# Detection of Polluting Test Objectives for Dataflow Criteria

Thibault Martin[1], Nikolai Kosmatov[1,2], Virgile Prevosto[1], and Matthieu Lemerre[1]

[1]CEA, LIST, Software Safety and Security Laboratory, Palaiseau, France
firstname.lastname@cea.fr
[2]Thales Research & Technology, Palaiseau, France
nikolaikosmatov@gmail.com

**Abstract.** Dataflow test coverage criteria, such as all-defs and all-uses, belong to the most advanced coverage criteria. These criteria are defined by complex artifacts combining variable definitions, uses and program paths. Detection of polluting (i.e. inapplicable, infeasible and equivalent) test objectives for such criteria is a particularly challenging task. This short paper evaluates three detection approaches involving dataflow analysis, value analysis and weakest precondition calculus. We implement and compare these approaches, analyze their detection capacities and propose a methodology for their efficient combination. Initial experiments illustrate the benefits of the proposed approach.

## 1 Introduction

Among a large range of test coverage criteria proposed in the literature, dataflow (coverage) criteria [1, 2], such as all-defs and all-uses, belong to the most advanced. These criteria are defined by complex artifacts combining a (program) location where a variable is defined, a location where it is used, and a path from the definition to the use such that the variable is not redefined in between (called a *def-clear path*). Like for many other criteria (e.g. conditions, mutations), some test objectives are not relevant (or *polluting*): they should be removed to properly ensure or evaluate test coverage [3, 4]. Polluting test objectives for dataflow criteria include *inapplicable* test objectives [5], where a def-use pair cannot be linked by a def-clear path. They also include *infeasible* test objectives, where a def-use pair can be linked by at least one def-clear path, but none of these paths is feasible (i.e. can be activated by a test case). Finally, they include *duplicate* (or *equivalent*) test objectives, which are always covered simultaneously: it suffices to keep only one objective for each equivalence class.

While creating a list of (candidate) test objectives for dataflow criteria can look easy, detection of polluting objectives for such criteria is challenging, due to a complex definition, mixing reachability and def-clear paths. Yet it is crucial to avoid a waste of time during test generation (trying to cover polluting test objectives) and to allow a correct computation of coverage ratios (by ignoring polluting objectives in the total number of objectives). While applying dataflow criteria for testing [2, 6–10] and detection of polluting test objectives for simpler criteria (see [3, 11] for some recent results and related work) were previously studied, evaluating and combining various program analysis techniques for their detection for dataflow criteria—the purpose of this work—was not investigated.

*Contributions.* This short paper evaluates three approaches to detecting polluting test objectives for dataflow criteria, involving dataflow analysis, (abstract interpretation based) value analysis, and weakest precondition calculus. We implement these approaches inside the LTest open-source testing toolset [12]. We evaluate and compare them by some initial experiments and analyze their detection capacities and limitations. We focus on the key ingredients of dataflow criteria: def-use pairs. The detection capacities we observed appear to be different from similar experiments made previously for other criteria. Finally, we propose a methodology for an efficient combination of several techniques.

## 2 Background and Motivating Example

*Background.* A large range of test coverage criteria have been defined [1]. Recent work proposed HTOL (Hyperlabel Test Objective Language) [4], a generic specification language for test objectives, that can express most of these criteria. We present here only the subset of HTOL that is useful to express dataflow criteria.

Given a program $P$, a *label* $\ell$ (in the sense of [13]) is a pair $(loc, \varphi)$ where $loc$ is a location in $P$ and $\varphi$ is a predicate. Label $\ell$ is *covered* by a test case $t$ when the execution of $t$ reaches $loc$ and satisfies $\varphi$. While labels can express many simple criteria, test objectives for more complex criteria, including dataflow criteria, need a more general notion, hyperlabels. In our context, we use labels only for reachability, thus we always consider $\varphi = true$ and simplify the notation $\ell \triangleq (loc, true)$ as $\ell \triangleq loc$, that is, a usual label in the sense of C.

