POTENTIALS OF CONSTRAINT-BASED METHODS IN SOFTWARE VERIFICATION AND VALIDATION

Ralf Gerlich[(1)], Rainer Gerlich[(2)]

Dr. Rainer Gerlich System and Software Engineering, Auf dem Ruhbuehl 181, D-88090 Immenstaad, Germany
[(1)] Email: ralf.gerlich@bsse.biz
[(2)] Email: rainer.gerlich@bsse.biz

ABSTRACT

We give an overview over the principles of constraint-based test data generation, discuss its limitations and potentials and touch some of the domains which may be interesting to combine with constraint-based testing techniques. Automated generation of test data is an example where this technique can be applied and significantly increase the degree of automation, but it is not limited to. This paper is intended to give interested readers a quick entry into the methods and applications to allow a deeper understanding and an informed verdict about the actual capabilities and potential future directions.

1. INTRODUCTION

In software test, automation of the test procedures itself can be considered solved. Automation toolkits such as JUnit for Java or its numerous counterparts for Ada, C/C++, Python, PHP and even Perl, for example, cover the relevant languages of almost all application domains.

The limits of automation can be stretched further by parameterising test cases and using hand-written test data generators to allow statistically relevant test case counts.

For two major problems of software testing, however, current practice does not or only very restrictively apply automatic approaches:

- test data generation, and
- generation of verdicts - also known as the "oracle problem" [1].

Constraint-based test-data generation (CBTDG)[2] contributes to the solution of the first of these issues. CBTDG can deliver test inputs that fulfil formal test objectives. These include generic coverage testing as well as specification-based approaches. Theory-testing, which explores the input space for possible violations of axiomatic properties of a functional unit, can also be supported by CBTDG.

With this paper we want to give an overview over the domain of CBTDG for practitioners and introduce limitations and potentials of this technique. The paper shall also be a starting point for interested readers to delve deeper into the topic.

2. BASIC APPROACH

Constraint-based techniques – in contrast to random test data generation or parameterized testing with data tables – work by solving systems of logical constraints – e.g. equations or inequations. The free variables in these constraint systems represent the initial and intermediate states of the program under test.

The idea is that these constraints represent the conditions under which the desired test objective is fulfilled. Test inputs may be found by solving the constraint system for the variables representing the initial state.

In cases of specification-based black-box testing these constraints can often be lifted directly from the specification. Examples here are the enforcement of invariants over specific input variables or the definition of nominal and faulty ranges for input data.

In the case of specification-based testing – e.g. based on a test model established independently from the implementation to be tested – the expected outputs may be taken from the free variables representing intermediate states as well as the final state of the program, as far as they are observable on the implementation.

For example, intermediate and final states of internal variables of a program may not be observable, while the return value of a function as a part of the final state of the function is typically observable.

In case of white-box coverage testing, the constraints need to be constructed from the control structure of the program itself.

Typically, an annotated form of the control-flow graph (CFG) is used for that, where the annotation consists of the formal semantics of the instructions and conditions contained in the nodes and edges of the CFG[3][4][5][6].

Consider the following code example:

```
unsigned int gcd(unsigned int a,
                 unsigned int b)
{
   while (a!=b) {
          if (a>b) {
                 a=a-b;
          } else {
                 b=b-a;
          }
   }
   return a;
}
```

This is the classical implementation of Euclid's algorithm for finding the greatest common divisor of two positive integers. Figure 1 shows the CFG for this algorithm. The circles represent the nodes and the arrows represent the edges. The nodes are labelled by the instruction that is executed when they are reached, and the edges are labelled by the conditions that must be fulfilled so that the edges can be traversed.

We can easily identify the conditions from the loop, $a \neq b$, and from the if-statement, $a > b$. The other conditions are the respective logical negations of these conditions. The center node at the top is the entry node and the node to the top-right is the exit node.

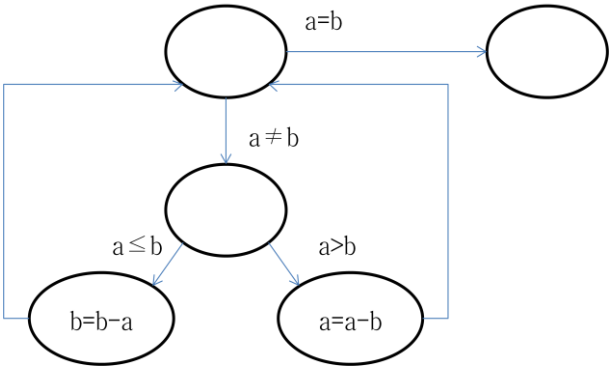

Figure 1: CFG for Euclid's GCD algorithm

Other methods[7] use the structure of the source code itself and apply transformation techniques such as loop unrolling to generate test data.

2.1 Infeasible Paths

While each of these methods has its advantages and disadvantages for different kinds of test objectives, they all share a common issue: infeasible paths of execution[5][7][6].

Infeasible paths are such paths which are possible to construct, e.g., in the CFG, but for which there is no input to the program that will lead to their execution.

They resemble dead code in the sense that if none of the paths leading to a specific part of the code is feasible, then this code part constitutes dead code. Put another way, as long as there is at least one feasible path leading to a specific code part, that code part is alive.

In practice, infeasible paths are much more frequent than feasible paths, so that avoiding the construction of infeasible paths is necessary for adequate performance.

One example is given in the following code example:

```
void sort(int len, int a[]) {
   int i, j, min, tmp;
   for (i=0;i<len;i++) {
          min=i;
          for (j=i+1;j<len;j++) {
                 if (a[j]<a[min])
                        min=j;
          }
          tmp=a[min];
          a[min]=a[i];
          a[i]=tmp;
   }
}
```

This is a classical implementation of the selection sort, which repeatedly searches for the minimum element in the remaining array and places that at the end of the already sorted array. Here `i` gives the length of the already sorted part of the array, and the inner loop iterates over the remaining elements, identifying the smallest.

