33.34 | 10 | 16 | -6 | -60.00% |
| termination-15/cstrncmp_mixed_alloca.yml | 33.33 | 66.67 | -33.34 | 10 | 16 | -6 | -60.00% |
| termination-15/cstrncmp_reverse_alloca.yml | 33.33 | 66.67 | -33.34 | 10 | 16 | -6 | -60.00% |
| loops/trex03-1.yml | 66.67 | 100.00 | -33.33 | 11 | 1 | 10 | 90.91% |
| heap-manipulation/bubble_sort_linux-2.yml | 35.59 | 66.10 | -30.51 | 58 | 1 | 57 | 98.28% |
| array-tiling/revcpyswp2.yml | 11.11 | 33.33 | -22.22 | 11 | 1 | 10 | 90.91% |
| forester-heap/sll-optional-2.yml | 71.43 | 92.86 | -21.43 | 13 | 1 | 12 | 92.31% |
| forester-heap/sll-rb-sentinel-2.yml | 76.19 | 52.38 | 23.81 | 16 | 1 | 15 | 93.75% |
| forester-heap/sll-rb-sentinel-1.yml | 80.95 | 52.38 | 28.57 | 16 | 1 | 15 | 93.75% |
| forester-heap/dll-circular-1.yml | 62.50 | 31.25 | 31.25 | 15 | 1 | 14 | 93.33% |
| float-benchs/filter2.yml | 66.67 | 33.33 | 33.34 | 9 | 1 | 8 | 88.89% |
| termination-memory-alloca/Toulouse-BranchesToLoop-alloca.yml | 66.67 | 33.33 | 33.34 | 6 | 1 | 5 | 83.33% |
| termination-memory-alloca/Toulouse-MultiBranchesToLoop-alloca.yml | 94.74 | 47.37 | 47.37 | 14 | 1 | 13 | 92.86% |
| float-benchs/filter2_reinit.yml | 80.00 | 20.00 | 60.00 | 10 | 1 | 9 | 90.00% |

Table 5.14: Outliers for Symbiotic (CondTest/ Naïve Algorithm). Correlation Coefficient $r = 0.5805$. Threshold = 15.

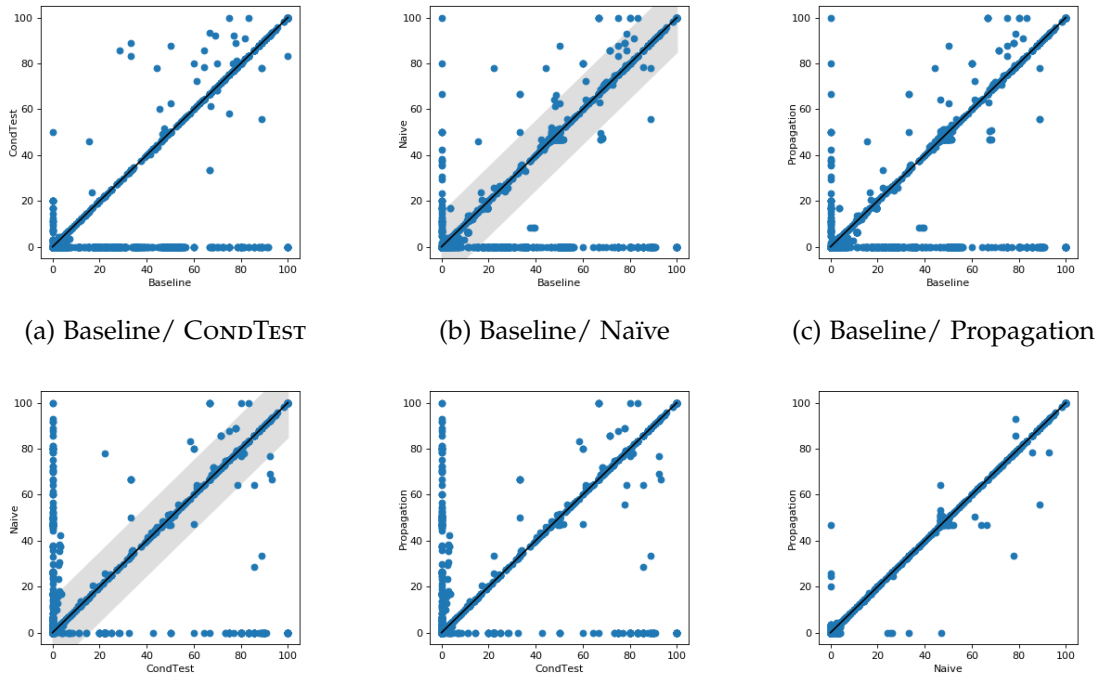
| | Branch Coverage | | | Cyclomatic Complexity | | | | |
|---|---|---|---|---|---|---|---|---|
| Task | CondTest | Naïve | Δ | Initial | CondTest | Naïve | Δ | % red. |
| array-tiling/mlceu.yml | 25.00 | 75.00 | -50.00 | 8 | 8 | 9 | -1 | -12.50% |
| loop-industry-pattern/mod3.c.v+cfa-reducer.yml | 33.33 | 77.78 | -44.45 | 8 | 8 | 7 | 1 | 12.50% |
| termination-memory-alloca/openbsd_cstrncmp-alloca-1.yml | 27.27 | 63.64 | -36.37 | 10 | 10 | 12 | -2 | -20.00% |
| termination-memory-alloca/HarrisLalNoriRajamani-2010SAS-Fig3-alloca.yml | 33.33 | 66.67 | -33.34 | 6 | 6 | 2 | 4 | 66.67% |
| termination-memory-alloca/cstrncmp-alloca-2.yml | 33.33 | 66.67 | -33.34 | 12 | 12 | 16 | -4 | -33.33% |
| termination-15/cstrncmp_diffterm_alloca.yml | 33.33 | 66.67 | -33.34 | 10 | 10 | 16 | -6 | -60.00% |
| termination-15/cstrncmp_mixed_alloca.yml | 33.33 | 66.67 | -33.34 | 10 | 10 | 16 | -6 | -60.00% |
| termination-15/cstrncmp_reverse_alloca.yml | 33.33 | 66.67 | -33.34 | 10 | 10 | 16 | -6 | -60.00% |
| loop-invgen/sendmail-close-angle.yml | 27.27 | 54.55 | -27.28 | 9 | 9 | 14 | -5 | -55.56% |
| termination-15/cstrncpy_reverse_alloca.yml | 50.00 | 75.00 | -25.00 | 7 | 7 | 4 | 3 | 42.86% |
| loops/trex03-1.yml | 83.33 | 100.00 | -16.67 | 11 | 11 | 1 | 10 | 90.91% |
| loop-invgen/nest-if3.yml | 25.00 | 41.67 | -16.67 | 10 | 10 | 8 | 2 | 20.00% |
| loops/trex03-2.yml | 66.67 | 50.00 | 16.67 | 11 | 11 | 1 | 10 | 90.91% |
| termination-15/cstrcmp_diffterm_alloca.yml | 80.00 | 60.00 | 20.00 | 7 | 7 | 6 | 1 | 14.29% |
| loop-lit/ddlm2013.yml | 88.89 | 66.67 | 22.22 | 8 | 8 | 1 | 7 | 87.50% |