Hyperlabels [4] extend labels by relating them with several constructions, that include, among others, sequences, conjunctions and disjunctions. Given two labels $\ell$ and $\ell'$ and a predicate $\psi$, a *sequence* hyperlabel $h \triangleq \ell \xrightarrow{\psi} \ell'$ is covered by a test case $t$ when the execution of $t$ covers both labels $\ell$ and $\ell'$ (in that order), such that the path section between them satisfies predicate $\psi$. A *conjunction* $h_1 \cdot h_2$ requires both hyperlabels $h_1, h_2$ to be covered by (possibly distinct) test cases. A *disjunction* $h_1 + h_2$ requires covering at least one of hyperlabels $h_1, h_2$.

A key test objective of dataflow criteria is a def-use pair. For a given variable $v$ and two labels $\ell, \ell'$, we say that $(\ell, \ell')$ is a *def-use pair for* $v$ if $\ell$ is a definition of $v$ and $\ell'$ is a use (i.e. a read) of $v$. It is linked by a *def-clear path for* $v$ if there is a path from $\ell$ to $\ell'$ such that $v$ is not redefined (strictly) between $\ell$ and $\ell'$. A def-use pair $(\ell, \ell')$ (for $v$) is *covered* by a test case $t$ when the execution of $t$ covers both labels (i.e.—in our context—passes through both locations) $\ell, \ell'$ so that the path between them is a def-clear path (for $v$). When there is no ambiguity, we omit the variable name.

Thus, the test objective to cover a def-use pair $(\ell, \ell')$ can be expressed by a sequence (hyperlabel) $h \triangleq \ell \xrightarrow{dc(v)} \ell'$, where predicate $dc(v)$ requires the path to be def-clear for variable $v$. Dataflow criteria rely on such sequences and require to cover all or some of them in various ways. Therefore we focus in this paper on detection of polluting sequences.

*Polluting Test Objectives.* While the generation of candidate def-use pairs by combining definitions and uses for each variable can seem to be a simple task, many of them are irrelevant, or polluting, for various reasons. First, a def-use

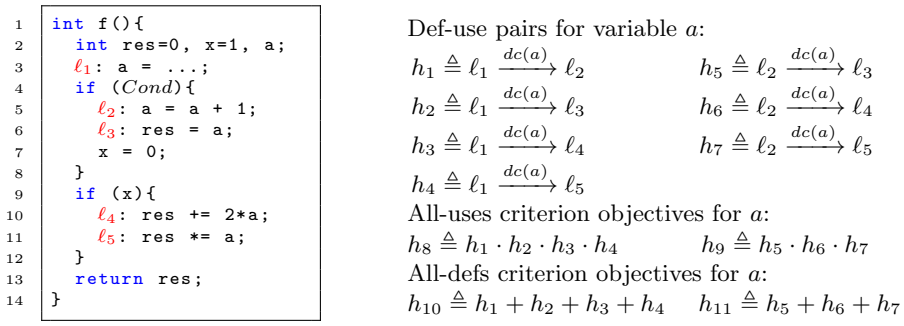


**Fig. 1.** Example of def-use pairs and objectives for all-uses and all-defs for variable $a$.

pair $(\ell, \ell')$ for variable $v$ is *inapplicable* if there is no structurally possible def-clear path from $\ell$ to $\ell'$ for $v$. Second, a def-use pair $(\ell, \ell')$ is *infeasible* if such def-clear paths exist (structurally) but are all infeasible (i.e. cannot be executed by any test case). Inapplicable and infeasible def-use pairs are both *uncoverable*. Finally, a def-use pair $(\ell, \ell')$ is *equivalent* to another def-use pair $(\ell, \ell'')$ if for every test case $t$, the execution of $t$ covers either both pairs or none of them.

Recent research showed that value analysis and weakest precondition calculus can be efficient to detect polluting test objectives for several (non dataflow) criteria [3, 11] expressed in HTOL. For dataflow criteria, model-checking was applied to detect infeasible test objectives [14]. Continuing those efforts, this work adapts several existing program analysis techniques to detect polluting test objectives for dataflow criteria, implements and evaluates them.

Generating only relevant test objectives for dataflow criteria for an arbitrary program is undecidable. Indeed, it requires to identify which uses can be reached from a specific definition. If it were possible, one could apply it to solve the general reachability problem for a label $\ell$ in a given program $P$ : `{code1; ℓ : code2;}` by considering program $P'$ : `{`$\ell_0$ `:int new=0; code1; ℓ : return new; code2;}` with a fresh variable `new` and checking whether the def-use pair $(\ell_0, \ell)$ is generated for $P'$.