Examining the source code closely we find that the total number of iterations of the outer loop is len, i.e. the number of entries in the array to be sorted. Further we see that with every iteration of the outer loop, the number of iterations of the inner loop decreases by 1, starting at $len - 1$ and finally reaching 0. Thus, the total number of iterations of the inner loop can be derived to be $\frac{len(len-1)}{2}$.

Thus the iterations of the inner and the outer loop are strictly interdependent and the execution sequence – except for the conditional statement inside the inner loop – is strictly defined by just the parameter `len`.

However, this dependency is not visible in the CFG as shown in Figure 2. The CFG allows a path through the program where the outer loop is iterated once while the total count of iterations of the inner loop is 2.

Let us construct the constraints in the order in which they come up during traversal. Note that all variables get an additional index, as the program variables support destructive assignments while the variables in the constraints are logical variables which may hold only one value. We increment the index whenever we observe an assignment. We will ignore the comparisons involving the array and `min` as they do not contribute to the explanation.

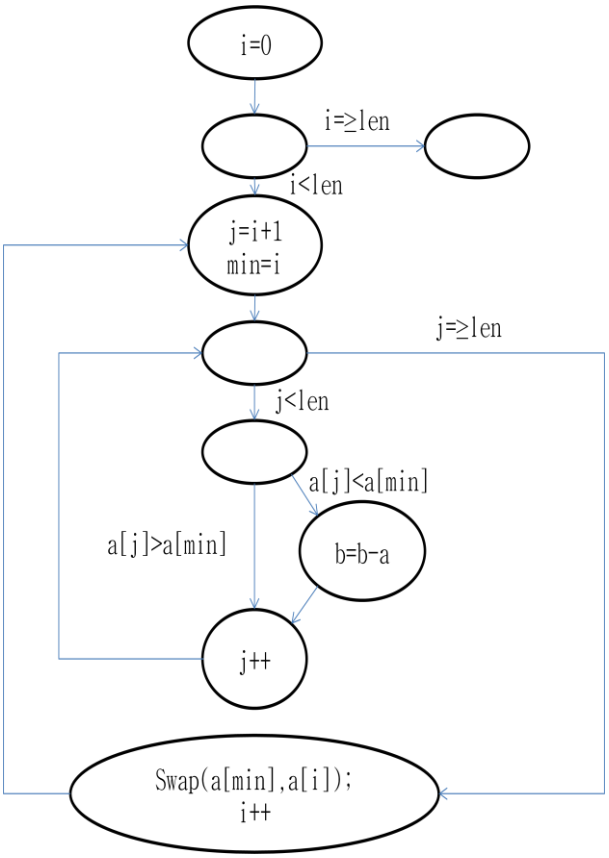

Figure 2: CFG of selection sort

So these are the constraints as they come up:

$$i_1 = 0$$
$$i_1 < len$$
$$j_1 = i_1 + 1$$
$$j_1 < len$$

$$j_2 = j_1 + 1$$
$$j_2 < len$$
$$j_3 = j_2 + 1$$
$$j_3 \geq len$$

$$i_2 = i_1 + 1$$
$$i_2 \geq len$$

We find that $i_2 = 1$ and thus $1 \geq len$ follows from $i_2 \geq len$, but also $j_1 = 1$ and $j_1 < len$, thus $1 < len$. This is a contradiction, indicating an infeasible path, as there is no solution for the constraint system.

Unfortunately, these contradictions often do not become visible before the full path was constructed. As we can see in the example, the constraint system contained no contradiction until $i_2 \geq len$ was introduced at the end of the sequence.

This problem cannot be solved by a simple trial-and-error approach, where the identification of a contradiction and thus an infeasible path simply results in the generation of a new path. The ratio of feasible to infeasible paths in a single program can be so small that the mean number of retries required to find a single feasible path can be too large for the approach to be practical[5][6].

Instead, special mechanisms are necessary for avoiding infeasible paths in the first place. These rely on detecting inconsistencies in the constraint system as early as possible during construction.

Further, in some approaches the path construction algorithm tries to predict the behaviour of the software[6][7] in order to find additional constraints early which may lead to contradictions and thus indicate an infeasible path.

Here the structure of a program may be indicative of code sequences which are executed in any case, independent of what decision is taken at the current branching point. For example, the code coming after the body of a loop is typically executed independently of whether the body of the loop is traversed or not.

The only time when this is not the case is when the body of the loop contains a jump statement by which the code after the loop is skipped, e.g. a `return` statement.

2.2 The Use of Constraint Solvers

Once a feasible path has been found and the set of constraints to be fulfilled is established, a solution to these constraints needs to be found.

A solution to a constraint system is a list of assignments of the form "V=x", where V is a variable and x is an atomic value, so that the constraint system is fulfilled when the variables are replaced by their assigned variables[8].

For example, considering the inequations $7 \leq x + y \leq 9$ and $-3 \leq x - y \leq -1$, a solution could consist of the assignments $x = 3$ and $y = 5$.

The algorithm that transforms a system of constraints into a solution is called a constraint solver. Depending on the type of constraints to be solved, different algorithms need to be used.

As in most cases there is more than one solution, the main task of the constraint solver is to find an approximate parameterisation of the solution space. The approximation should be as exact as possible, while not being a too heavy burden on performance – for both memory and CPU consumption.

In the context of CBTDG, constraint solvers are not only used to find a solution, but also to find whether a system of constraints is actually solvable. This is of specific importance when trying to avoid infeasible paths.

3. LIMITATIONS

The generation of input data is limited by the Halting Problem, which states that there is no procedure which can determine within finite time whether an arbitrary given program – in this case the program under test – will terminate within finite time on an arbitrary given input.