#### 5.2.1.6 TracerX

Table 5.15: Status Codes for TracerX.

| Status | Baseline | CondTest | Naïve Algorithm | Propagation |
|---|---|---|---|---|
| Done | 1030 | 1271 | 1426 | 1422 |
| Error | 249 | 174 | 455 | 442 |
| False | 132 | – | – | – |
| Out of Memory | 61 | 52 | 56 | 55 |
| Timeout | 1022 | 1034 | 594 | 612 |
| Unknown | 29 | – | – | – |

TracerX had some successful runs. As many runs resulted in a timeout. As we explained in previous sections this is not bad because it still produces valid test suites and test cases.




(a) Baseline/ CondTest (b) Baseline/ Naïve (c) Baseline/ Propagation

(d) CondTest/ Naïve (e) CondTest/ Propagation (f) Naïve/ Propagation

Figure 5.7: Branch Coverage (in %) for TracerX.

Generally, we can see that TracerX does not profit as much from a conditional testing approach as other testers do. Most of the time conditional testing generated a higher coverage, sometimes it did not. Our algorithms once again were virtually identical, although this time there were bigger differences than with other testers.

When analyzing the outliers we once again see that cyclomatic complexity does not correlate to branch coverage: Table 5.17 shows some outliers where our algorithm made the program as much as six times as complex and still was able to calculate a higher branch coverage than CondTest. Conversely, some programs where our algorithm was able to reduce a lot of the program clearly favored CondTest. Still, the large number of programs with a significantly higher cyclomatic complexity stand out and require further analysis.

Table 5.16: Outliers for TracerX (Baseline/ Naïve Algorithm). Correlation Coefficient $r = 0.1456$. Threshold = 15.

| Task | Branch Coverage | | | Cyclomatic Complexity | | | |
|---|---|---|---|---|---|---|---|
| | CondTest | Naïve | Δ | Initial | Naïve | Δ | % red. |
| array-patterns/array4_pattern.yml | 22.22 | 77.78 | -55.56 | 11 | 3 | 8 | 72.73% |
| array-industry-pattern/array_of_struct_break.yml | 50.00 | 87.50 | -37.50 | 8 | 5 | 3 | 37.50% |
| list-ext-properties/simple-ext.yml | 44.44 | 77.78 | -33.34 | 15 | 2 | 13 | 86.67% |
| loop-acceleration/multivar_1-2.yml | 33.33 | 66.67 | -33.34 | 5 | 22 | -17 | -340.00% |
| termination-15/cstrchr_malloc.yml | 33.33 | 66.67 | -33.34 | 6 | 5 | 1 | 16.67% |
| termination-memory-alloca/b.03-no-inv_assume-alloca.yml | 66.67 | 100.00 | -33.33 | 7 | 2 | 5 | 71.43% |
| termination-memory-alloca/cstrchr-alloca-1.yml | 66.67 | 100.00 | -33.33 | 8 | 5 | 3 | 37.50% |
| termination-15/cstrchr_diffterm_alloca.yml | 66.67 | 100.00 | -33.33 | 6 | 5 | 1 | 16.67% |
| termination-15/cstrchr_reverse_alloca.yml | 66.67 | 100.00 | -33.33 | 6 | 5 | 1 | 16.67% |
| list-ext-properties/list-ext.yml | 15.38 | 46.15 | -30.77 | 20 | 2 | 18 | 90.00% |
| termination-memory-alloca/Avery-2006FLOPS-Tabel1_true-alloca.yml | 75.00 | 100.00 | -25.00 | 8 | 1 | 7 | 87.50% |
| loops/terminator_03-1.yml | 80.00 | 100.00 | -20.00 | 8 | 25 | -17 | -212.50% |
| loops/trex02-1.yml | 60.00 | 80.00 | -20.00 | 7 | 2 | 5 | 71.43% |
| loops/trex02-2.yml | 80.00 | 100.00 | -20.00 | 7 | 2 | 5 | 71.43% |
| loop-acceleration/diamond_1-2.yml | 60.00 | 80.00 | -20.00 | 6 | 3 | 3 | 50.00% |
| termination-memory-alloca/a.09_assume-alloca.yml | 60.00 | 80.00 | -20.00 | 8 | 2 | 6 | 75.00% |
| termination-memory-alloca/b.03_assume-alloca.yml | 60.00 | 80.00 | -20.00 | 8 | 2 | 6 | 75.00% |
| eca-rers2012/Problem10_label52.yml | 48.76 | 66.17 | -17.41 | 954 | 1 | 953 | 99.90% |
| termination-memory-alloca/gcd1-alloca.yml | 33.33 | 50.00 | -16.67 | 8 | 7 | 1 | 12.50% |
| termination-memory-linkedlists/cll_search-alloca-1.yml | 83.33 | 100.00 | -16.67 | 9 | 2 | 7 | 77.78% |
| eca-rers2012/Problem10_label10.yml | 47.76 | 64.18 | -16.42 | 954 | 1 | 953 | 99.90% |
| eca-rers2012/Problem10_label08.yml | 67.66 | 49.75 | 17.91 | 954 | 1 | 953 | 99.90% |
| eca-rers2012/Problem10_label49.yml | 68.16 | 47.76 | 20.40 | 954 | 1 | 953 | 99.90% |
| eca-rers2012/Problem10_label00.yml | 67.66 | 46.77 | 20.89 | 954 | 1 | 953 | 99.90% |
| eca-rers2012/Problem10_label20.yml | 67.66 | 46.77 | 20.89 | 954 | 1 | 953 | 99.90% |
| eca-rers2012/Problem10_label41.yml | 68.16 | 47.26 | 20.90 | 954 | 1 | 953 | 99.90% |
| seq-mthreaded/pals_lcr-var-start-time.4.1.ufo.BOUNDED-8.pals.yml | 37.50 | 8.33 | 29.17 | 82 | 75 | 7 | 8.54% |
| seq-mthreaded/pals_lcr-var-start-time.4.ufo.BOUNDED-8.pals.yml | 38.89 | 8.33 | 30.56 | 82 | 75 | 7 | 8.54% |
| seq-mthreaded/pals_lcr-var-start-time.4.2.ufo.BOUNDED-8.pals.yml | 39.44 | 8.45 | 30.99 | 79 | 72 | 7 | 8.86% |
| termination-memory-alloca/count_down-alloca-2.yml | 88.89 | 55.56 | 33.33 | 9 | 7 | 2 | 22.22% |

