Annales Univ. Sci. Budapest., Sect. Comp. 36 (2012) 217–240

A DOMAIN BASED NEW CODE COVERAGE METRIC AND A RELATED AUTOMATED TEST DATA GENERATION METHOD

Dániel Leskó and Máté Tejfel

(Budapest, Hungary)

Communicated by Zoltán Horváth

(Received December 20, 2011; revised February 20, 2012;
accepted February 29, 2012)

Abstract. Since programmers write programs there has always been a need to analyze the correctness of these programs, which is mostly done by testing. However, testing our programs does not give any direct quality guarantee on them, because it highly depends on the used test data set. Numerous code coverage metrics can be applied to measure the quality of our test set, but the majority of them were primarily designed for imperative programs, and they rely mostly on control structures like branching and looping. The problem is that expression-heavy programs and functional programming languages normally do not have these structures. Hence, the corresponding code coverage metrics are unsuitable at least, but mainly useless for these kinds of programs.

In this paper we propose a new code coverage (*domain coverage*) metric, which is based on (arithmetic) expressions. The relations and effects among them are taken into account, such as some kind of semantics information about the programming language constructs. The paper also presents an automated test data generation method, which is related to *domain coverage*, and aims to reach the highest possible coverage ratio.

Key words and phrases: Code coverage metrics, domain, test data generation, symbolic execution.

1998 CR Categories and Descriptors: D.2.5, D.2.8.

Supported by ELTE TÁMOP-4.2.2/B-10/1-2010-0030 and the Hungarian National Development Agency (KMOP-1.1.2-08/1-2008-0002)

https://doi.org/10.71352/ac.36.217

1. Introduction

Since programmers write programs there has always been a need to verify the correctness of these programs. Today, the most commonly used method is testing, which is a very labor intensive work. There are estimations that a common software development project uses 50-70% of its total resources to achieve the desired software quality.

But simply the fact that we test our programs does not give any direct quality guarantee on them. The quality of the used test cases has to be determined so as to be able to properly evaluate the test results and use them for reasoning about the correctness of the tested program. We need to measure the percentage of the reached features of the program currently under testing. Based on this ratio, the usefulness of a particular test base can be easily judged.

Numerous code coverage metrics exist, such as statement coverage, decision coverage, path coverage, etc. The common point of these approaches are that they are working on some kind of syntactical/semantical representation of the analyzed program. These metrics are used in various coverage based testing tools [1] to measure the test case quality, and/or to recognize equivalence classes on the problem space. This latter one is a more advanced use of metrics. If two different test data yield exactly the same coverage result (been on the same path, exercised the same code pieces), then they are considered equivalent from the coverage metrics point of view; namely, they are in the same equivalence class. As a consequence, it is perfectly enough to generate one test value from each equivalence class, which will largely reduce the problem space.

The existing coverage metrics are more or less suitable for most kind of programs. The expression-heavy ones (e.g. programs written in Feldspar [12] or Matlab) or the functional ones (e.g. Haskell, Feldspar) are exceptions, because these typically lack branching and iteration statements (e.g. `if`, `for`, `while`). Of course, functional programs can express these notions by pattern matching or recursion. Additionally a relatively complex arithmetic expression can be written in *one* line. As a result of these properties, the existing code coverage metrics are unsuitable at least but mostly useless for these kinds of programs.

The problem with existing coverage metrics is that they are concentrating mostly on control structures (like branching and looping) and statements. The expressions are considered only in those cases when they have an effect on the control flow. However, the relations and effects between sub-expressions are never examined.

In the rest of this paper, we propose a new code coverage metric (Section 2), which solves the previously mentioned shortage by starting on the expression level and then going upwards to the control layer. The next section describes a related test data generation method, which could reach the highest possible domain coverage ratio. Section 4 explains briefly how to put the theory into practice, then we finish up with related work (Section 5), conclusion and future work (Section 6)

2. Code coverage metrics/measurement

Testing methods and code coverage metrics should always come hand in hand, because a coverage metric is simply not applicable without a given test set. On the other hand, a test set without its coverage information is basically useless, when we try to reason about correctness. The only exception i, when the whole input domain is in the test set, but this is very rarely feasible for interesting, nontrivial programs.

Since computer programs exist, there has been a hard-to-reach desire, namely, the programs should be bug free. In order to fulfill this desire numerous testing approaches were developed. However, testing could not give us any direct quality information about the program. A code coverage metrics has to be used to collect information about which parts of the source code and which program features were reached by the applied test set. The result can be represented as percentage value between 0% and 100%, where the higher is the better.

It would be desirable, if 100% coverage could mean that the program is tested with all possible inputs and in all possible ways. In this hypothetical case, the program would be not only validated by testing, but verified too, since it was used with every possible input. Currently, we are far from that, and also the strength and reliability of a specific coverage percentage depend highly on the used coverage metrics. So using simply a coverage rate (without knowing the used metrics) as a quality measurement is unsafe, and could even give the feeling of false safety, which is even worse than having no clue about the quality of the used test set.

From another point of view, a code coverage metrics specifies equivalence classes over the input data space of the tested program. This means that we need only one test value per class to reach the same coverage. Ideally, a class should be a group of such input data that use exactly the same parts of the tested code in exactly the same way. However, this highly depends on the specific coverage metrics and, unfortunately does not hold generally.

2.1. Example

The following small program (written in WHILE language [3]) will be used as a running example in this paper. The program is composed of 10 programming language constructs (see boxes in Figure 2). Nine of them are expressions, so our example is really expression heavy. Please note that the code fragment is buggy, and the x=0 case will cause failure.

```
if ((x - 1 < 10) && (x mod 2 = 0)) then
       res := 100 div x
    else
       res := 10 * x
```

Figure 1. Example program in WHILE

Figure 2 shows the original syntax tree of the example code-fragment. Later, there will be two modified versions of it.

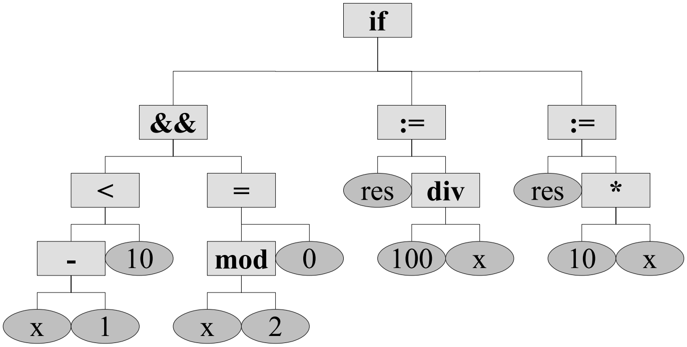

Figure 2. Original syntax tree of the example

2.2. Thoughts about existing code coverage metrics

In this section, we will examine some well-known and commonly used code coverage metrics. A short informal definition will be given mostly from the point of view of our example in Figure 1 Also such test data sets will be given for every metric, which reaches 100% coverage. Most of the metrics were taken from Beizer's paper [9], and some of them came from the industry [10, 11].