This is a fundamental problem in computer science. Because it is a theoretical and therefore absolute limitation, it cannot be removed by optimisation, only by specialisation.

In one specific instantiation, the halting problem implies that the CBTDG algorithm might not terminate while trying to find an input that covers the last statement of a program. More generically, this also applies to any statement in a program, as any program can be transformed in a syntactically and semantically valid way so that the target statement becomes the last statement in the program. This again leads to a program that may fall under the limitation imposed by the halting problem.

In addition to this theoretical limitation, there are practical limitations of mostly two kinds. The first kind is related to performance limitations for finding a feasible path and solving the constraints or detecting their insatisfiability – meaning that there is no solution – , the second kind are inaccuracies of the formal model used to describe the semantics of the program or the execution environment.

3.1 Performance Limitations

Performance problems come both from the fact that solving systems of equations and inequations is often difficult or impossible to do in a scalable way. This means that the execution time for the solver often grows non-linearly with the number of free variables involved. In turn, the number of free variables often grows at least linearly with the length of the constructed execution path.

Already the solution of systems of linear equations over real or rational numbers using the Gaussian elimination method may require a worst-case execution time growing with the cube of the number of variables.

Unfortunately, these are not simple specialisations of the more generic constraints over the real or rational numbers. While every integer and every floating point number is also a real and a rational number, the solution methods for the latter are not generally suitable for the former. Therefore other algorithms need to be applied to integer and floating point numbers.

In addition, there is a difference between the theoretical mathematical – infinite – set of integer and real numbers and their computational representation as a finite set of combinations of bits. In this context we take the notation integer, rational and real for the mathematical sets and two's-complement integer, fixed-point – as a generalisation of two's-complement integer – and floating point for the computational sets, to avoid overloading of terms.

The following examples outline a difference between the mathematical and the computational world.

3.1.1. Integer Constraints

For example, constraint systems which have a solution over the real or rational numbers do not necessarily have a solution over the integers.

A simple example for this is shown in Eq. (1). It is obvious that there is no integer between 2 and 3, however, a solution over the rational numbers would be x=2.5. This is also true if x where a two's-complement integer.

$$2 < x < 3 \tag{1}$$

Unfortunately, in practice typically neither the solutions nor their existence are as obvious as in Eq. (1). For example, Eqs. (2) and (3) show a system of inequations which has an infinite set of solutions over the real or rational numbers, but none over the integers.

$$1 < 4x + 3y < 5 \tag{2}$$

$$-1 < 2x - y < 2 \tag{3}$$

The existence of real and rational solutions can be shown using Fourier-Motzkin-elimination. From each of the two inequations two lower and two upper bounds for x can be extracted, all of which depend on y.

In order for a value of x to exist that satisfies these bounds, both lower bounds must be less than each of the upper bounds, leading to two trivially satisfied inequations – $1 - 3y < 5 - 3y$ and $-1 + y < 2 + y$ – and two non-trivially satisfied inequations $1 - 3y < 4 + 2y$ and $-2 + 2y < 5 - 3y$.

The latter two can be transformed into a lower and an upper bound for y: $-\frac{3}{5} < y < \frac{7}{5}$.

The significance of this – in the realm of real and rational numbers – is that as long as y lies within these bounds then there is also a value for x so that x and y satisfy the original inequations Eq. (2) and Eq. (3).

One could now consider only integer values of y that satisfy the bounds and try to find a matching integer value for x.

In this case the possible integer values for y would be $y = 0$ and $y = 1$.

Substituting $y = 0$ into Eqs. (2) and (3) results in $1 < 4x < 5$ and $-1 < 2x < 2$. These can be combined

to $\frac{1}{4} < x < 1$, using the maximum lower bounds and the minimum upper bounds from the inequations. We can see that there is no integer value for x in these bounds.

Substituting $y = 1$ into Eqs. (2) and (3) we arrive at $-2 < 4x < 2$ and $0 < 2x < 3$, which can be simplified to $0 < x < \frac{1}{2}$, which again obviously contains no integer value for x.

So the pure Fourier-Motzkin-method is not sufficient for solving integer inequations or checking for the existence of a solution. This is due to the fact that Fourier-Motzkin depends on the solution space to be compact, i.e. for any pair of distinct values x and z with $x < z$ there must be a value y so that $x < y < z$. This compactness criterion is clearly not fulfilled by the set of integer values, as we can see in the fact that Eq. (1) does not have an integer solution.

Although Fourier-Motzkin is able to reduce the search space for integer inequations, it is still necessary to check all the solution candidates individually and for all variables whether they are of integer type, resulting in computational effort which grows exponentially with the number of variables.

As an alternative the Omega-Test[9] can be applied, which in a first step uses a modified Fourier-Motzkin approach wherein the upper and lower bounds are narrowed down to ensure that the new bounds actually include integer solutions, given that the bounds do not represent an empty set.

As narrowing down may remove some solutions, these solutions have to be checked in a second step. Although this second step, again, may involve exponential computation time in the number of variables, the actual computation time spent in this step is considerably smaller than for the original Fourier-Motzkin approach, as not all but only a small subset of solutions are enumerated this way.

3.1.2. Floating-Point Constraints

Consider the following code snippet:

```
float v, inc, v0, limit;
inc = /* ... */;
v0 = /* ... */;
limit = /* ... */;
for (v=v0;v<limit;v+=inc) {
   /* ... */
}
```

It seems that this loop should terminate in any case – provided that inc is positive, as v seems to increase monotonically and thus at some point in time $v \geq limit$ must hold.

However, due to the way floating point values are represented, this loop may not terminate in the case that limit is large enough so that there is an $v < limit$ where $v + inc$ cannot be represented as a floating point value and is rounded towards v. In that case, the loop would continue endlessly without v ever reaching a value greater than or equal to $limit$.