Table 5.17: Outliers for TracerX (CondTest/ Naïve Algorithm). Correlation Coefficient $r = 0.3809$. Threshold = 15.

| Task | Branch Coverage | | | Cyclomatic Complexity | | | | |
|---|---|---|---|---|---|---|---|---|
| | CondTest | Naïve | Δ | Initial | CondTest | Naïve | Δ | % red. |
| array-patterns/array4_pattern.yml | 22.22 | 77.78 | -55.56 | 11 | 11 | 3 | 8 | 72.73% |
| seq-pthread/cs_lazy.yml | 3.26 | 42.39 | -39.13 | 81 | 81 | 487 | -406 | -501.23% |
| seq-pthread/cs_read_write_lock-1.yml | 2.94 | 38.24 | -35.30 | 87 | 87 | 644 | -557 | -640.23% |
| seq-pthread/cs_stateful-1.yml | 3.12 | 37.50 | -34.38 | 83 | 83 | 402 | -319 | -384.34% |
| seq-pthread/cs_stateful-2.yml | 3.12 | 37.50 | -34.38 | 83 | 83 | 402 | -319 | -384.34% |
| loops/terminator_01.yml | 33.33 | 66.67 | -33.34 | 5 | 5 | 4 | 1 | 20.00% |
| loop-acceleration/multivar_1-1.yml | 33.33 | 66.67 | -33.34 | 5 | 5 | 4 | 1 | 20.00% |
| loop-acceleration/multivar_1-2.yml | 33.33 | 66.67 | -33.34 | 5 | 5 | 4 | 1 | 20.00% |
| termination-15/cstrchr_malloc.yml | 33.33 | 66.67 | -33.34 | 6 | 6 | 5 | 1 | 16.67% |
| termination-memory-alloca/b.03-no-inv_assume-alloca.yml | 66.67 | 100.00 | -33.33 | 7 | 7 | 2 | 5 | 71.43% |
| termination-memory-alloca/cstrchr-alloca-1.yml | 66.67 | 100.00 | -33.33 | 8 | 8 | 5 | 3 | 37.50% |
| termination-15/cstrchr_diffterm_alloca.yml | 66.67 | 100.00 | -33.33 | 6 | 6 | 5 | 1 | 16.67% |
| termination-15/cstrchr_reverse_alloca.yml | 66.67 | 100.00 | -33.33 | 6 | 6 | 5 | 1 | 16.67% |
| seq-pthread/cs_stack-1.yml | 2.80 | 35.66 | -32.86 | 123 | 123 | 629 | -506 | -411.38% |
| seq-pthread/cs_sync.yml | 3.06 | 30.61 | -27.55 | 84 | 84 | 492 | -408 | -485.71% |
| seq-pthread/cs_time_var_mutex.yml | 2.59 | 29.31 | -26.72 | 93 | 93 | 213 | -120 | -129.03% |
| termination-15/cstrncat_reverse_alloca.yml | 58.33 | 83.33 | -25.00 | 11 | 11 | 12 | -1 | -9.09% |
| seq-pthread/cs_queue-1.yml | 2.67 | 23.53 | -20.86 | 147 | 147 | 911 | -764 | -519.73% |
| loops/terminator_03-1.yml | 80.00 | 100.00 | -20.00 | 8 | 8 | 8 | 0 | 0.00% |
| termination-memory-alloca/a.09_assume-alloca.yml | 60.00 | 80.00 | -20.00 | 8 | 8 | 2 | 6 | 75.00% |
| termination-memory-alloca/b.03_assume-alloca.yml | 60.00 | 80.00 | -20.00 | 8 | 8 | 2 | 6 | 75.00% |
| termination-memory-alloca/gcd1-alloca.yml | 33.33 | 50.00 | -16.67 | 8 | 8 | 7 | 1 | 12.50% |
| termination-memory-linkedlists/ll_search-alloca.yml | 83.33 | 100.00 | -16.67 | 10 | 10 | 4 | 6 | 60.00% |
| seq-pthread/cs_peterson.yml | 2.88 | 18.27 | -15.39 | 88 | 88 | 136 | -48 | -54.55% |
| array-tiling/skippedu.yml | 92.31 | 76.92 | 15.39 | 12 | 12 | 1 | 11 | 91.67% |
| list-ext2-properties/simple_and_skiplist_2lvl-1.yml | 85.71 | 64.29 | 21.42 | 21 | 21 | 2 | 19 | 90.48% |
| forester-heap/sll-simple-white-blue-2.yml | 92.31 | 69.23 | 23.08 | 12 | 12 | 1 | 11 | 91.67% |
| forester-heap/dll-simple-white-blue-1.yml | 93.33 | 66.67 | 26.66 | 13 | 13 | 1 | 12 | 92.31% |
| heap-data/cart.yml | 88.89 | 33.33 | 55.56 | 11 | 11 | 1 | 10 | 90.91% |
| array-patterns/array9_pattern.yml | 85.71 | 28.57 | 57.14 | 10 | 10 | 3 | 7 | 70.00% |

#### 5.2.1.7 Summary

We have seen that most testers benefit from a conditional testing approach. Our algorithm most of the time fared slightly better than CondTest's pruner. It did not matter whether we used our approach with propagation or simply our naïve algorithm. This is because in many cases they produce very similar conditions and thus the reduced programs were similar.

Definition 6 gives us an explanation why conditions might be similar: A path $\pi$ is covered if (among others) "$q_k$ is an accepting state". We used this by generating unaccpting states ($\perp$) whenever we found an uncovered node: 1) In the naïve algorithm's case uncovered nodes are those that are not in our covered goals list. 2) For our propagating algorithm uncovered nodes are those whose children are uncovered or that can't be found in our covered goals list. We can observe that all paths that aren't covered for the first case aren't covered in the second case either, the cutoff (i.e. the point where we generate our $\perp$ assumption) can just happen *earlier*. This means that our optimized algorithm always removes as much or more than our naïve approach.

Although somewhat surprisingly our contribution sometimes generated programs that were as much as six times as complex as the original program, this did not have any influence on branch coverage whatsover. We found that the amount that was reduced (in terms of cyclomatic complexity) and branch coverage did not correlate. This could be explained as follows: We use a tester that has particular strengths. If it struggles with one part of the program it can't produce many test cases for it. When reducing the program we remove those parts for which we could successfully generate test cases leaving all parts the tester struggled with in. If the hard part of a program was just one branch our cyclomatic complexity would decrease sharply, nevertheless it wouldn't have any large impact on our tester.