Statement coverage (also known as line coverage): The basic block coverage is very similar, with the difference that there the unit of code measured is not a statement, but a sequence of non-branching statements. Statement coverage reports whether each executable statement is encountered. Full coverage can be reached with the following two test values: x=10; x=11.

Decision/branch coverage: The metric reports whether logical expressions are tested in control structures (such as branching or looping) evaluated to both true and false. The full coverage can also be reached with two test values: `x=10; x=11`.

Condition coverage (or condition operand coverage): Condition operand coverage improves the thoroughness of decision coverage by testing each operand of decision conditions with both true and false values, rather than just the whole condition. Full coverage with two smartly selected test values: `x=10; x=11`.

This metric has an even more thorough variant, named Multiple condition coverage (and also known as condition operator coverage), which looks at the various combinations of Boolean operands within a condition. Each Boolean operator – within a condition – has to be evaluated four times, with the operands taking each possible pair of combinations of true and false. For this multiple version, we need four test values: `x=9; x=10; x=11; x=12`.

Path coverage (also known as predicate coverage): The metric reports whether each of the possible paths in each function have been followed. A path represents the flow of execution from the start of a method to its exit. A method with N decisions has 2^N possible paths, and if the method contains a loop, it may have an infinite number of paths. Full coverage can be reached with `x=10; x=11`.

Fortunately, we can use the Cyclomatic Complexity [13] to reduce the number of paths we need to cover. It directly measures the number of linearly independent paths through a program's source code. Cyclomatic complexity is computed using the control flow graph of the program: the nodes (N) of the graph correspond to indivisible groups of commands of a program, and a directed edge (E) connects two nodes if the second command might be executed immediately after the first command. Formula: $M = E - N + 2P$, where P is the number of connected components.

The number of linearly independent paths through a method is generally smaller than the total number of paths, although every possible path can be formed by combining several linearly independent paths. The cyclomatic complexity value is the maximum number of test cases needed to get 100% branch coverage, and the minimum number of test cases needed to exercise every path through a method, so it provides a good way to tell how well a method is tested.

Relational operator coverage: This one is not so commonly known and used. It could be considered as an accessory and also a little bit exotic metric rather than a regular one. It reports whether boundary situations occur with relational operators (`<`, `<=`, `>`, `>=`). The hypothesis is that boundary test cases find off-by-one mistakes and uses of the wrong relational operators such as `<` instead of `<=`. Full coverage with `x=10; x=11` can be reached.

Data flow based metrics: Most of the previously mentioned metrics are based on control flow. The strength of coverage metrics can be enhanced by examining the data flow (e.g. how variables are defined and used in the program).

Frankl and Weyuke [14] proposed numerous data flow based coverage metrics, which were all based on the idea that, in principle, for each statement in the program we should consider all possible ways of defining the variables used in the statement. At the end, all of these metrics select a set of paths which satisfies their criterion.

Other code coverage metrics: Of course, several other code coverage metrics exist (such as function, call, loop, Linear Code Sequence and Jump (LCSAJ), ...), which were not mentioned earlier, because they are not really relevant to our example code (see Section 2.1), or they are just a mutated version of a presented one.

2.3. Revealing the problem

The previous subsection also showed that although it is easy to create a program that scores 100% code coverage while the result is still buggy. Currently we are not aware of such metric that would *not* reach full coverage unless there is a specific test case (in our case `x=0`), which results "division by zero" error.

The main problem with the example in Section 1 is that it is not sophisticated enough for the problem that it aims to solve. Specifically there should be another condition before the division. These types of bugs – those that can be fixed by adding new code lines – are sometimes called faults of omission. Unfortunately, they are quite common in real life, as numerous studies have shown. One of them is from Glass [4], in which the author describes faults of omission as "code not complex enough for the problem". Faults of omission are not just unchecked status returns: they include missing conditions in "if" statements, missing "catch" statements for exceptions and other, more complex cases.

In hardware design (especially for microprocessors), there are two notions: data-path layer and control-path layer. The data-path layer performs the arithmetic operations and the control-path layer tells the data-path layer, memory, and I/O devices what to do according to the instructions of the program. These notions could be easily used for programs written in high-level programming languages as well. The control-path layer would contain the control structures (branching, looping), function and procedure calls (essentially every statement). The data-path layer would be the lower one, containing arithmetic and logical expressions.

The problem of the previously shown code coverage metrics originates in the fact that they are relying almost entirely on the control-path layer of the tested program. In the following subsection, we will propose a new coverage metric, which is based on the data-path layer, and on top of that could handle the control-path layer too.

2.4. Domain coverage

A computer program always relies on lower layers, such as libraries, core language constructs, or another programs written by other programmers. To make sure that our program is bug free and works as it should, we have to see and test the whole picture, not just our few lines of code.

In most cases, the lower layers are well tested, and also our program can be tested with various tools and coverage metrics, but the interaction between them are the weakest link. Testing these interactions basically means a low level integration testing between our program and some core language constructs. However, this approach is not used, mostly because it would require some support from the programming languages, which does not exist yet.

In order to be able to perform thoroughly the previously mentioned low level integration testing, we need such code coverage metric which will produce a reliable test quality information. As shown earlier, the currently used metrics fail to detect the ”possibly non-complete handling of an interaction” kind of errors. We will present a new coverage metric, which aims to detect if a test set for the previously mentioned low level integration testing is not as thorough as it should be.