*Motivating Example.* Figure 1 gives a simple C code illustrating various cases of polluting objectives. The upper right of the figure shows all (candidate) def-use pairs for variable $a$ $(h_1, \ldots, h_7)$, that include polluting objectives. The all-uses criterion requires to cover all possible def-use pairs for each definition. For the definition $\ell_1$ (resp., $\ell_2$), this corresponds to the conjunctive hyperlabel $h_8$ (resp., $h_9$). The all-defs criterion requires to cover at least one def-use pair for each definition, which is illustrated by the disjunctive hyperlabel $h_{10}$ (resp., $h_{11}$). We see that sequences are key ingredients to express test objectives of these dataflow criteria, and we focus on them below. Naturally, detecting polluting sequences will automatically simplify the combined hyperlabels.

In this example, $h_2$ is inapplicable: structurally, there is no def-clear path from $\ell_1$ to $\ell_3$ since all such paths pass through $\ell_2$. This sequence should be strictly speaking discarded, i.e. erased from the combined objectives.

In addition, we can see that $h_6$ and $h_7$ are both infeasible since no test case can cover $\ell_2$ and $\ell_4$ (or $\ell_5$) at the same time because of $x$. In this case,

the combined objective $h_9$ becomes infeasible as well. It is also easily seen that having *Cond* always false (or true) also makes some objectives infeasible.

Finally, since $\ell_4$ and $\ell_5$ lie in the same consecutive block, sequences $h_3, h_4$ are equivalent: a test case $t$ either covers both of them, or none of them; and so are $h_6, h_7$. Keeping only one in each group would be sufficient both for test generation or infeasibility detection. We can keep $h_3$ for $h_3, h_4$. If $h_3$ is infeasible, $h_4$ is as well. If $h_3$ is covered by some test case $t$, $h_4$ is covered too[1].

For simplicity, we assume here that the C code has been normalized (like it is automatically done in FRAMA-C [15]), in particular, expressions contain no side effects and each function has a unique return point.

## 3 Detection Techniques

### 3.1 Dataflow Analysis for Inapplicable and Equivalent Sequences

*Inapplicable Sequences.* A simple approach to generate sequences expressing def-use pairs consists in performing a simple run through the Abstract Syntax Tree (AST) and creating a sequence for each definition and use of the same variable in the program. This approach leads to a significant number of inapplicable objectives. Their detection for an arbitrary program (e.g. with goto's) is non trivial. For Fig. 1, we would generate $h_1, ..., h_7$, including the inapplicable objective $h_2$.

To avoid generating this kind of objectives, we use a standard dataflow analysis [16]. This analysis propagates (over the program statements) a state associating to each variable $v$ the set $\text{Defs}_v$ of labels corresponding to definitions of $v$ that may reach this point through a def-clear path. This dataflow analysis, denoted $M_{\text{NA}}$, is very efficient to identify Non-Applicable sequences.

Figure 2 illustrates this method for Fig. 1 and variable $a$. The $\text{Defs}_a$ set near a node shows the set of definitions that may have assigned to $a$ the value that $a$ has at this node. So, after visiting $\ell_2$, the definition of $a$ at $\ell_1$ is replaced with that at $\ell_2$. Hence, at $\ell_3$, we will create one sequence ($h_5$) for this use of $a$, and $h_2$ will not be generated. Notice that at node $x$, the state contains both definitions of $a$ after the merge of both branches.

*Equivalent Sequences.* We distinguish two kinds of equivalent sequences. The first kind is trivial. If a variable $v$ is used more than once in the same expression, assuming expressions do not contain side effects, the value of $v$ will be the same for each occurrence. We consider each corresponding def-use pair only once.

The second kind is more complex and relies on the notions of *dominance* and *post-dominance* [17]. For two statements $S_1$ and $S_2$, we say that: $S_1$ *dominates* $S_2$ if all paths from the entry point of the function to $S_2$ pass through $S_1$; $S_2$ *post-dominates* $S_1$ if all paths from $S_1$ to the return point of the function pass through $S_2$.