We can now look at the resource consumption of each run. Figure 5.8 shows a box diagram of the cpu time. In blue is our baseline. CondTest is colored in green. Our naïve approach is called "thesis" and colored in orange. In pink we see our algorithm with propagation. We calculated the values for the diagram as follows: Baseline gets its data from Benchexec, the data was not modified. The other approaches take their measurements from CoVeriTeam. We added the duration of *tester1* and *tester2*.

We can see that PRTest almost always uses its given cpu time. Sporadically there are some runs that took less time. Symbiotic uses next to zero CPU time in most of the cases. There are quite a few outliers, though. Our approach fares very well with HybridTiger and CoVeriTest while CondTest took the lead in TracerX and had a

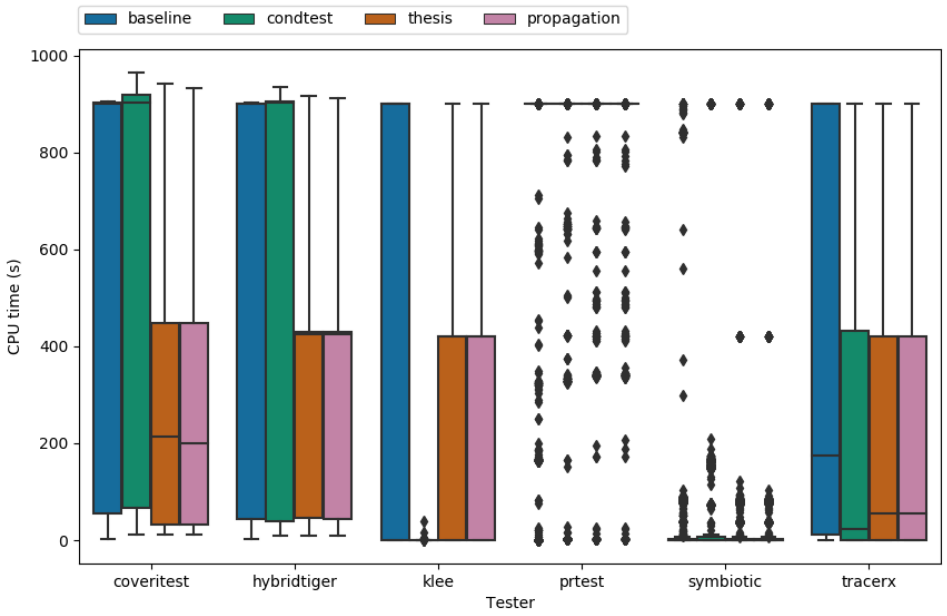
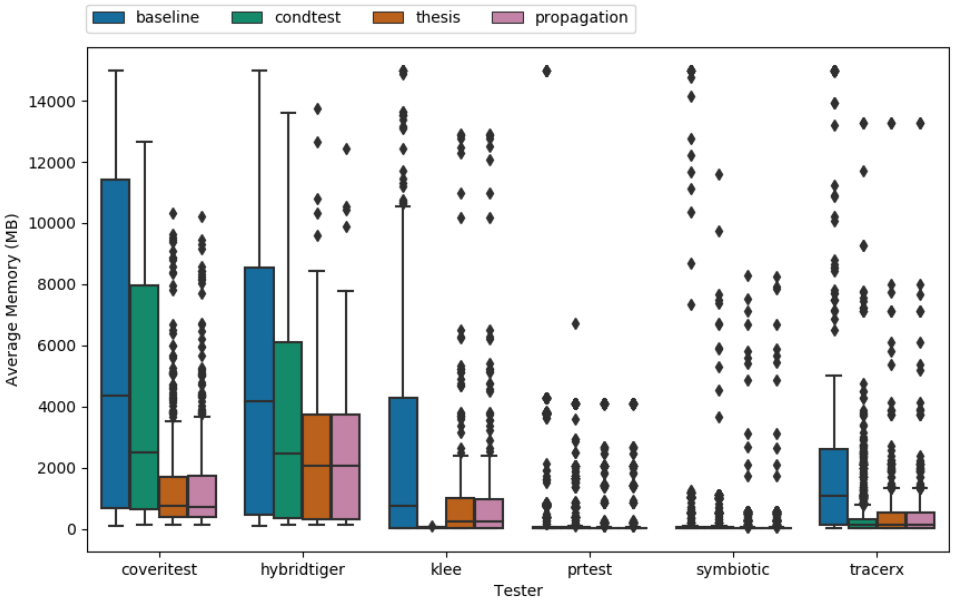



Figure 5.8: The CPU time consumption of all testers in seconds.

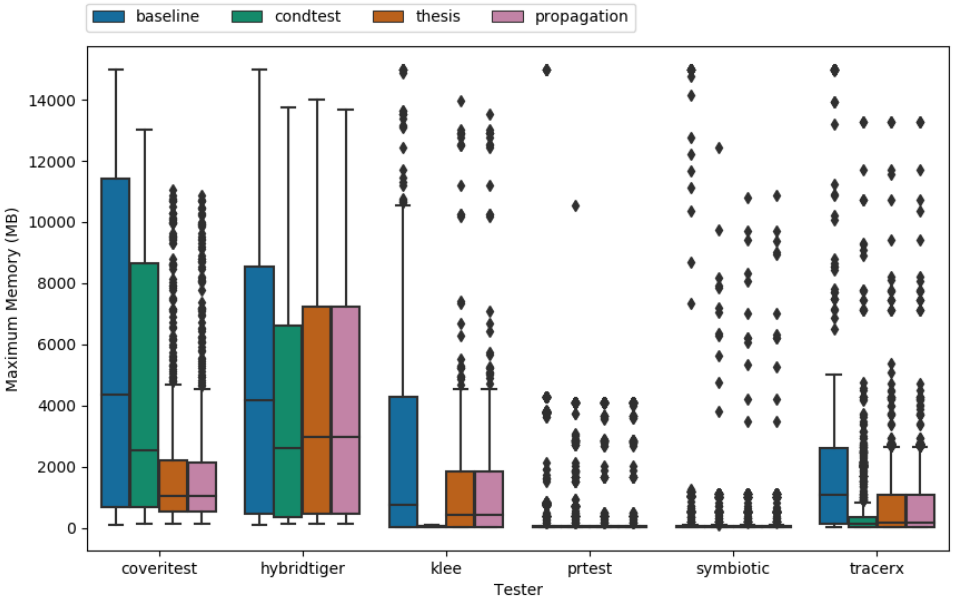
stellar performance with Klee. Conditional Testing was always on par or better than the traditional approach. Our optimization using propagation didn't use significantly more CPU time than the naïve variant.