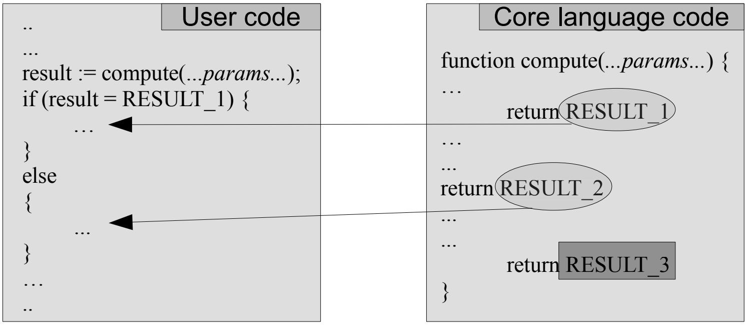

Figure 3. Example: ”possibly non-complete handling of an interaction”

Figure 3 shows an example, where the result of a function (`compute`) depends on some incoming data (`...params...`), and there are three possible outcomes. However, the user code has only one branch, which distinguishes `RESULT_1` from the other cases. On one hand, this could be on purpose, but on the other hand, maybe the programmer was sloppy, or actually he/she never realized that there are three possible outcomes. So a trustworthy code coverage metric should account all three outcomes to report 100% coverage. The problem is that this would require some information about the semantics of the function (`compute`).

The rest of this section will propose a new coverage metric, which mainly targets the previously mentioned interaction part, not just strictly the source code. The new metric does not try to find out whether there is a branch or some handling code for every possible outcome of a function, because the lack of these could be the programmer's intention. Rather we try to check that the test data set is so thorough that the outcome of the function covers every interesting part of the codomain.

Remark 2.1. In the followings we assume that an ordering exists (or it is easily definable) on every used variable's domain.

Definition 2.1. A *sub-domain* is a set of elements, more precisely it is a part of a specified variable's domain.

Notation: $(a, b, c, d, e \in \mathbb{N})$

- `[a]` means `a` as a single value
- `[a..b]` means (interval from `a` to `b`)
  - $(a \leq b)$ $\forall x \in \mathbb{N}:$ $x \in \texttt{[a..b]} \iff a \leq x \leq b$
  - $(a > b)$ $\forall x \in \mathbb{N}:$ $x \in \texttt{[a..b]} \iff (a \leq x \leq \mathrm{MAX}) \vee$ $\vee(\mathrm{MIN} \leq x \leq b)$
- `[a,b..c]` means (`a` is the first element, `b` is the second, `c` is the end-bound)
  - $(a \leq b \leq c)$ $\forall x \in \mathbb{N}:$ $x \in \texttt{[a,b..c]} \iff$ $(x \bmod (b-a) = a \bmod (b-a)) \wedge (a \leq x \leq c)$
  - $(a > b > c)$ $\forall x \in \mathbb{N}:$ $x \in \texttt{[a,b..c]} \iff$ $(x \bmod (b-a) = a \bmod (b-a)) \wedge ((a \leq x \leq \mathrm{MAX}) \vee (\mathrm{MIN} \leq x \leq c))$

Example: `[10,12..20]` = {10, 12, 14, 16, 18, 20}

Definition 2.2. The *domain* of a specified variable is a partitioning of its codomain. Practically, it is a set of sub-domains, where C is the codomain, D is the corresponding domain, S is a sub-domain, and $\forall x \in \mathrm{C} : \exists! \mathrm{S} \in \mathrm{D} : x \in \mathrm{S}$.

Remark 2.2. A sub-domain could have infinite elements, because the previously mentioned MIN and MAX are not actual values by all means. They could be just symbols. As a result of this, our model is not limited to finite domains.

Remark 2.3. In cases where there is not enough information about the codomain, and therefore we could not or would not split it into sub-domains, we will use the $\square$ sign. It represents such a domain which has only one sub-domain, containing the complete codomain.

In the following definition and also in the rest of this paper, we consider each and every programming language construct as a function. For example, the `if..then..else..` construct will be thought as a function with three arguments.

Definition 2.3. A *behavior* is a mapping from a programming language construct (represented as a function) to a tuple of domains, where the arity of the function and the arity of the tuple are equal. It assigns *initial domains* to every programming language construct.

Notation: (`behavior` : $\mathcal{F} \to \mathcal{D}$; where $\mathcal{F}$ is a set of every possible programming language construct and operator; $\mathcal{D}$ is a set of domain tuples with varying arity)

- `behavior`$(\star)$: $-$
- `behavior`$(\star \to \star)$: $\mathbf{D}$
- `behavior`$(\star \to \star \to \star)$: $\mathbf{D} * \mathbf{D}$
- `behavior`$(\star \to \star \to \star \to \star)$: $\mathbf{D} * \mathbf{D} * \mathbf{D}$

Remark 2.4. Mostly not the whole behavior will be required, just one domain out of it. To make it more handy, the $f \in \mathcal{F}, n \in \mathbb{N} : behavior(f)(n) : \mathcal{F} \to \mathbb{N} \to \mathcal{D}$ notation will be used. This means that we only need the n^{th} element of the resulted tuple.

Definition 2.4. An *initial domain* is a domain with a specific role. Each sub-domain of an initial domain groups such values which behave similarly. Basically it represents equivalence classes – according to the semantics of the construct – on the codomain of the current language construct.

Ideally the initial domains of operators and other language constructs should be given by the language designer.

The `behavior` function and the corresponding initial domains shown in Figure 4 will be used in the rest of this paper. The design of these initial domains is completely the tester's duty and responsibility. It basically reflects the tester's opinion about which partitions of each programming language construct's domain is important and interesting. For example, we think 0 as a special value according to Figure 4. This `behavior` function intends to check the program also in mathematically incorrect cases ([0] for div).

(0) behavior($variable$) $= \square$

(1) behavior(`+`) $=$ ({[MIN..(-1)], [0], [1..MAX]} $*$ {[MIN..(-1)], [0], [1..MAX]})

(2) behavior(`-`) $=$ ({[MIN..(-1)], [0], [1..MAX]} $*$ {[MIN..(-1)], [0], [1..MAX]})

(3) behavior(`*`) $=$ ({[MIN..(-1)], [0], [1..MAX]} $*$ {[MIN..(-1)], [0], [1..MAX]})

(4) behavior(`div`) $=$ ({[MIN..(-1)], [0], [1..MAX]} $*$ {[MIN..(-1)], [0], [1..MAX]})

(5) behavior(`mod`) $=$ ({[MIN..(-1)], [0], [1..MAX]} $*$ {[MIN..(-1)], [0], [1..MAX]})

(6) behavior(`=`) $= \begin{pmatrix} (\{[true],[false]\} * \{[true],[false]\}) \text{ for logical expr.} \\ (\{[\text{MIN..MAX}]\} * \{[\text{MIN..MAX}]\}) \text{ for arithmetic expr.} \end{pmatrix}$