Put more formally, the Eqs. (4) and (5) have no solution over the domain of the real or rational numbers, but they may have one over the floating point numbers, depending on the rounding mode in which the addition is executed.

$$x + y = x \tag{4}$$

$$y > 0 \tag{5}$$

In order to consider these specifics, algorithms are required which are more complex both in terms of their implementation and their use of computation resources.

Unfortunately, all floating-point operations are inherently non-linear due to the non-linear nature of the floating-point representation.

So-called normalised floating-point numbers are represented as $s\,(2^k + m)2^{e-k}$, where s is the sign (± 1), m is the mantissa consisting of $k - 1$ bits, and e is the (binary) exponent.

Only the non-normalised floating-point numbers have a linear representation, namely $s\,m\,2^{e_{min}}$, where e_{min} is the minimum exponent in the respective format.

In essence, non-normalised floating-point numbers are represented as fixed-point values. Unfortunately, normalised numbers occur much more often than non-normalised numbers as the results of computations.

Current approaches for constraint solving over floating-point arithmetic are limited to domain-filtering[10].

3.1.3. Domain Filtering

Domain filtering can be applied to almost any kind of constraint, including arithmetic constraints over integer, real, rational or floating-point values. Although we are discussing domain filtering in the context of floating point values, we will use integer values as examples as they are more suitable to explain the basic idea behind domain filtering.

In domain-filtering approaches[3][11], each free variable is associated with a set of allowed values for that variable. This set is called the domain of the variable. Typically contiguous intervals such as $x \in [5; 10]$ or finite set enumerations such as $x \in \{2,4,7,11\}$ are used.

The process of domain filtering iteratively tries to remove as many values from the domain of a variable that do not take part in a solution.

In the example from Eq. (1) we could start with the assumption that $x \in \{0, \ldots, 2^{32} - 1\}$. Given $2 < x$, we can remove all values from the domain that are less than

or equal to 2, resulting in the new domain $x \in \{3, \dots, 2^{32} - 1\}$.

Further, using x<3, we can remove all values from the domain that are greater than or equal to 3. This means that we have to remove all elements from the domain, resulting in an empty domain.

As a consequence we can conclude that there is no value for xx that would fulfil the constraints, and thus there is no solution within the initial domain of integer values.

In a constraint solver that is based on domain-filtering this kind of removal of values is continued until the domains of all variables have reached a fix point, i.e. until the domains cannot be refined anymore.

Unfortunately, the representation of the domains must be finite, i.e. it must not require an infinite amount of memory, and typically, the amount of memory required for representing a single domain not only has to be finite, but also bounded, meaning that no more than a given maximum of memory may be consumed by the representation of a single domain.

This often leads to poor approximations. For example, consider $x \neq 0$. The domain of x shall be represented as a contiguous set of integers, given by the lowest and the highest number in the set.

If the domain of x is $x \in \{0, \dots, 2^{32} - 1\}$, we can remove 0 from the domain and the filtered domain is $x \in \{1, \dots, 2^{32} - 1\}$.

However, if we start off with $x \in \{-2^{31}, \dots, 2^{31} - 1\}$, the result of removing 0 is $x \in \{-2^{31}, \dots, -1\} \cup \{1, \dots, 2^{31} - 1\}$, which is not a contiguous set. We can only approximate it as $x \in \{-2^{31}, \dots, 2^{31} - 1\}$. Thus the domain contains a value of x which cannot take part in a solution.

Some constraint solvers employ a combination of contiguous sets and enumeration of individual values by representing the domains as the union of a finite number of contiguous subsets. However, that also implies higher memory consumption. Perhaps non-intuitively it also implies higher CPU impact, as filtering for a constraint involving k variables now has to performed $N = N_1 * \dots * N_k$ times, where N_i is the number of subsets the domain of variable i has.

So even those systems at some point have to perform approximation, and while it is possible to minimise the impact of such an approximation, it is not generally possible to keep the impact acceptable.

The filtering algorithms for floating point constraints are a bit more complex as they have to consider proper rounding. In some cases – such as the example from Eqs. (2) and (3) – they exhibit inadequate convergence of the domains, taking inacceptably long to reach a fix point.

3.1.4. Non-Linear Constraints

Although still complex, solutions for linear equations and inequations over the integers are easier to find than for the same constraints over floating point values [9].

While for a purely linear system of constraints a proper parameterisation of the solution space is often possible, this is typically not the case for non-linear constraints such as multiplication of variables or bitwise binary constraints such as bitwise `and`, or, `xor` as well as bitwise shifting operators in their signed an unsigned forms.

This can be easily seen by considering the problem of finding the roots of a polynom of arbitrary degree greater than 2. For example, it is possible to solve $x^2 + px + q = 0$ using the general solution formula for quadratic equations $x_{1,2} = \frac{-p \pm \sqrt{p^2 - 4q}}{2}$. However, there is no known algorithm for determining the roots of $x^3 + px^2 + qx + r$ in general.

Furthermore, even the solution formula for quadratic equations only holds if the solutions may be real numbers, not, e.g. for floating point values.

As a consequence, the presence of non-linear constraints in a constraint system makes construction of solution by search necessary. This process often implies several steps of trial-and-error before a solution is actually found.

For some non-linear constraints, so-called dynamic linear relaxations exist, where the actual constraint is approximated using linear constraints[12]. Nevertheless, these relaxations also require search.

For example, consider a constraint $c = a * b$, involving the three variables `a`, `b` and `c`. As none of them is a constant, this clearly is a non-linear constraint.

Once we add constraints limiting the range of `a` and `b`, such as $-10 < a < 10$ and $-20 < b < 20$, we can further constrain the value of `c`.

By rewriting $-10 < a$ as $a + 10 > 0$ and $b < 20$ as $20 - b > 0$, we find that the product of the two positive values $a + 10$ and $20 - b$ is also positive. The new constraint is therefore: $20a - ab + 200 - 10b > 0$.