We enrich the state propagated by $M_{\text{NA}}$ by a set associating to each variable $v$ the set $\text{Uses}_v$ of (labels corresponding to the) uses of $v$ that must (i.e. are guaranteed to) reach this point through a def-clear path. Before creating a

[1] unless the flow is interrupted by a runtime error between $\ell_4$ and $\ell_5$; hence we recommend keeping the first sequence $h_3$, so that a test case covering it either covers $h_4$ as well, or detects a runtime error, that is thus detected and can be fixed.

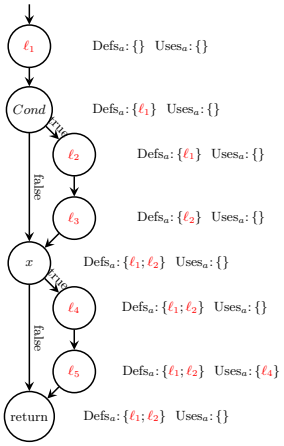


**Fig. 2.** Dataflow analysis for Fig. 1.

```
1  int f(void)
2  {
3    /*@ ghost int C_6^a = 0; */
4    int res = 0, x = 1, a;
5    /*@ ghost C_6^a = 0; */
6    ℓ1: a = ...;
7    if (Cond) {
8      /*@ ghost C_6^a = 0; */
9      ℓ2: a = a + 1;
10     /*@ ghost C_6^a = 1; */
11     ℓ3: res = a;
12     x = 0;
13   }
14   if (x) {
15     /*@ check C_6^a != 1; */
16     ℓ4: res += 2 * a;
17     ℓ5: res *= a;
18   }
19   return res;
20 }
```

**Fig. 3.** Figure 1 instrumented for $h_6$.

sequence with a use at $\ell''$, we check its associated Uses set. If it contains a label $\ell'$ for the same variable, it means that $\ell'$ dominates $\ell''$. Then we check if $\ell''$ post-dominates $\ell'$ using standard dataflow analysis. If so, for each definition $\ell$ in our state, def-use pairs $(\ell, \ell'), (\ell, \ell'')$ are equivalent. Figure 2 illustrates that the $\text{Uses}_a$ state at $\ell_5$ contains $\ell_4$, and since $\ell_5$ also post-dominates $\ell_4$, $h_4$ (resp., $h_7$) is found equivalent to $h_3$ (resp., $h_6$). Notice that after the merge of branches at the last node, $\text{Uses}_a$ is empty. We denote this method by $M_{\text{Eq}}$.

### 3.2 Static Analysis for Uncoverable Sequences

Consider a sequence hyperlabel $h_i$ expressing a def-use pair $\ell \xrightarrow{dc(v)} \ell'$ for $v$. Let $\mathcal{C}_i^v$ be a fresh variable associated to sequence $h_i$, that will represent its status: 0 for uncovered, and 1 for partially covered, i.e. after seeing only its definition, but not its use. More precisely, $\mathcal{C}_i^v$ is initialized to 0 at the beginning of the function. When we reach $\ell$ we set it to 1, meaning that we covered the first member of our sequence $h_i$. If $dc(v)$ is violated (i.e. if we encounter another definition on our path) we set $\mathcal{C}_i^v$ back to 0. If we can prove that $\mathcal{C}_i^v \neq 1$ is always true at $\ell'$, then we show that covering $h_i$ is impossible, that is, $h_i$ is inapplicable or infeasible. A similar method was used in [14] with a model-checking approach.

Figure 3 illustrates this method for Fig. 1 for $h_6 = \ell_2 \xrightarrow{dc(a)} \ell_4$. We express the instrumented code in ACSL [15] as ghost code, used to provide additional code for analyzers, without interfering with the original code semantics. We create the ghost variable $\mathcal{C}_6^a$ that stands for the status of sequence $h_6$ over variable $a$ (cf. lines 3, 5, 8, 10 in Fig. 3). We also generate an ACSL `check` clause $\mathcal{C}_6^a \neq 1$ before $\ell_4$ (line 15). If it is proved, then covering $h_6$ is impossible. Notice that $h_7$ would be proved infeasible as well, since they are equivalent.