(7) behavior(`<`) $= \begin{pmatrix} (\{[true],[false]\} * \{[true],[false]\}) \text{ for logical expr.} \\ (\{[\text{MIN..MAX}]\} * \{[\text{MIN..MAX}]\}) \text{ for arithmetic expr.} \end{pmatrix}$

(8) behavior($\neg$) $=$ {[true], [false]}

(9) behavior($\wedge$) $=$ ({[true], [false]} $*$ {[true], [false]})

(10) behavior(`skip`) $= -$

(11) behavior(`:=`) $= \square$

(12) behavior(`;`) $= \square * \square$

(13) behavior(`if`) $=$ ({[true], [false]} $* \square * \square$)

(14) behavior(`while`) $=$ ({[true], [false]} $* \square$)

Figure 4. Behavior and initial domains for the WHILE language

Definition 2.5. A *behavior class* is related to the notion of sub-domain. Both aim to group such data which exercise a code fragment in exactly the same way. The difference is that sub-domains are for single variables, while behavior classes are for single programming language constructs. A behavior class is a tuple of sub-domains.

The `behavior` function results a tuple of domains. $\mathcal{B}_p$ is a set of every possible behavior class (the previously mentioned tuple is interpreted as a Cartesian product) for a p specific language construct or operator. $b \in \mathcal{B}_p : b$ is a behavior class.

$$\mathcal{B}_p = \bigtimes_{1 \leq i \leq arity(p)} \texttt{behavior}(p)(i)$$

Remark 2.5. In those cases when a constant value (which appears in the source code as a parameter) is involved in a computation, we will always perform *currying* on it. For example, if `x+1` is given, then we will use `(+1)` : : $\mathcal{N} \rightarrow \mathcal{N}$, instead of `(+)` : $\mathcal{N} \rightarrow \mathcal{N} \rightarrow \mathcal{N}$. This approach will significantly reduce the number of not coverable behaviors.

Figure 6 shows some behavior classes for the example code (shown in Figure 3). Each rectangular with dashed lines marks a behavior class.

Domain coverage metric (informal definition): The coverage is strongly connected to the evaluation process of a program. The interpretation of a program starts from the bottom of its tree-like representation, and proceeds to the top by evaluating the sub-expressions. Basically, the idea is to record the result of every sub-expression and statement during the evaluation phase. This would result a modified syntax tree (see Figure 5), where the structure is the same, but the operator nodes became a simple value. These values can be used to check which behavior classes were covered.

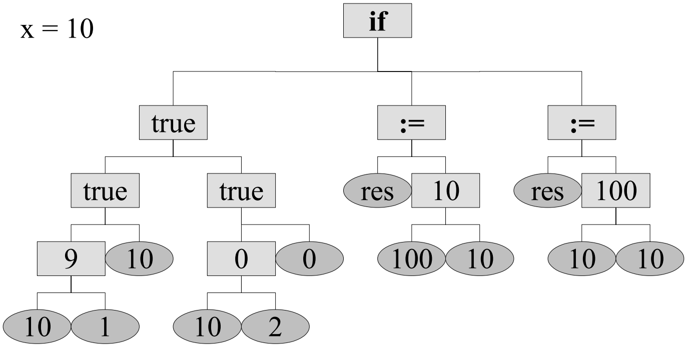

Figure 5. Modified syntax tree of the example, showing internal results

The smallest unit is a sub-domain, which is covered if it was reached at least once during the testing. A behavior class is a tuple of sub-domains and we consider it covered if its every sub-domain was covered by the same test data. An f programming language construct is covered if $\forall b \in \mathcal{B}_f : b$ behavior class was covered. A program ($\mathcal{P}$) can be considered as a set of programming language constructs, and therefore it is covered if $\forall f \in \mathcal{P} : f$ was covered.

Definition 2.6. A *covered* behavior is a function which gets a $p \in \mathcal{P}$ programming language construct and a test data (*par*) while results the covered behavior class.

Notation:
`cover` $: par \to p \to b \quad$ where $p \in \mathcal{P}, b \in \mathcal{B}_p$

Figure 6 shows covered behaviors for the example code (shown in Figure 3) in the case of `x=10` input data. Each dashed-line rectangular shows a behavior class containing the covered sub-domains.

To sum up, the domain coverage metric has two levels (lower-upper), the notion of behavior is the border-line between them. Below that, the sub-domains are taken dependently, which means that the metric handles the parameters of a programming language construct in a dependent way. The parts above the behavior notion are taken independently, which means that the statements and other programming constructs of a specific program are handled independently by the metric.

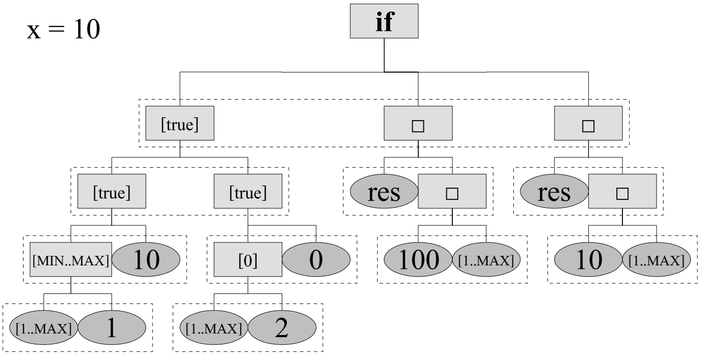

Figure 6. Modified syntax tree of the example, showing the covered behaviors

Definition 2.7. Domain coverage metric : $\frac{\text{covered behavior classes}}{\text{total number of behavior classes}}$

where

- *params* is the set of input test data
- *covered behavior classes* $= \sum_{p \in \mathcal{P}} \texttt{size}(\bigcup_{par \in params} \texttt{cover}(par)(p))$
- *total number of behavior classes* $= \sum_{p \in \mathcal{P}} \texttt{size}(\mathcal{B}_p)$
- P is the tested program, represented as a set of programming constructs and operators
- $\texttt{size}(\mathcal{B}_p)$ means the number of behavior classes of $\mathcal{B}_p$

Relation to other code coverage metrics: Domain coverage subsumes decision / branch coverage – and as a consequence statement coverage too – because the `behavior` function ((13),(14) in Figure 4) assigns such initial domains for `if` and `while`, where the logical values are split up into sub-domains.