With $ab = c$ we finally get $20a - 10b + 200 > c$. So now we have found an upper bound for `c`, which is given in terms of `a` and `b`. Using the remaining three combinations of the lower and upper bounds for a and b we further get $20a + 10b + 200 + c > 0$, $200 - 20a - 10b + c > 0$ and $200 - 20a + 10b - c > 0$ as further bounds for c.

Of course, these are only linear approximations of the range of `c`. It is not possible to just remove the constraint $ab = c$ from the system and solve the remaining system of linear inequations.

It is said that the non-linear constraint is *relaxed* by introducing linear approximations. These relaxations are *dynamic*, because they are adapted whenever new bounds for `a` or `b` are introduced. However, it is not possible to completely replace non-linear constraints by such relaxations.

3.1.5. Linearization

A typical parameterisation used in context of bitwise binary operations is the bit vector. Here, the value of a variable with n bits is represented by an n-element vector of bits, each bit having either value 0 or 1. Once a value is available in that form, bitwise operations are easy to solve.

However, the naive conversion to a bit vector implies the creation of n additional logical variables for every value handled in this way, thereby also increasing the complexity of the system.

Some forms of bitwise operators can be expressed in linear form, most specifically those where one operand is constant. For example, the C-expression `b=a<<4` can be transformed into $b = 16a$. Similarly, b=a&12 can be represented by the Eqs. (6) to (9).

$$a = a_1 + 4a_2 + 16a_3 \quad (6)$$

$$0 \leq a_1 < 4 \quad (7)$$

$$0 \leq a_2 < 4 \quad (8)$$

$$b = 4a_2 \quad (9)$$

The idea is that $a$can be uniquely expressed as $\sum_{i=0}^{n-1} c_i 2^i$, where n is the width of n in bits and the c_i are the values of the individual bits with $0 \leq c_i \leq 1$.

However, we can combine some of the bits together and form Eq. (6), where $a_1 = c_0 + 2c_1$, $a_2 = c_2 + 2c_3$ and $a_3 = \sum_{i=0}^{n-5} c_{i+4} 2^i$. The additional constraints in Eqs. (7) and (8) ensure that there is a unique mapping between the a_i and the c_i.

We find that $b = 4c_2 + 8c_3 = 4a_2$. Using this representation we can avoid splitting a into all its individual bits: Instead of introducing 32 new variables for a 32-bit integer, we only introduce 3 new variables, all representing one or more bits of a.

It should be noted that the sum-of-bits-representation for `a` given above assumes that $a$is unsigned. However, Eqs. (6) to (8) are also valid for signed integers.

For example, if $a = -17$, we get $a_1 = 1$, $a_2 = 6$ and $a_3 = -2$ as solution.

Unfortunately, no such linearization exists for expressions such as `b=1<<a`.

3.1.6. Memory Model

The examples discussed above were very theoretical in so far as they did not involve explicit memory access. However, for most practical applications explicit memory access is necessary and often used, especially in the domain of embedded systems.

Consider, just as an example, processing of a telecommand onboard a spacecraft. The telecommand is provided in the form of a byte-stream, and groups of bytes in this stream are combined to form the individual command and data fields.

Languages such as C or C++ allow use of pointer arithmetic, allowing the code to access any place in memory directly. It is even possible to write an unsigned integer value to some address `addr` (assuming a representation of 4 bytes) in memory and sometime later read a floating point value from the same address. As a more extreme example, it is even possible to read an `unsigned short` (2 bytes) from the address 1byte after `addr`.

Whenever a read to memory occurs in the program for which test data shall be generated, the constraint solver must find the result of the read access. This means that the solver needs to find the latest preceding write access to a matching address.

When discussing the memory model, one should keep in mind that in most cases the values of the variables are represented by logical variables which have no distinct value until a solution is selected.

For example, the value of the pointer returned by a call to `malloc()` is not known in advance. It may take any value – except, typically, values indicating addresses on the stack or in the fixed code and data segments. Fixing the value of said pointer already during the process of constructing the path and the constraint system could therefore preclude specific solutions which would otherwise be possible.

This means that for a completely symbolic address, potentially all previous write accesses may match. As a consequence, the solver may have to iterate over a large number of alternatives, each of which may influence further path construction in a different way.

However, it is sometimes possible to effectively conclude whether a write access and a read access are related to the same address range. The underlying problem is called the aliasing problem, and there are several approximations of a solution.

For example, if a one-byte write access on the address $4k + 1$ is performed, followed by a one-byte write access to the address $4j + 3$, with k and j both being integers, it is possible to conclude that the two accesses do not overlap, even without knowing the concrete values of k and j.

If the two addresses would be the same, they would also have to be congruent modulo 4. However, $4k + 1 - 4j - 3 \equiv -2 \ (mod\ 4)$ and thus the two addresses are not congruent modulo 4. As a consequence, they cannot be equal.

The reverse, however, is not true. Although $4k \equiv 4j \ (mod\ 4)$, the addresses $4k$ and $4j$ do not necessarily refer to the same location.

Another possibility to exclude overlaps is when the two addresses refer to the same base address, which is typically the case when accessing objects on the stack, in the fixed data segment or within a single array.

Consider, for example, the following code:

```
void swap(int a[], int i, int j) {
   int tmp;
   if (i==j)
          return;
   tmp = a[i];
   a[i] = a[j];
   a[j] = tmp;
}
```

Let us assume that the base address of the array `a` is `p`. As an example, we want to know whether the memory accesses in the line `a[i] = a[j]` refer to the same address.

The address accessed in `a[i]` is $p + 4i$ (assuming an `int` size of 4). The address accessed in `a[j]` is $p + 4j$. When we have reached that line, we have already established that $i \neq j$ due to the `if`-statement containing a `return`-statement.