In Figure 5.9 we can see a box plot of the RAM usage of our experiments. Figure 5.9a takes the memory usage of *tester1* and *tester2* and halfs them (i.e. the artihmetic average). Figure 5.9a shows the **maximum** amount of RAM each pair of testers used (i.e. $\max(\text{tester1}, \text{tester2})$).

PRTest and Symbiotic used next to zero memory. Our algorithms fared really well when combined with CoVeriTest or HybridTiger. However, for the latter when comparing maxium memory usage they performed slightly worse than CondTest. Klee once again profited massively from being combined with CondTest, too. The tool performed well with TracerX – even better than our algorithm – however our algorithms only performed worse in the upper quartile, leaving 75% about as good as CondTest. Again our naïve algorithm and the optimized version were nearly identical.




(a) Combined approach's memory usage is averaged out.




(b) Combined approach's memory usage maximum.

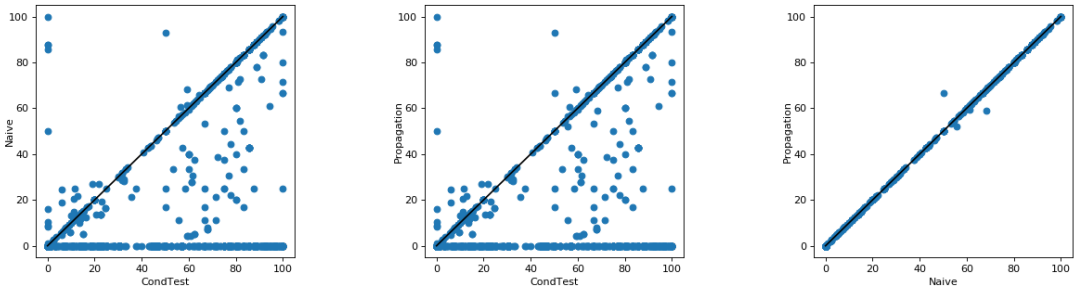
Figure 5.9: The RAM consumption of all testers in MB.

### 5.2.2 Different Testers

We tried executing different testers after another. We chose PRTest as the first tester because it is relatively simple in nature and can be used to quickly create a baseline for tests [22]. Other, more sophisticated testers can then take over and try to generate tests for harder parts of the program.

It might be beneficial to use more than two test generators (i.e. three or more). Future research is needed to find other good pairings and try out those ideas.

#### 5.2.2.1 PRTest and CoVeriTest



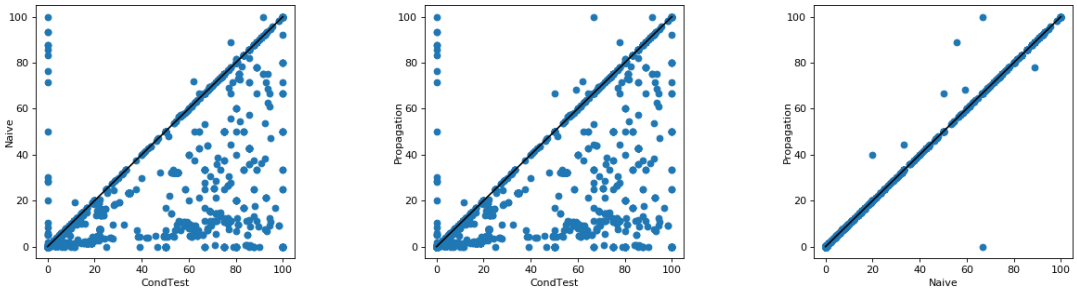

(a) CondTest/ Naïve (b) CondTest/ Propagation (c) Naïve/ Propagation

Figure 5.10: Branch Coverage (in %) for PRTest + CoVeriTest.

From Figure 5.10 it is obvious that our algorithm performed worse than CondTest in nearly every task. The simple implementation and its counterpart once again performed virtually identical.

#### 5.2.2.2 PRTest and HybridTiger

The impression from the previous section continues. In Figure 5.11 we can see that our algorithms performed even worse than before. Again, propagation and no propagation are somewhat similar, however now there are data points in which propagation had a slight advantage.




(a) CondTest/ Naïve (b) CondTest/ Propagation (c) Naïve/ Propagation

Figure 5.11: Branch Coverage (in %) for PRTest + HybridTiger.

#### 5.2.2.3 Summary

We have seen that our algorithms did not fare well when combining PRTest with HybridTiger or CoVeriTest.

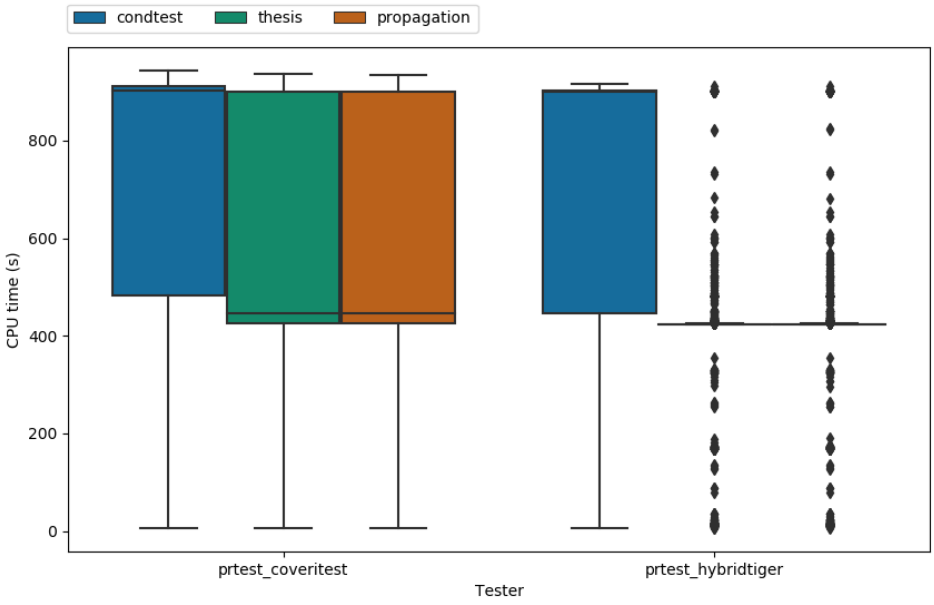



Figure 5.12: The CPU time consumption of all runs of our different testers in seconds.

In Figure 5.12 we can see that for CoVeriTest our algorithms didn't use nearly as much CPU time as CondTest. This continues the trend from Figure 5.8, where our algorithms outperformed CondTest, too. HybridTiger stands out, though. It looks like most tasks ran for about 7 minutes, which coincidentally also is the time we gave PRTest to run. We also know that the tool runs until it is interrupted or killed. If the second tester didn't run at all (or just for a short time) this could explain the discrepancy in branch coverage.