Simple condition coverage is subsumed by domain coverage, since every logical operator has such initial domains ((6)-(9) in Figure 4), where the *true* and *false* are separated into different sub-domains. As a result of this, every logical expression has to be evaluated both *true* and *false* in order to reach 100% domain coverage, but this is also enough for 100% condition coverage.

The multiple version accounts the logical expressions dependently, and therefore that is a stronger metric, which cannot be subsumed generally by domain coverage.

Path coverage, data flow coverage and also relational operator coverage are hard to compare with domain coverage, since they use a quite different approach, but it is safe to say that generally domain coverage cannot subsume either of them. The ”normal” path coverage and also the cyclomatic complexity based one are not designed and not fit well for the programs we aim to test. The almost complete lack of branching and looping statements makes them really weak. Data flow coverage adapts really well, because expression heaviness does not lower the size of the data flow graph. However, its main target (variable definition-usage relations) is quite far from the idea of domain coverage which originates in the semantics and behavior classes of the programming language constructs.

Note that the power of domain coverage metric strongly relies on the given `behavior` function. In this comparison, and in Figure 4 we tried to use it in a smart way. However, a not so mature `behavior` function could result a much weaker metric, which even can be subsumed by statement coverage.

3. Automated test data generation

Having a code coverage metric which tells that our current test data set is not good enough is nice, but generally not so useful, because it does not say anything about how to improve the test set to reach higher coverage.

This section presents a new test data generation approach close to coverage based test data generation. The main difference is that our approach does not use the coverage metric as-is, instead it only borrows some notion and function definition from the metric and uses them with a slightly different purpose.

To be more specific, the coverage metric used the `behavior` function to check that, which behavior classes were covered by the internal results during execution with real input test data, while test data generation uses the `behavior` function to perform a symbolic evaluation of the current program. During this evaluation the domains of each variable of the program are refined (split into more sub-domains) and propagated upwards. The result of this symbolic evaluation is a refined domain for every input argument. Using this information, we can easily generate such test data sets which will reach 100% domain coverage.[1]

[1] If there are no infeasible combinations, otherwise the ratio is a bit less, but still the highest possible

Definition 3.1. (Refining a domain.) Let us take two domains, $d_1, d_2 \in \mathbf{D}$, and refine them with each other: $\texttt{refine}(d_1 \cup d_2)$

where

$$\texttt{refine(d)} = \begin{cases} \texttt{refine}((d \setminus \{s_1, s_2\}) & \\ \qquad\qquad \cup\, \texttt{split}(s_1, s_2)) & \exists s_1, s_2 \in d, \exists e \in s_1 : e \in s_2 \\ d & \forall s_1, s_2 \in d, \nexists e \in s_1 : e \in s_2 \end{cases}$$

$$\texttt{split}(s_1, s_2) = \{[x|x \in s_1 \wedge x \notin s_2], [y|y \in s_1 \wedge y \in s_2], [z|z \notin s_1 \wedge z \in s_2]\}$$

The `refine` function takes a set of sub-domains, which is usually not a domain (has overlapping sub-domains), and refines it to a correct domain by splitting and eliminating sub-domains. The `split` takes two – usually – overlapping sub-domains, and returns a set of sub-domains without overlapping.

A domain is basically a complete partitioning of its codomain. The union or refinement of two domains (of the same codomain) could be imagined as follows. Interpret each sub-domain starting and ending point as a delimiter, and build a new domain, which contains only such sub-domains that start from a delimiter and end at the next one.

3.1. Target oriented domain refinement

The main idea of our test case generation method is to do a bottom → top symbolic evaluation, and during this process, gather such information about the processed `behavior`s, which guarantees that the generated test data set will cover every reachable behavior class.

One step of this recursive symbolic evaluation means the followings:

- Given:
  - $p \in \mathcal{P}$ programming language construct or operator
  - $\texttt{arity}(p)$ number of domains (they will be referred as input domains), which types are match to p's signature
  - target domain, which is the expected domain for the result of p
- Tasks:
  1. Refine the input domains according to `behavior`(p).
  2. Refine further the input domains in order to guarantee that if a test set – generated from the knowledge of the refined input domains – covers p's every behavior class, then the result of p will always cover the target domain.

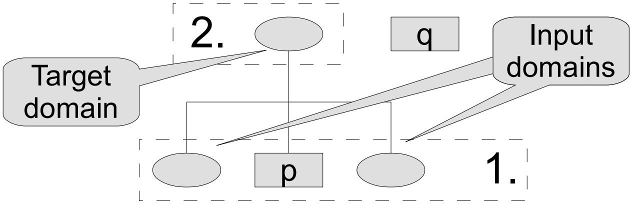

Figure 7. Outlining one step of the symbolic evaluation

Figure 7 outlines one step of the used symbolic evaluation, where $p, q \in$ $\in \mathcal{P}$. According to the figure, p has exactly two arguments and q has at least one. The number of arguments are orthogonal to the hereinafter described method. Losing a bit of abstraction in this example hopefully results a better understanding and a more comprehensible figure.

The input domains are either coming from the symbolic evaluation of lower structures (sub-expressions), or they are the □ domain in the case the corresponding argument of p is a constant or a variable.

Note that a domain is always associated with a variable, so when we talk about a domain we always mean that specific domain which is associated with the current variable name. Generally we have an associated array of domains where the variable names are the keys. The elements (domains) of this array are updated (refined) during the symbolic evaluation of an expression and then the whole array is propagated upwards.

The first part (marked as 1. in Figure 7) is easy, we only have to refine each input domain with a corresponding `behavior`(p) (i) domain (in our example: i $\in \{1, 2\}$). This ensures that every behavior class of p will be covered, if the used test data set is generated according to the refined domains (details of the generation in subsection 3.2).

The second part (marked as 2. in Figure 7) is much more tricky. The input domains – which are the domains of p's arguments – have to be refined again. The goal is to refine the input domains in a way which will ensure that if p is executed (with some generated test data based on the refined domains), then the set of p's results will completely cover `behavior(q)(i)` (in our example: i $\in \{1\}$). We should use such refinement strategy that keeps the number of sub-domains low, because this will also keep the size of the generated test set low. Unfortunately, there is no general strategy for this task, but the following technique works in common cases:

1. Check whether refinement is needed or not

2. If the input domain is close to the target domain (according to the number and boundaries of the sub-domains), then refine this input domain with

the target domain. After that, refine the rest of the domains with such sub-domain that contains only the identity element (respectively to p)

3. User defined, specific strategies for p and maybe even for different kinds of domains

Please note that the previous (1) or (2) could always solve the problem, and mostly their efficiency is not so bad. The efficiency of a refinement strategy could be measured by the number of newly introduced sub-domains. We consider a strategy efficient if it tries to keep this number low.