To detect polluting sequences automatically, we apply static analysis techniques on the instrumented program, more precisely, Value Analysis based on abstract interpretation and Weakest Precondition calculus. The resulting detection techniques are denoted $M_{\text{VA}}$ and $M_{\text{WP}}$.

| Ex./ Size | Indi- cator | Cand. obj.gen. | Polluting objective detection | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | $M_{\mathrm{NA}}$ | $M_{\mathrm{Eq}}$ | $M_{\mathrm{VA}}$ | $M_{\mathrm{WP}}$ | $M_{\mathrm{NA,Eq}}$ | $M_{\mathrm{NA,Eq,VA}}$ | $M_{\mathrm{NA,Eq,WP}}$ | $M_{\mathrm{NA,Eq,VA,WP}}$ |
| cwe787 34 loc | time | 0.3s | 0.3 | 0.3s | 4.9s | 28.3s | 0.3s | 1.1s | 5,6s | 6,4s |
| | #seq. | 207 | 144 | 99 | 144 | 108 | 171 | 171 | 171 | 171 |
| | %seq. | | 69.6% | 47.8% | 69.6% | 52.2% | 82.6% | 82.6% | 82.6% | 82.6% |
| 2048 376 loc | time | 0.6s | 0.5s | 0.5s | 6.3s | 1m45 | 0.5s | 2.9s | 50.2s | 52.7s |
| | #seq. | 560 | 159 | 187 | 161 | 14 | 293 | 295 | 294 | 295 |
| | %seq. | | 28.4% | 33.4% | 28.8% | 2.5% | 52.3% | 52.7% | 52.5% | 52.7% |
| papa- bench 1399 loc | time | 1.2s | 1.3s | 1.3s | 3.4s | 1m7s | 1.2s | 4.1s | 42.4s | 45.3s |
| | #seq. | 376 | 41 | 106 | 102 | 21 | 139 | 181 | 139 | 181 |
| | %seq. | | 10.9% | 28.2% | 27.1% | 5.6% | 37.0% | 48.1% | 37.0% | 48.1% |
| debie1 5165 loc | time | 1.5s | 1.5s | 1.5s | 1m20s | 9m3s | 1.5s | 37.7s | 4m2s | 4m43s |
| | #seq. | 2149 | 815 | 825 | 876 | 181 | 1272 | 1320 | 1280 | 1323 |
| | %seq. | | 37.9% | 38.4% | 40.8% | 8.4% | 59.2% | 61.4% | 59.6% | 61.6% |
| gzip 4790 loc | time | 3.3s | 2.7s | 2.9s | 22m20s | 71m14s | 2.6s | 4m40s | 29m28s | 33m58s |
| | #seq. | 7299 | 3710 | 2042 | 3806 | 1264 | 4738 | 4834 | 4741 | 4835 |
| | %seq. | | 50.8% | 28.0% | 52.1% | 17.3% | 64.9% | 66.2% | 65.0% | 66.2% |
| itc- benchm. 11825 loc | time | 8.4s | 8.4s | 8.2s | 32.3s | 16m57s | 8.2s | 33.2s | 11m45s | 12m18s |
| | #seq. | 3776 | 301 | 892 | 1145 | 152 | 1107 | 1703 | 1174 | 1708 |
| | %seq. | | 8.0% | 23.6% | 30.3% | 4.0% | 29.3% | 45.1% | 31.1% | 45.2% |
| Mono- cypher 1913 loc | time | 28.2s | 1.4s | 7.0s | MO | TO | 0.9s | 16m32s | 64m31s | 80m10s |
| | #seq. | 45707 | 38880 | 23839 | – | – | 43410 | 43414 | 43410 | 43414 |
| | %seq. | | 85.1% | 52.2% | – | – | 95.0% | 95.0% | 95.0% | 95.0% |
| Average | %seq. | | 41.5% | 35.9% | 41.5% | 15% | 60.0% | 64.4% | 60.4% | 64.5% |

**Fig. 4.** Polluting objectives detected by different techniques and their combinations.

## 4 Implementation and Evaluation

*Implementation.* We implemented the detection techniques described in Sect. 3 in LTest[2] [12], a set of tools for coverage oriented testing, mostly written in OCaml as plugins of Frama-C [15], a program analysis platform for C code. One of the tools, LAnnotate, creates test objectives for a given criterion. It supports various dataflow criteria (such as def-use, all-defs, all-uses) and generates (candidate) objectives inside each function. We implemented dataflow analysis techniques $M_{\mathrm{NA}}$ and $M_{\mathrm{Eq}}$ in LAnnotate to filter out, resp., Non-Applicable and Equivalent objectives. It does not support pointers yet, and overapproximates arrays (meaning that an assignment at index $i$ is seen as an assignment to the entire array). We implemented $M_{\mathrm{VA}}$ and $M_{\mathrm{WP}}$ in another tool, LUncov. detecting uncoverable objectives. It performs interprocedural analysis and relies on Frama-C plugins Eva for value analysis and Wp for weakest precondition.