Figure 5.13 shows the RAM usage. In Figure 5.13a we averaged out both runs, in Figure 5.13b we used the maximum. It can be observed that our algorithms used significantly less memory than our reference. We can see that the *average* memory naïve and propagation used is about half of our maxima. This would support our theory from the previous paragraph that our second testers only ran for a short amount of time or didn't run at all. It should however be noted that PRTest's memory usage also shrank in the average case compare to the maximum.

## 5.3 Threats to validity

In this section we'll briefly describe challenges that we encountered and other risks that might threaten the validity of our data.

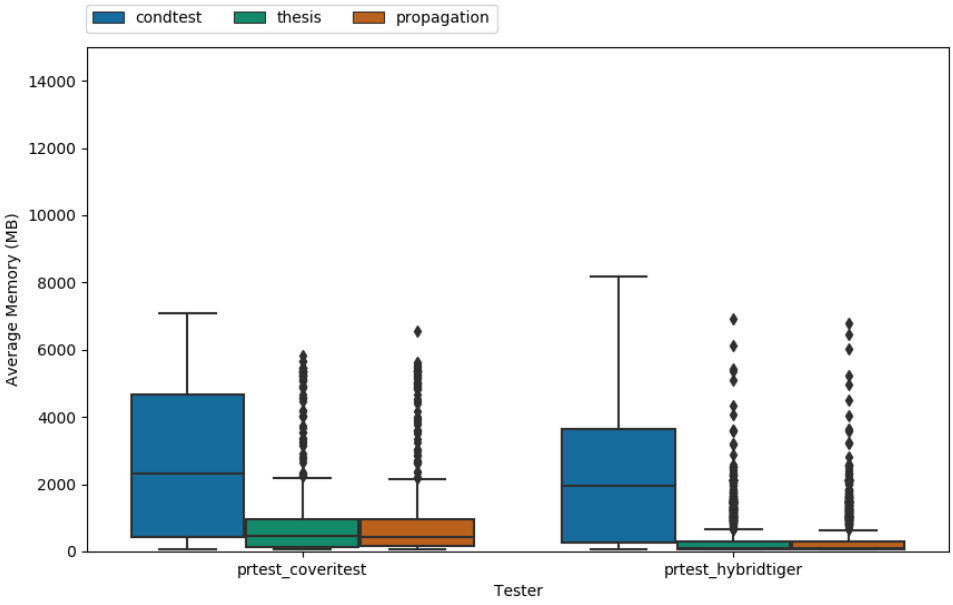
### 5.3.1 Bug in the linux kernel

While evaluating our algorithms we discovered a bug in the linux kernel that temporarily prevented Benchexec from running some of our tools[8]. This forced us to update some of our tools mid-evaluation.

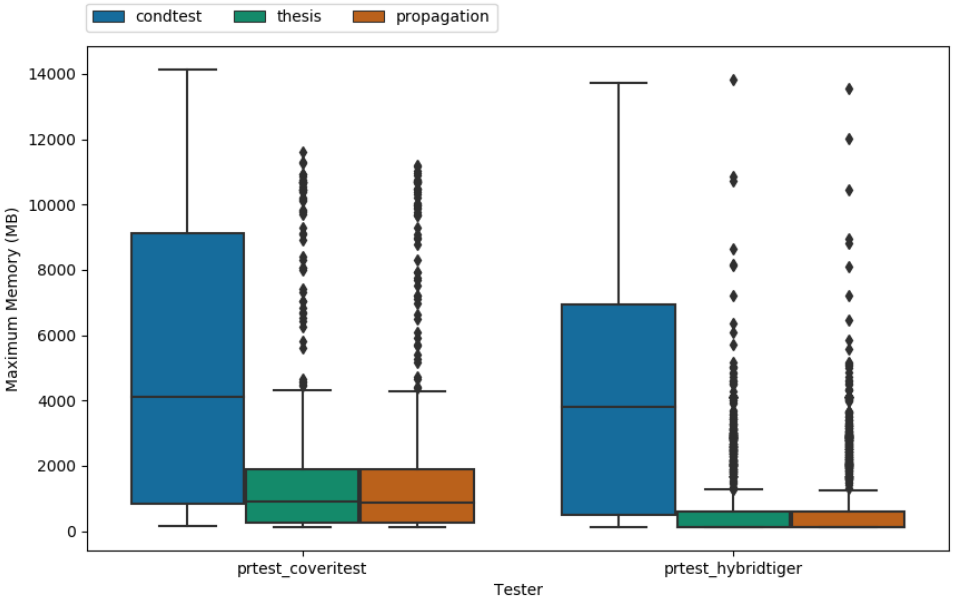
### 5.3.2 Differing versions of dependencies

Test-Comp 2020's candidates ran on a cluster of Ubuntu 18.04 machines. The servers were updated to Ubuntu 20.04 in the following months which led to some dependencies (like `psutil` and `pycparser`) to not work properly anymore. We had to update them, but some tasks ran before the upgrade (with dependencies that were specifically built

[8] See issue #621 in Benchexec's GitHub repository (`https://github.com/sosy-lab/benchexec/issues/621`) and the associated bug in Ubuntu's bug tracker (`https://bugs.launchpad.net/ubuntu/+source/linux/+bug/1900141`).




(a) Combined approach's memory usage is averaged out.




(b) Combined approach's memory usage maximum.

Figure 5.13: The RAM consumption of the different testers in MB.

for Ubuntu 18.04) while most ran after the upgrade. We couldn't repeat the experiments due to the time constraints of a bachelor's thesis.

### 5.3.3 CoVeriTeam fails when it should have succeeded

A valid test suite is a folder (or zip file) that at least contains one `metadata.xml` and zero or more test case XML files. Under some circumstances (e.g. the program requires no inputs because it has been reduced) testers do not produce a valid test suite. Instead, the output directory is empty. This however causes CoVeriTeam to not generate a `execution_trace.xml`, which in turn causes our local analysis to fail. We can't discern between runs that did actually fail and ones that did not follow this specification. In some cases this greatly reduces the number of runs we were able to analyze and is especially concerning because it throws out the most successful runs: If our run of tester 1 produced a test suite with 100% coverage and our program subsequently was reduced to `int main(){ }` then our second tester wouldn't generate a test suite. This would cause a run with 100% coverage to be thrown out.