Informal description of $\texttt{refine\_to\_target}_p(arr, target\_domain)$ function, where $target\_domain \in \mathbf{D}$, and arr is an associative array of domains (the keys are the corresponding variables).

- $p \in \mathcal{P}$, p is a language construct, $\mathcal{P}$ is the program under testing, represented as a set of language constructs
- In practice we will need such version of p which
  - works on domains instead of single values (map to each sub-domain's every element)
  - has an inverse semantics of p (e.g. `+` → `-`)

  Example: In the case of `x-1`, the p will be `(-1)` (because we curry the constants), and the inverse semantics would mean that we increment each value by one.
- The arr associative array contains a domain for every arg_p(i) where 1 ≤ ≤ i ≤ arity(p)
- The result of this function is an associative array of domains (based on arr), where the domains for arg_p are modified by the inverse p and refined, if needed (depending on the given $target\_domain$)

Let us take the following small example, where p is `(-1)`, the arr contains a domain for `x`, and the target domain is [MIN..MAX].
$\texttt{refine\_to\_target}_{(-1)}$([x → {[MIN .. (-1)], [0], [1 .. MAX]}, [MIN .. MAX]])

Firstly we have to apply the previously described version of p (inverse semantics, works on domains) to the elements of the associative array. This propagates our current knowledge upwards in the syntax tree. The result of this step is [x → {[(MIN+1) .. 0], [1], [2 .. MIN]}]. The second step is to refine the result with the given target domain. Currently the target domain is the □ domain, which means that the `refine` function will not change anything, so the overall result will be the result of the first step.

3.2. Symbolic evaluation and test data generation

A special bottom-up symbolic evaluation will be used to refine, propagate and gather domain information about the input variables of the evaluated code. This domain information is a basis of our guided/restricted test data generation, where we pick one value randomly from each sub-domain.

A domain is always connected to a variable; therefore, during symbolic evaluation we could represent them in an associative array. The union of such arrays is an array that contains every occurrent domain. If a variable appears in more then one array, than the union/refinement of the corresponding domains will be used in the resulted associative array.

The following rules will be used for the symbolic evaluation. The 'initial' is only used for the first occurrence of a variable, in every other case the second, 'standard' rule is used. We always start the evaluation by calling `sym_eval` for the topmost $p \in \mathcal{P}$ programming construct of the analyzed program, and with $\Box$, as a target domain.

$$\texttt{sym\_eval}\colon p \to d \to arr \qquad (p \in \mathcal{P}, d \in \mathbf{D}, arr \text{ is an assoc. array of domains})$$

$$\texttt{sym\_eval}(\textit{input variable}, \Box) = [\textit{input variable} \to \Box] \qquad \text{(initial rule)}$$

$$\texttt{sym\_eval}(p, \textit{target\_domain}) = \texttt{sndP} \qquad \text{(standard rule)}$$

where

$$\texttt{fstP} = \bigcup_{i \in 1..arity(p)} \texttt{refine}(\texttt{sym\_eval}(arg_p(i), \texttt{behavior}(p)(i)), \texttt{behavior}(p)(i))$$

$$\texttt{sndP} = \texttt{refine\_to\_target}_p(\texttt{fstP}, \textit{target\_domain})$$

Firstly the function traverses the program's syntax tree downwards. The task of this traversal is to set the *target_domain* for every argument. The second step is to traverse upwards, and do the main task: compute and propagate the refined domains.

Number of input arguments: According to the previously described test data generation method, the size of the yielded test data set considerably depends on the number of input arguments.

The result of the symbolic evaluation is a refined domain for every input variable of the program under testing, then we take the Cartesian product of the domains, which will result every possible behavior class. The last step is to generate one test data from every behavior class.

In practice, normally we would not want to take the Cartesian product, because this would mean that the size of the test set is exponential to the number of input arguments. In this way, we would take dependently such arguments (or occurrences of arguments) which are independent in the tested program, and would also generate numerous infeasible combinations.

By tracking the *age* (when it was created by refinement, during the symbolic evaluation) and history (how it was refined/split during the evaluation phase) of each sub-domain, we could gather extra information. This could be used to create a smarter test data generator, which takes into account only such behavior classes from the previously mentioned Cartesian product, where the sub-domains are really dependent.

3.3. Example in details

In this subsection we will perform manually the symbolic evaluation to show how it works. The code fragment of our running example (Figure 1) will be used. The method will be presented thoroughly on the `x-1` sub-expression; the later parts will be rougher.

`sym_eval`(`x-1`, [MIN..MAX]) = `snd_p`

The target domain of `x-1` *is* [MIN..MAX]*, which is coming from the* `<` *operator.*

`fst_p` =

(substituting into the formula: `x` *is the first argument the behavior is taken from* `-`*'s first argument)*

= `refine`(`sym_eval`(x,{[MIN..(-1)], [0], [1..MAX]}), {[MIN..(-1)],[0],[1..MAX]})

(calculating the inner `sym_eval` *call by using the initial rule)*

= `refine`([x → {[MIN..(-1)], [0], [1..MAX]}], {[MIN..(-1)], [0], [1..MAX]})

(performing the refinement; this was marked as 1. in Figure 7)

= [x → {[MIN..(-1)], [0], [1..MAX]}]

`snd_p` =

(substituting, and currying the constant, the p is `(-1)`*)*

= `refine_to_target`$_{(-1)}$(`fst_p`, [MIN .. MAX])

(inlining `fst_p`*)*

= `refine_to_target`$_{(-1)}$([x → {[MIN..(-1)], [0], [1..MAX]}, [MIN..MAX]])

(performing `refine_to_target` *marked as 2. in Figure 7)*

= [x → {[(MIN+1) .. 0], [1], [2 .. MIN]}]