*Experiments.* In our evaluation, we address the following research questions:

**RQ1**: Is dataflow analysis with $M_{\mathrm{NA}}$ and $M_{\mathrm{Eq}}$ effective to detect inapplicable and equivalent test objectives? Can it scale to real-world applications?

**RQ2**: Can sound static analysis techniques $M_{\mathrm{VA}}, M_{\mathrm{WP}}$ effectively find uncoverable objectives? Can they scale to real-world applications?

**RQ3**: Is it useful to combine these approaches? What is the best combination?

We use a set of real-life C benchmarks[3] of various size (up to 11 kloc) and nature, and focus on sequences encoding def-use pairs (cf. Sect. 2). Figure 4 illustrates the results. For each benchmark, we first generate all candidate def-use pairs using LAnnotate without any additional analysis (see the third column in Fig. 4). Next, we apply the techniques, first separately (columns $M_{\mathrm{NA}}$–$M_{\mathrm{WP}}$) and then in combination (last four columns). We report execution time, the

[2] available at `https://github.com/ltest-dev/LTest`

[3] taken from `https://git.frama-c.com/pub/open-source-case-studies`

number of sequences (i.e. def-use pairs) detected as polluting, and the percentage it represents over the total number of candidate objectives. The last line gives an average percentage. TO and MO denote a timeout (set to 10 hours) and memory-out. Experiments were run on an Intel(R) Xeon(R) E-2176M with 32 GB RAM.

Notice that $M_{\mathrm{VA}}$ requires an entry point function and an initial context to start the analysis, meaning that objectives are identified as uncoverable with respect to these starting point and initial context. Value analysis can require a certain expertise to find optimal settings for a better analysis. As we want our tool to be as automatic as possible, we used default parameters. As for $M_{\mathrm{WP}}$, it does not require a global entry point but can be made more precise by providing contracts, i.e. pre- and postconditions and loops annotations. Again, in our experiments, annotations were not written for the same reason. Hence, an expert user can probably further improve the reported results. Similarly, using Frama-C plugins dedicated to generating ACSL annotations might improve these results as well. However, this demands some adaptations of our own implementation and is left for future work.

*Results.* Regarding **RQ1**, dataflow analysis techniques $M_{\mathrm{NA}}$ and $M_{\mathrm{Eq}}$ are very fast and very effective. $M_{\mathrm{NA}}$ detects an average rate of 41.5% (of all objectives) as inapplicable. $M_{\mathrm{Eq}}$ detects an average of 35.9% as equivalent. The rate of $M_{\mathrm{NA}}$ (between 8% and 85.1%) strongly depends on the example. Together, they identify a very significant number of polluting objectives (column $M_{\mathrm{NA,Eq}}$).

Regarding **RQ2**, $M_{\mathrm{VA}}$ performs really well on smaller programs, and becomes more expensive for larger examples (e.g. it runs out of memory for Monocypher). It detects between 27.1% and 69.6% (with an average of 41.5%). $M_{\mathrm{WP}}$ is by far the slowest method of detection. It takes up to 71m14s, times out on Monocypher, and detects almost no new uncoverables compared to $M_{\mathrm{VA}}$ (see below).

Regarding **RQ3**, while it is natural to expect benefits of a combination of different analyses, the results were somewhat surprising. Unlike in the previous work for other (non dataflow) criteria [3,11], the weakest precondition based technique brings only very slight benefits (see columns $M_{\mathrm{NA,Eq,VA}}$–$M_{\mathrm{NA,Eq,VA,WP}}$). We believe it is due to the complex nature of dataflow criteria where infeasibility is less likely to be detected by local reasoning. It is left as future work to study whether these results can be significantly improved using additional annotations. Using $M_{\mathrm{NA,Eq}}$ before $M_{\mathrm{VA}}$ or $M_{\mathrm{WP}}$ to filter out some objectives is clearly very efficient (and makes it possible for $M_{\mathrm{VA}}$, $M_{\mathrm{WP}}$ to terminate on the Monocypher example). Overall, our results show that the best combination appears to be $M_{\mathrm{NA,Eq,VA}}$, whereas executing $M_{\mathrm{WP}}$ in addition is very costly and detects at most 0.2% more sequences. When execution time is very limited, $M_{\mathrm{NA,Eq}}$ can be already very effective.