Comparing the two addresses, we find that $p + 4i = p + 4j$ if and only if $i = j$. This obviously contradicts the constraint $i \neq j$, so that we can conclude that the two accesses do not refer to the same address.

In contrast, consider the following example:

```
int scalar(int a[], int b[]) {
   int i, res = 0;
   for (i=0;i<3;i++) {
          res += a[i]*b[i];
   }
   return res;
}
```

Again assuming that the start address of the array `a` is `p`, and assuming that the start address of the array `b` is `q`, the accesses in the body of the loop would be to the addresses $p + 4i$ and $q + 4i$. In that case it is not possible to conclude whether these addresses are the same, as we have no information about `p` and `q`.

3.2 Discrepancies between Model and Reality

As has been discussed in the previous sections, constraint-based test data generation requires exact mathematical models of the semantics of a program. These are available if a program is considered to be a purely mathematical construct in isolation from any physical hardware or timing aspects.

However, when executing a program on a certain platform it may be not known how a mathematical operation on abstract numbers is mapped onto computational operations on reals numbers on a given platform.

As soon as the software to be tested communicates directly with hardware or when timing aspects are relevant, the models become inherently more complex. While it is theoretically possible to model timing in various ways, it is currently impractical in reality.

Also, mathematical models of program semantics may be incomplete regarding the actual behaviour of hardware, both periphery and the processor itself.

For example, the C standard ISO/IEC 9899 describes the result of many operations as being undefined, such as bitwise left-shifting with negative shift offsets. The rationale for the standard is of course that these cases are not sensible to define as they may be interpreted differently on different hardware.

Still, such operations are not necessary undefined given a specific compiler and platform.

Another example are transcendental functions – `sin`, `cos`, `tan`, etc. – in floating-point calculations.

The standard IEEE754 requires that the results of addition, subtraction, multiplication, division and square-root shall be correctly rounded, i.e. the results should be same as if the operation had been executed in full accuracy and then rounded. This means that the results of these operations are well-defined.

The standard does not enforce an equivalent requirement on other operations, including, e.g., the transcendental functions. This is not a case of neglect on the part of the authors of the standard, but instead a result of the so-called table maker's dilemma: An algorithm producing correctly rounded results would first have to calculate the result with a larger precision. The number of bits to which the result would have to be precise cannot be known in advance, and calculation according to a – yet unknown – worst case would be computationally expensive.

As a result, these functions are not well-defined according to the standard and their values are dependent on the respective implementation on the target. This makes it difficult for a solver to find a solution while not knowing exactly the conditions of a representation.

One way to solve the problems around these operations is to completely avoid them. But in some cases this may not be desirable. For example it is known that on some platforms left-shifting with negative offsets is handled

in a sensible manner. In contrast, explicitly checking the sign of the offset and using right shift for negative values and left shift for positive values may impair performance and is difficult to automatically optimise by a compiler.

The other way is to make sure that the implementation on the given platform matches the definition in the theoretical model on which the test data generator is based.

The simulation of hardware/software-interaction may also be impaired by another limitation of the approaches available today. One of the issues introduced by interaction between hardware and software in terms of the models is parallelism: Processes in (periphery) hardware and in software execute in parallel to each other.

As previously mentioned, the avoidance of infeasible paths is very important for feasibility of constraint-based test data generation. Unfortunately, none of the currently known methods for avoidance of infeasible paths is able to properly handle parallelism.

In addition, timing is often very important in hardware/software interaction. The timing of complex hardware circuits is not only difficult to simulate accurately, but – as mentioned earlier – is also difficult to model in a feasible way in the context of CBTDG.

If timing is not considered properly in a parallelised system, the state of the system – including memory and message queues – cannot be determined for a given point in time. This means that the interaction between the various processes cannot be considered for test-data generation.

4. BIAS CONSIDERATIONS

Testing is imperfect in the sense that only the presence of defects can be shown, but not their absence. As a consequence, a software test very much resembles a scientific experiment aiming to disprove the hypothesis of the software being correct.

Therefore the proper selection of samples – in this case: test inputs or test cases – is important for the significance of the result. If – by chance – only test cases are selected for which the software does not exhibit faulty behaviour, although the software is not correct, the hypothesis of correctness may be accepted by mistake.

The same is true if the software exhibits faulty behaviour but the faultiness of the behaviour stays unnoticed, e.g. due to being hidden in masses of data.

Classical coverage criteria such as statement coverage or even Modified Condition/Decision Coverage (MC/DC) – the latter of which is normally considered to be very challenging – can help guide the selection of appropriate test inputs, but are not sufficient on their own[13].

Therefore, even though any test input from a given partition as defined by the coverage criterion may seem equivalent to any other input from the partition, this is not the case.

This not only implies that not just any input that matches the criterion may be selected. It also implies that giving precedence to some of the inputs – for whatever reason, including technical reasons rooted in algorithm optimisation – may negatively impact the validity of the test conclusion.

Therefore, algorithms used for automatic test data generation need to be carefully tuned not to be biased towards any specific subset of the possible solution set.

Assuming that no more specific information is available on which values from the candidate set are more likely to activate fault conditions than others, the algorithms must therefore apply random selection of input data from the solution set.

If, on the other hand, more specific information is available, this information should instead be formalised by way of a more specific coverage criterion.

However, this may be difficult to achieve. For example, the selection of a path through the system by way of a random walk will favour shorter paths, if at each step the edge to be traversed is chosen by a uniform distribution from the available set of edges. If it is acceptable to limit the length of the path to a given maximum, then a method is available that can be used to ensure uniform distribution among the paths with that given maximum length[14]. However, this method needs to be integrated with the method used for avoiding infeasible paths.