3.3.1. Short overview of the evaluation

- `x-1`:
  - $\mathrm{arr}_1 = [\mathrm{x} \rightarrow \Box]$
  - $\mathrm{arr}_1 \cup [\mathrm{x} \rightarrow \texttt{refine}(\Box, \{[\mathrm{MIN}..(\text{-}1)], [0], [1..\mathrm{MAX}]\})]$
  - $\mathrm{arr}_1 = \texttt{refine\_to\_target}_{(-1)}\ (\mathrm{arr}_1,\{[\mathrm{MIN}..\mathrm{MAX}]\}) = [\mathrm{x} \rightarrow \{[(\mathrm{MIN}+1)..0], [1], [2..\mathrm{MIN}]\}]$
- `...<10`:
  - $\mathrm{arr}_1 \cup [\mathrm{x} \rightarrow \texttt{refine}(\{[\mathrm{MIN}..(\text{-}1)], [0], [1..\mathrm{MAX}]\}, [\mathrm{MIN}..\mathrm{MAX}])]$
  - $\mathrm{arr}_1 = \texttt{refine\_to\_target}_{(<10)}\ (\mathrm{arr}_1,\{[\mathrm{true}], [\mathrm{false}]\}) = [\mathrm{x} \rightarrow \{\ [(\mathrm{MIN}+1)..0], [1], [2..9], [10..\mathrm{MIN}]\}]$
- `x mod 2`:
  - $\mathrm{arr}_2 = [\mathrm{x} \rightarrow \Box]$
  - $\mathrm{arr}_2 \cup [\mathrm{x} \rightarrow \texttt{refine}(\Box, \{[\mathrm{MIN}\ ..\ (\text{-}1)], [0], [1\ ..\ \mathrm{MAX}]\})]$
  - $\mathrm{arr}_2 = \texttt{refine\_to\_target}_{(\mathrm{mod}\ 2)}\ (\mathrm{arr}_2,\{[\mathrm{MIN}..\mathrm{MAX}]\}) = [\mathrm{x} \rightarrow \{\ [\mathrm{MIN}..(\text{-}1)], [0], [1..\mathrm{MAX}]\}]$
- `...=0`
  - $\mathrm{arr}_2 \cup [\mathrm{x} \rightarrow \texttt{refine}(\{[\mathrm{MIN}..(\text{-}1)], [0], [1..\mathrm{MAX}]\}, \Box)]$
  - $\mathrm{arr}_2 = \texttt{refine\_to\_target}_{(=0)}\ (\mathrm{arr}_2,\{[\mathrm{true}], [\mathrm{false}]\}) = [\mathrm{x} \rightarrow \{\ [(\mathrm{MIN}+1), (\mathrm{MIN}+3)..0], [(\mathrm{MIN}+2)..(\mathrm{MIN}+4)..0], [1], [2, 4..\mathrm{MIN}], [3, 5..\mathrm{MIN}]\}]$
- `...&&...`
  - This construct does not result further refinements, therefore we just collect the results.
  - $\mathrm{arr}_1 \cup \mathrm{arr}_2 = [\mathrm{x} \rightarrow \{[(\mathrm{MIN}+1), (\mathrm{MIN}+3)..0], [(\mathrm{MIN}+2), (\mathrm{MIN}+4)..0], [1], [2, 4..9], [3, 5..9,] [10, 12..\mathrm{MIN}], [11, 13..\mathrm{MIN}]\}] = \mathrm{arr}_{1,2}$
- `100 div x`:
  - $\mathrm{arr}_3 = [\mathrm{x} \rightarrow \Box]$
  - $\mathrm{arr}_3 \cup [\mathrm{x} \rightarrow \texttt{refine}(\Box, \{[\mathrm{MIN}..(\text{-}1)], [0], [1..\mathrm{MAX}]\})]$
  - $\mathrm{arr}_3 = \texttt{refine\_to\_target}_{(100\ \mathrm{div})}\ (\mathrm{arr}_3,\{\Box\}) = [\mathrm{x} \rightarrow \{[\mathrm{MIN}..(\text{-}1)], [0], [1..\mathrm{MAX}]\}]$

- `10 * x`:
  - $\mathrm{arr}_4 = [\mathrm{x} \to \square]$
  - $\mathrm{arr}_4 \cup [\mathrm{x} \to \texttt{refine}(\square, \{[\mathrm{MIN}..(\text{-}1)], [0], [1..\mathrm{MAX}]\})]$
  - $\mathrm{arr}_4 = \texttt{refine\_to\_target}_{(10\ *)}\ (\mathrm{arr}_4,\{\square\}) = [\mathrm{x} \to \{[\mathrm{MIN}..(\text{-}1)], [0], [1..\mathrm{MAX}]\}]$
- `res:=... ;    res:=...`
  - These constructs do not necessitate further domain refinements.
- `if ... ... ...`
  - This construct does not result further refinements, therefore we just collect the results.
  - $\mathrm{arr}_{1,2} \cup \mathrm{arr}_3 \cup \mathrm{arr}_4 = [\mathrm{x} \to \{[(\mathrm{MIN}+1), (\mathrm{MIN}+3)..(\text{-}1)], [(\mathrm{MIN}+2), (\mathrm{MIN}+4)..(\text{-}1)], [0],[1], [2, 4..9], [3, 5..9], [10, 12..\mathrm{MIN}], [11, 13..\mathrm{MIN}]\}] = \mathrm{arr}_{1,2,3,4}$

Outcome: The result of the symbolic evaluation is the following associative array, which – in our case – contains the refined domain of `x`:
{[(MIN+1), (MIN+3) .. (-1)], [(MIN+2), (MIN+4) .. (-1)], [0],[1], [2, 4 .. 9], [3, 5 .. 9], [10, 12 .. MIN], [11, 13 .. MIN]}

According to the previous result, the following test set would reach 100% domain coverage: `x` ∈ {-2, -1, 0, 1, 2, 3, 10, 11}

4. Domain coverage in practice

It is obvious that the presented code coverage metric and the annotated test data generation method are not lightweight solutions: they require a massive support from the used programming languages. Therefore, it is unlikely that the presented methods will be reachable soon in major main-stream programming languages like Java, C++ or Haskell. However, this is not the case with embedded domain specific (EDSL) languages, which are nowadays very popular due to their small size, flexibility and rapid prototyping possibilities. Therefore, it is much easier to work out the needed support for the techniques presented in this paper. In fact – as a member of the Feldspar development team –, we are planning to create a real life case study about the presented theoretical results, using Feldspar [12].

Adding the feature of calculating domain coverage to an existing language requires the followings. Firstly we have to create the machinery which will calculate the coverage: a new interpreter, which is just a smart wrapper of the original one. The wrapper interpreter will evaluate the program in a bottom-up manner, while in each step it will call the original interpreter to really evaluate a sub-expression, and then records the internal results for the coverage ratio calculation. This wrapping can be avoided if the interpreter supports such callback function which is invoked after each sub-expression's evaluation. For domain coverage we also need the `behavior` function defined for every construct of the programming language. The cost of creating the wrapper interpreter is constant, and does not depend on the actual language, while the cost of the `behavior` function scales as the complexity of the current language.