### 5.3.4 pycparser doesn't correctly parse C in some cases

We used CondTest for instrumentation. CondTest depends on `pycparser` for parsing C programs. Alas in some cases pycparser is not able to parse a valid C file. Consider the excerpt in Figure 5.14. Even though a compiler like GCC is successfully able to compile the program and accepts it as valid C[9], pycparser throws a *parser error* in line 631. This causes our evaluation to erroneously fail even though it could have succeeded.

We estimate the amount of programs to have mistakenly been thrown out at around 100.

### 5.3.5 Limited selection of pairs of testers

In this thesis we only looked at two pairs of testers. To properly understand phenomena related to combining different testers we should increase our sample size (e.g. by using different pairs of testers). This way we could find combinations where our algorithms perform as well or better than the reference.

[9]See the corresponding godbolt: `https://godbolt.org/z/sqPEra`.

```
626 void ldv_set_del(Element e, Set s) {
627         struct ldv_list_element *m;
628         struct ldv_list_element *n;
629         for (m = ({ const typeof( ((typeof(*m) *)0)->list ) *__mptr = ((s
                )->next); (typeof(*m) *)( (char *)__mptr - ((size_t) &((typeof
                (*m) *)0)->list) );}), n = ({ const typeof( ((typeof(*(m)) *)
                0)->list ) *__mptr = ((m)->list.next); (typeof(*(m)) *)( (char
                 *)__mptr - ((size_t) &((typeof(*(m)) *)0)->list) );}); &m->
                list != (s); m = n, n = ({ const typeof( ((typeof(*(n)) *)0)->
                list ) *__mptr = ((n)->list.next); (typeof(*(n)) *)( (char *)
                __mptr - ((size_t) &((typeof(*(n)) *)0)->list) );})) {
630             if(m->e == e) {
631                     ldv_list_del(&m->list);
632                     free(m);
633             }
634         }
635 }
```

Figure 5.14: An excerpt from *ldv-sets/test_add-1.i* for which PYCPARSER fails.

# 6 Future Work

As explained in chapter 3 there are more coverage criterions than just branch coverage. One that might be particularly suitable for our use case is *condition coverage*.

Condition coverage requires each part of the assumption in an `if` statement to evaluate to both `true` and `false` at least once. In our generated condition we could — instead of just using $\top$ and $\bot$ assumptions — have our assumptions contain which parts have been explored. In addition to the list of covered goals we would have to save the relevant local variables to determine which parts of the if-condition have evaluated to true. Then our condition generator would use that information to generate these conditions.

This thesis only evaluates sequential combinations of testers. However, there are various other combinations, that we could try [12]:

- *Cyclic Tester*. We could let a tester run for a certain amount of time, prune the program and feed it back to the same tester. We tried this for 1 round, it would be interesting to see how the results change with $N$ rounds.
- *Strategy-Selection Tester*. We can use a function to determine which tester should generate tests for a particular path of a program. This function might be completely random or do an analysis "pre-pass" to select an appropriate tool.
- *Compositional Tester.* We can split our set of test goals to be covered and feed each tester only half by creating corresponding conditions. This way we can effectively halve the work each test case generator has to do.

In this thesis we weren't able to test some of Test-Comp 2020's candidates, namely Legion, LibKluzzer and VeriFuzz. We should evaluate them in the future.

It might be interesting how different pairings of testers influence our results. In this thesis we chose PRTest and one other tester because PRTest can quickly generate test cases and leave the harder goals to more sophisticated testers. However there might be other combinations that yield better results. It might even be beneficial to combine three or more testers.

Additionally, our time limits were completely arbitrary. We could measure the same set up with different time limits and try to find an optimal value that isn't as hard on the resource consumption while still providing a good branch coverage.

Another approach that consists of just generating a `true` assumption for each covered goal and a `false` assumption for each uncovered goal (without doing a search for leaf goals) could be used, too.

# 7 Conclusion

Testing is an important part of software development, but is very expensive. That's why we employ test case generators to create the tests for us. These have different strengths based on their technique: Some might use symbolic execution, others random testing. By combining various testers (e.g. by using reducers) we can achieve a higher branch coverage than if we let every tester run alone [12].

Beyer et. al [12] presented CondTest, a suite of tools that can both extract covered test goals from a program by means of instrumentation and reduce the program. We want to construct *condition automata* from lists of test goal labels that CondTest extracted.

For that we presented two algorithms, *naive* and *propagation* to generate condition automata from test goal lists and evaluated them against the benchmark set of Test-Comp 2020 using its candidates.

We found that for executing one tester for 7 minutes, reducing the program and then running the tester again for 8 minutes our algorithms usually improved branch coverage compare to running the tester for 15 minutes straight. Used together with CoVeriTest, HybridTiger, PRTest, Symbiotic or TracerX they performed as well or better than CondTest and used less resources (i.e. CPU time and RAM). Our algorithms almost always reduced more of the program in terms of cyclomatic complexity, however in some rare cases additional complexity was added. Nevertheless we saw that the amount of complexity of the program we could reduce did not correlate with branch coverage.

We discovered that our algorithms didn't work as well when combining two different testers; In nearly all cases branch coverage was equal to CondTest's or worse. We theorized this might be a sign of only the first tester running. This theory was supported by measurements of RAM usage and CPU time.

# List of Figures

# List of Tables

# List of Theorems

# Bibliography


[1] F. E. Allen. “Control flow analysis.” In: *Proceedings of a symposium on Compiler optimization* -. ACM Press, 1970. DOI: 10.1145/800028.808479.

[2] K. Beck. *Extreme Programming Explained: Embrace Change, 2nd Edition (The XP Series)*. Addison-Wesley, Nov. 2004. ISBN: 9780321278654.

[3] K. Beck. *Test Driven Development: By Example*. Addison-Wesley Professional, Nov. 2002. ISBN: 9780321146533.

[4] N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. “Proofs from Tests.” In: *Proceedings of the 2008 International Symposium on Software Testing and Analysis*. ISSTA ’08. Seattle, WA, USA: Association for Computing Machinery, 2008, pp. 3–14. ISBN: 9781605580500. DOI: 10.1145/1390630.1390634.

[5] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. “Generating tests from counterexamples.” In: *Proceedings. 26th International Conference on Software Engineering*. 2004, pp. 326–335. DOI: 10.1109/ICSE.2004.1317455.

[6] D. Beyer. “Second Competition on Software Testing: Test-Comp 2020.” In: *Proceedings of the 22nd International Conference on Fundamental Approaches to Software Engineering (FASE 2020, Dublin, Ireland, April 25-30)*. LNCS 12076. Springer, 2020, pp. 505–519. DOI: 10.1007/978-3-030-45234-6_25.

[7] D. Beyer, S. Gulwani, and D. A. Schmidt. “Combining Model Checking and Data-Flow Analysis.” In: *Handbook of Model Checking*. Ed. by E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem. Cham: Springer International Publishing, 2018, pp. 493–540. ISBN: 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8_16.