Even then, the solution of integer inequations according to the Omega Test involves two steps, one of which being the modified Fourier-Motzkin-Elimination and the other being the search involving the possible solutions that may have been missed in the first step. This second step may involve the consideration of multiple, possible non-disjoint sets of solution candidates.

Thus when selecting a solution, a decision needs to be taken over whether to select a solution that results from the first or the second step, and if a solution from the second step is targeted, one of the candidate sets needs to be selected.

In order to guarantee a uniform distribution among the two steps as well as between the individual candidate sets, the number of solutions provided by each of the choices must be known. However, there is no way to determine this number except by first enumerating all solutions.

However, not only CBTDG algorithms may show bias towards specific test inputs.

Their human counterpart – the test engineer – may exhibit engineering bias, selecting test input values based on previous knowledge about the system. This assumed knowledge may include invalid assumptions. Such invalid assumptions may lead the test engineer to assume that the results of some specific test cases would be obvious and irrelevant, thus not selecting these test cases, although they could actually show the invalidity of the assumption.

Another case of engineer bias is when a test engineer selects easily visible solutions to the test objective and discards solutions which are more difficult to calculate and thus may be erroneous when calculated manually.

5. CONSEQUENCES

Although some of the limitations seem grave, there are some simple rules for design and coding, which help counter these limitations and at the same time also can be considered good engineering practice.

Further, CBTDG should not be considered the method of first choice for test data generation when simpler and more effective methods are available.

Instead, CBTDG should be used to complement other approaches to test data generation such as random testing. Random testing is not only faster in generating test data sets with high variability, but also is based on very simple algorithms which are less prone to introduce bias in the selection of test data.

5.1 Design and Coding Rules

By applying the following rules, the produced code may become easier to test and verify automatically. Adherence to these rules is optional, but performance of constraint-based test and verification may suffer if the rules are not followed.

5.1.1. Separate Hardware Interaction from Logic

As a basic rule, interaction with peripherial hardware should be separated from operational logic as far as possible. This way CBTDG can be applied to the logic parts without raising any issues in hardware/software interaction. The parts directly interacting with periphery – e.g. transmitters or sensors – typically cannot be tested without hardware-in-the-loop anyway.

However, by separation of these concerns, the manual effort for testing the logic part can be reduced considerable, although the manual effort for the hardware-interaction part remains.

5.1.2. Avoid Overlays and `Union`

Overlays occur when a location in memory is accessed using different types, e.g. writing a 32-bit unsigned integer location only to later retrieve a float value from the same location. In symbolic execution, this requires expensive symbolic casting operations. A more extreme example is writing a float value to a location only to receive the lowest 16 bits in the form of an unsigned integer.

This is often an issue when communication data is to be processed. The communication packets are delivered by the hardware in the form of unstructured byte-streams, so that the data is often referred to in the form of a pointer to an array of bytes.

Access to the data may occur in the form:

```
uint16_t getSeqNum(char* packet) {
   uint16_t val;
   memcpy(&val,
          packet+OFFSET_SQNUM,
          sizeof(val));
   return val;
}
```

Note that although the use of `memcpy` seems inefficient and unnecessary here, this way of accessing fields in a telecommand was actually observed in actual flight software.

Instead, proper `struct`-types defining the structure of the individual packet types should be defined and the cast from the byte-pointer to a pointer to the given structure should be done as early as possible in the processing sequence.

Often alignment issues are claimed, but they do not justify the construct above, as they can be solved satisfyingly with non-exotic constructs. For example, Almost all compilers support specific language extensions by which a struct can be packed, i.e. the alignment requirements of each field are explicitly set to 1 byte. If the underlying architecture does not support non-aligned memory access – which is the case in some RISC architectures – appropriate memory access scheduling can be left to the compiler. Even though this might inspire concerns about performance, as a single 32-bit access may have to be split up into as much as 4 individual byte reads, these concerns are even more justified with the `memcpy`-approach shown above.

For example, packet structure could be defined involving a common header and different structures for the different command types:

```
typedef enum {
   Cmd_Set_State,
   Cmd_Get_State,
   ...
} CommandType;
typedef enum ProcessType;
typedef enum StateType;
```

```
typedef struct {
   uint16_t     crc16;
   uint16_t     length;
   uint16_t     seqNum;
   CommandType  command;
   ...
} PacketHeader;
typedef struct {
   PacketHeader       header;
   ProcessType         process;
} PacketGetState;
typedef struct {
   PacketHeader        header;
   ProcessType         process;
   StateType           targetState;
} PacketSetState;
char TC_PacketBuffer[TC_MAX_SIZE];
void IRQ_PacketReceived();
void DispatchTC(PacketHeader*);
void GetState(PacketGetState*);
void SetState(PacketSetState*);
```

The function `IRQ_PacketReceived` handling the reception interrupt from the transceiver could first verify the length of the packet and the CRC16 checksum, and pass by casting

`(PacketHeader*)TC_PacketBuffer`

to the dispatch function `DispatchTC`. The latter would then check the command in the header and dispatch the TC to the respective handler function, casting it to the respective structure type in the process, in this case `(PacketGetState*)` for `GetState` or `PacketSetState*` for SetState.

This way, the handler functions as well as the dispatch functions have well-defined interfaces and all memory accesses are typed, different from the `memcpy` approach.

Overlays are avoided – with one exception in the example: The CRC16 calculation will probably access every byte in the packet individually, while the same bytes may be accessed using different types later on. However, as this is limited to a single function, it does not impact the test data generation for `GetState` and `SetState`.

For the reader concerned by the performance overhead introduced by the dispatch function calls it should be noted that of course the compiler can be instructed to inline some of the functions, avoiding the overhead while at the same time keeping up the well-defined interface.

5.1.3. Use Small Functions for Specific Purposes

Tasks like sorting, CRC-calculation, etc. should be exported to small specialized functions. If there is concern about the call overhead, the functions can be inlined by the compiler.