For test data generation we have to create a simple symbolic evaluator, which will handle the propagation and refinement of the domains connected to variables. Furthermore, we will need an inverse-like function for each language construct (as described in Section 3.1). The cost of the symbolic evaluator is almost constant, but it is hard to give a complexity estimation for the inverse-like definitions, because the creation of them completely depends on the current programming language and its special constructs.

5. Related work

Cheng and Hsiao [2] start with almost the same idea (examine the domain of some predefined internal variables) and they use it as a coverage metric and also for test data generation. But the approach differs, because ours has a formalized background, and a statical analyzer is built on top of that, while their approach uses simulation, dynamic analysis and heuristics to determine the equivalence classes. As a consequence of the chosen approach, our proposal is much more reliable, and it can be used even for reasoning about complete code coverage.

According to the survey [1] made by Zhu et al, the domain analysis and domain based approaches for testing are known for a long time. The main idea of domain based approaches is the following (according to Myers [5]). Try to partition the input domain of a program into a finite number of equivalence classes so that you can reasonably assume (but, of course, not be absolutely sure) that a test of a representative value of each class is equivalent to a test of any other value.

To partition the input space, we need information, which could come either from the specification or from the program. When partitioning the input space according to the specification [6], we consider a subset of data as a sub-domain if the specification requires the same function on the data. The software input space can also be partitioned according to the program (mostly evaluated symbolically [7]). In this case, two input data belong to the same sub-domain if they cause the same computation of the program. Usually, the same execution path of a program is considered as the same computation. Therefore, the sub-domains correspond to the paths in the program.

We would place our result between the two previously mentioned approaches, because our partitioning is done according to the program, but each level of the syntax tree is handled independently, which means that we do not collect path information. Instead of that – during symbolic evaluation – we use semantics information of the programming language constructs.

White and Cohen [8] use the result of domain partitioning to test the ”corner cases” of the program by generating minimum, maximum, just inside/outside values from each sub-domain. Our method randomly generates one value from each sub-domain, but a different generation strategy (such as boundary testing) is easily applicable.

6. Conclusion and future work

A new code coverage metric and a related automated test data generation method were presented. We realized that most of the existing methods concentrate only on the statements and control structures (branching, looping, etc.), but the lower layer (expressions) are mostly not considered thoroughly from the metrics point of view. Mostly, the analysis of expressions are about to determine their effects on the control flow, but the relation between two expressions – using some kind of semantics information – was never considered by coverage metrics (as far as we know).

The proposed metric takes both expressions and statements into account while calculating coverage ratio. During this calculation it uses specific domain information and behavior classes. These notions are describing equivalence classes on arguments and on programming language constructs, based on their internal behavior. Basically we pushed the analysis level lower, and the coverage reports only if the *implementation* of a programming language construct is covered. To be able to do this we need some semantics-like information, the so-called *initial domains*. They group such values for each language construct which trigger exactly the same behavior (observable and internal too!). This information is ideally given by the language designers.

The related test data generation aims to reach the highest possible domain coverage. The method uses symbolic evaluation to refine (split into smaller sub-domains) and propagate domain information upwards to the top. During the evaluation we could use special refinement strategies to keep the result of the refinement efficient. A possible future work is to investigate more advanced strategies, or try to find such ones that subsumes every other. As a result of this evaluation, we get such solidly refined domains which enable us to reach our goal. At the end we generate a single data from every behavior class.

The presented method is so far theoretical. Only a small scale case study (for a simple functional-ish language, embedded into Haskell, which was designed only for this purpose) was created to prove the feasibility of the idea. So it is definitely future work to put the theory into practice and create a real life, working case study.

References

[1] **Zhu, H., P.A.V. Hall and J.H.R. May,** Software unit test coverage and adequacy, *ACM Computing Surveys*, 1997, 366–427.

[2] **Cheng, X. and M.S. Hsiao,** Simulation-based internal variable range coverage metric and test generation model, in: *Proceedings of International Conference on Software Engineering and Applications* (*SEA*), 2006.

[3] **Nielson, H.R. and F. Nielson,** *Semantics with Applications: A Formal Introduction*, John Wiley & Sons, Inc., New York, NY, USA, 1992.

[4] **Glass, R.L.,** Persistent software errors, *IEEE Transactions on Software Engineering*, Los Alamitos, CA, USA, 1981, pp. 162–168.

[5] **Myers, G.J. and C. Sandler,** *The Art of Software Testing*, John Wiley & Sons, New York, NY, USA, 2004.

[6] **Hall, P.A.V.,** Relationship between specifications and testing, *Information and Software Technology*, 1991, 47–52.

[7] **Girgis, M.R.,** An experimental evaluation of a symbolic execution system, *Software Engineering Journal*, Herts, UK, 1992, pp. 285–290.

[8] **White, L.J. and E.I. Cohen,** A domain strategy for computer program testing, *IEEE Transactions on Software Engineering*, Piscataway, NJ, USA, 1980, pp. 247–257.

[9] **Beizer, B.,** *Software testing techniques* (2nd ed.), New York, NY, USA, 1990.

[10] **Information Processing Ltd.,** *Structural Coverage Metrics Executive Summary*, `http://www.ipl.com/pdf/p0823.pdf`, 2012.

[11] **Cornett, S.,** *Code Coverage Analysis*, `http://www.bullseye.com/coverage.html`, 1996.

[12] **Axelsson, E., G. Dévai, Z. Horváth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson and A. Vajda,** Feldspar: A domain specific language for digital signal processing algorithms, in: *Proc. Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign, MemoCode*, 2010.

[13] **McCabe, T.J.,** A complexity measure, in: *Proceedings of the 2nd international conference on Software engineering*, Los Alamitos, CA, USA, 1976.

[14] **Frankl, P.G. and E.J. Weyuker,** An applicable family of data flow testing criteria, *IEEE Transactions on Software Engineering*, 1988, pp. 1483–1498.

D. Leskó and M. Tejfel
Department of Programming Languages and Compilers
Faculty of Informatics
Eötvös Loránd University
H-1117 Budapest, Pázmány P. sétány 1/C
Hungary
`ldani@caesar.elte.hu`
`matej@caesar.elte.hu`