## 5 Conclusion and Future Work

Polluting test objectives can be an important obstacle to efficiently applying test coverage criteria, both for test generation or computation of coverage ratios. We adapted, implemented and evaluated several sound techniques to detect (a subset of) such objectives for dataflow criteria. Combining dataflow analysis

to detect inapplicable and equivalent objectives with value analysis to identify uncoverable ones appears to be the best trade-off for effective and fast detection. While this work provided a comparative analysis of the detection power of the analysis techniques, future work includes an evaluation of their results with respect to the *real set* of polluting objectives (or its overapproximation computed by replaying a rich test suite). Future work also includes a better support of the C language constructs (pointers and arrays) in the implemented tools, improving the analyses, notably $M_{\mathrm{WP}}$, by automatically generating additional annotations, as well as extending this study to subsumed (i.e. implied) test objectives and to other coverage criteria.

*Acknowledgements.* This work was partially supported by ANR (grant ANR-18-CE25-0015-01). We thank Sébastien Bardin, and the anonymous reviewers for valuable comments.

## References

1. Ammann, P., Offutt, J.: Introduction to Software Testing. Cambridge University Press (2017)
2. Rapps, S., Weyuker, E.J.: Data flow analysis techniques for test data selection. In: ICSE. (1982) 272–278
3. Bardin, S., Delahaye, M., David, R., Kosmatov, N., Papadakis, M., Traon, Y.L., Marion, J.: Sound and quasi-complete detection of infeasible test requirements. In: ICST. (2015) 1–10
4. Marcozzi, M., Delahaye, M., Bardin, S., Kosmatov, N., Prevosto, V.: Generic and effective specification of structural test objectives. In: ICST. (2017) 436–441
5. Frankl, P.G., Weyuker, E.J.: An applicable family of data flow testing criteria. IEEE Trans. Softw. Eng. **14**(10) (1988) 1483–1498
6. Rapps, S., Weyuker, E.J.: Selecting software test data using data flow information. IEEE Trans. Softw. Eng. **11**(4) (1985) 367–375
7. Harrold, M.J., Soffa, M.L.: Interprocedual data flow testing. SIGSOFT Softw. Eng. Notes **14**(8) (1989) 158–167
8. Weyuker, E.J.: The cost of data flow testing: An empirical study. IEEE Trans. Software Eng. **16** (1990) 121–128
9. Clarke, L.A., Podgurski, A., Richardson, D.J., Zeil, S.J.: A formal evaluation of data flow path selection criteria. IEEE Trans. Softw. Eng. **15**(11) (1989) 1318–1332
10. Su, T., Wu, K., Miao, W., Pu, G., He, J., Chen, Y., Su, Z.: A survey on data-flow testing. ACM Comput. Surv. **50**(1) (2017)
11. Marcozzi, M., Bardin, S., Kosmatov, N., Papadakis, M., Prevosto, V., Correnson, L.: Time to clean your test objectives. In: ICSE. (2018) 456–467
12. Marcozzi, M., Bardin, S., Delahaye, M., Kosmatov, N., Prevosto, V.: Taming coverage criteria heterogeneity with LTest. In: ICST. (2017) 500–507
13. Bardin, S., Kosmatov, N., Cheynier, F.: Efficient leveraging of symbolic execution to advanced coverage criteria. In: ICST. (2014) 173–182
14. Su, T., Fu, Z., Pu, G., He, J., Su, Z.: Combining symbolic execution and model checking for data flow testing. In: ICSE. (2015) 654–665
15. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. Formal Asp. Comput. **27**(3) (2015) 573–609
16. Kildall, G.A.: A unified approach to global program optimization. In: PoPL. (1973) 194–206
17. Prosser, R.T.: Applications of boolean matrices to the analysis of flow diagrams. In: Eastern Joint IRE-AIEE-ACM Computer Conference. (1959)