[8] D. Beyer, T. A. Henzinger, M. E. Keremoglu, and P. Wendler. “Conditional Model Checking: A Technique to Pass Information between Verifiers.” In: *Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2012, Cary, NC, November 10-17)*. Ed. by T. Bultan and M. Robillard. ACM, 2012. ISBN: 978-1-4503-1614-9.

[9] D. Beyer, A. Holzer, M. Tautschnig, and H. Veith. "Information Reuse for Multi-goal Reachability Analyses." In: *Programming Languages and Systems*. Ed. by M. Felleisen and P. Gardner. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 472–491. ISBN: 978-3-642-37036-6.

[10] D. Beyer and M.-C. Jakobs. "CoVeriTest: Cooperative Verifier-Based Testing." In: *Proceedings of the 22nd International Conference on Fundamental Approaches to Software Engineering (FASE 2019, Prague, Czech Republic, April 6-11)*. LNCS 11424. Springer, 2019, pp. 389–408. DOI: 10.1007/978-3-030-16722-6_23.

[11] D. Beyer, M.-C. Jakobs, T. Lemberger, and H. Wehrheim. "Reducer-Based Construction of Conditional Verifiers." In: *Proceedings of the 40th International Conference on Software Engineering (ICSE 2018, Gothenburg, Sweden, May 27 - June 3)*. ACM, 2018, pp. 1182–1193. ISBN: 978-1-4503-5638-1. DOI: 10.1145/3180155.3180259.

[12] D. Beyer and T. Lemberger. "Conditional Testing - Off-the-Shelf Combination of Test-Case Generators." In: *Proceedings of the 17th International Symposium on Automated Technology for Verification and Analysis (ATVA 2019, Taipei, Taiwan, October 28-31)*. Ed. by Y.-F. Chen, C.-H. Cheng, and J. Esparza. LNCS 11781. Springer, 2019, pp. 189–208. DOI: 10.1007/978-3-030-31784-3_11.

[13] S. Cruanes, G. Hamon, S. Owre, and N. Shankar. "Tool Integration with the Evidential Tool Bus." In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by R. Giacobazzi, J. Berdine, and I. Mastroeni. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 275–294. ISBN: 978-3-642-35873-9.

[14] J. Edvardsson. *A Survey on Automatic Test Data Generation*. Mar. 2002.

[15] P. Godefroid, N. Klarlund, and K. Sen. "DART: Directed automated random testing." In: vol. 40. June 2005, pp. 213–223. DOI: 10.1145/1065010.1065036.

[16] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. D. Tetali. "Compositional May-Must Program Analysis: Unleashing the Power of Alternation." In: *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '10. Madrid, Spain: Association for Computing Machinery, 2010, pp. 43–56. ISBN: 9781605584799. DOI: 10.1145/1706299.1706307.

[17] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. "SYNERGY: A New Algorithm for Property Checking." In: *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. SIGSOFT '06/FSE-14. Portland, Oregon, USA: Association for Computing Machinery, 2006, pp. 117–127. ISBN: 1595934685. DOI: 10.1145/1181775.1181790.

[18] A. Holzer, C. Schallhart, M. Tautschnig, and H. Veith. "Query-Driven Program Testing." In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by N. D. Jones and M. Müller-Olm. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 151–166. ISBN: 978-3-540-93900-9.

[19] *Information technology – Programming languages – C*. Standard. Geneva, CH: International Organization for Standardization, 2018.

[20] Y. Kim, Z. Zu, M. Kim, M. B. Cohen, and G. Rothermel. "Hybrid Directed Test Suite Augmentation: An Interleaving Framework." In: *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. 2014, pp. 263–272. DOI: 10.1109/ICST.2014.39.

[21] B. Korel. "Automated software test data generation." In: *IEEE Transactions on Software Engineering* 16.8 (1990), pp. 870–879.

[22] T. Lemberger. "Plain random test generation with PRTest." In: *International Journal on Software Tools for Technology Transfer (STTT)* (2020). DOI: 10.1007/s10009-020-00568-x.

[23] R. Majumdar and K. Sen. "Hybrid Concolic Testing." In: *29th International Conference on Software Engineering (ICSE'07)*. 2007, pp. 416–426. DOI: 10.1109/ICSE.2007.41.

[24] T. Margaria, R. Nagel, and B. Steffen. "jETI: A Tool for Remote Tool Integration." In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by N. Halbwachs and L. D. Zuck. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 557–562. ISBN: 978-3-540-31980-1.

[25] T. J. McCabe. "A Complexity Measure." In: *IEEE Transactions on Software Engineering* SE-2.4 (1976), pp. 308–320.

[26] J. C. Miller and C. J. Maloney. "Systematic Mistake Analysis of Digital Computer Programs." In: *Commun. ACM* 6.2 (Feb. 1963), pp. 58–63. ISSN: 0001-0782. DOI: 10.1145/366246.366248.

[27] G. J. Myers. *The Art of Software Testing*. 2nd Edition. New York: John Wiley and Sons, 2004. ISBN: 978-0-471-67835-9.

[28] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer Berlin Heidelberg, 1999. DOI: 10.1007/978-3-662-03811-6.

[29] Y. Noller, R. Kersten, and C. S. Psreanu. "Badger: Complexity Analysis with Fuzzing and Symbolic Execution." In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2018. Amsterdam, Netherlands: Association for Computing Machinery, 2018, pp. 322–332. ISBN: 9781450356992. DOI: 10.1145/3213846.3213868.

[30] J. Rushby. "An Evidential Tool Bus." In: *Formal Methods and Software Engineering*. Ed. by K.-K. Lau and R. Banach. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 36–36. ISBN: 978-3-540-32250-4.

[31] B. Steffen, T. Margaria, and V. Braun. "The Electronic Tool Integration platform: concepts and design." In: *International Journal on Software Tools for Technology Transfer* 1.1-2 (Dec. 1997), pp. 9–30. DOI: 10.1007/s100090050003.

[32] K. Taneja, T. Xie, N. Tillmann, and J. de Halleux. "EXpress: Guided Path Exploration for Efficient Regression Test Generation." In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ISSTA '11. Toronto, Ontario, Canada: Association for Computing Machinery, 2011, pp. 1–11. ISBN: 9781450305624. DOI: 10.1145/2001420.2001422.

[33] Z. Xu, Y. Kim, M. Kim, and G. Rothermel. "A Hybrid Directed Test Suite Augmentation Technique." In: *2011 IEEE 22nd International Symposium on Software Reliability Engineering*. 2011, pp. 150–159. DOI: 10.1109/ISSRE.2011.21.