In unit testing the test for a single unit – in our case a single function – is often created under the assumption that units used by the unit under test are working correctly. So if a function checking the integrity of a telecommand packet uses a CRC-calculation function, the test of the former function assumes that the latter is working correctly.

Due to the declarative nature of the algorithms underlying CBTDG it is possible to replace individual functions – except, of course, the function under test – by their respective specification.

Coming back to the sorting function, for its functionality it is only important to know that the result is sorted and maybe whether the sort is stable – i.e. whether the order of elements that are considered equal is changed. It is not important to know how the sorting takes place.

Additionally, symbolically executing the sorting algorithm is almost certainly more computationally expensive than asserting that the resulting list is sorted.

By splitting specific tasks into specialized functions replacement of such functions by their respective specification may be possible – given that the tool used supports such a replacement.

This – in general – would be good engineering practice, as it separates the code into small, self-contained elements. These are also easier to manage in manual testing,. In consequence, such coding issues are not only a matter of CBDTG, but of good engineering practices in general.

5.1.4. Prefer fixed-point calculations over floating-point

This recommendation does not generally fit with the other rules as it cannot be considered good programming practice in general.

Further, it is not always possible or desirable from a design point of view to replace floating-point calculations by fixed-point calculations.

However, specification and design should in general also consider testability of a system and the effort impact implied by limited applicability of automatic testing solutions as well as their current technical limitations.

As mentioned previously, available constraint solvers for CBTDG generally deal better with integer – and thus also fixed-point – arithmetic than with floating-point.

5.1.5. Limit the range of parameters as far as possible

The free variables present in the constraint system represent the initial or intermediate states of variables of the program and thus the values of these free variables are implicitly limited to the type range of the associated program variables.

Solving a constraint system typically also includes search over the remaining search space, which, although in many cases already being reduced due to the known constraints, may in the worst case extend to the full type range of a variable.

Further, limits imposed on one variable may also imply further search space reductions for other variables. For example, consider the program z=x+y, where z is an unsigned 32-bit integer variable. Let us assume that the range of x has already been constrained to $\{2, \dots, 10\}$, whereas y is an unconstrained unsigned 32-bit integer. In that case, the solver cannot restrict the range of z beyond its regular type range.

If, however, $\boldsymbol{y}$ by design shall only take values that all fit in the range of an unsigned 16-bit integer, by declaring $\boldsymbol{y}$ with a matching type – e.g. unsigned short in C – the range of $\boldsymbol{z}$ can be constrained to $\{\mathbf{2}, \dots, \mathbf{65545}\}$. This is smaller than the full 32-bit integer range by a factor of $\mathbf{2^{16}}$, meaning that the solution may be found up to 16 times faster than with the original setup.

5.2 Combining CBTDG with Random Testing

In the last years, random test data generation and similar black-box approaches like grid-based testing have been shown to reach high – although not necessarily complete – test coverage on their own. Heuristic methods are already in use in the practice of test data generation that increase the coverage achieved without using constraint solution algorithms[15].

So in terms of test-data generation there is no reason to use CBTDG as the first and only method within a test preparation process.

Random and grid-based test data generation, for example, is able to generate and evaluate thousands of test inputs based on type ranges visible in a prototype within a few seconds, where evaluation includes analysis of coverage. In contrast, generating a single test input for practically relevant code with CBTDG may take a tenth of a second up to many minutes, depending on the code and the point in the code to be reached.

Further, type-range-based test data generation (TRTDG) allows unbiased generation of test input as well as generation of input data according to operational or other profiles without much complication. However, it is not perfect regarding reaching full coverage as it does not consider constraints at all which in some cases is required.

On the other hand, constraint-based methods require considerable efforts and complexity in order to ensure absence of bias. There are methods which, for example, ensure that from all paths with a given maximum length any path is selected with the same probability[14]. However, these methods may introduce issues with infeasible paths[5] and are difficult to combine with approaches for avoidance of such paths. This is where TRTDG can easily complement CBTDG.

So CBTDG is best used to complement the test coverage already provided by heuristic and type-range-based test data generation.

6. POTENTIAL

Although CBTDG seems to be laden with performance and theoretical limitations, its use can still be more efficient than manual test data selection. This, of course, can only be valid wherever CBTDG is applicable in the first place, i.e. where formal test objectives are available that can be translated into constraints. This is the case for typical and also extended coverage criteria[6].

After all, CBTDG is able to generate test data for some test objectives within less than a second on current computers, with potential for further increase in efficiency, while a test engineer might even require more than that time for even understanding the test objective.

An important benefit is that CBTDG can be advised to provide test inputs for a code segment or branch which could not be covered by TRTDG applied in a first step. This way most of the branches can be covered by TRTDG rather fast – typically up to 60 – 80 % - and the remaining coverage is provided by CBTDG – if full coverage is possible at all – by identifying the non-covered elements after the first step and then asking CBTDG for the remaining test inputs.

When combined with meta-heuristic testing approaches[16][17], proper formal oracles or plausibility checks, use of CBTDG may lead to a higher number of test inputs and consequently provide higher reliability of test results.

For example, the already introduced `gcd` function is commutable, i.e. `gcd(a,b)=gcd(b,a)` for any values of a and b. A constraint-based test data generator could systematically explore the set of possible paths through the function and check for violation of this invariant. Alternatively it might be possible to explicitly search for counter examples.

If there is a plausibility condition for the return value of a function f, such as $10 < f < 20$, then CBTDG could be used to explicitly search for inputs for which this plausibility condition is violated. These cases would then be interesting material for further analysis.

The (relative) lack of bias in combination with the exact representation of language semantics may lead to unexpected test input being selected – which a test engineer never would derive or consider as an

interesting input - resulting in evidence for unexpected behaviour of the software under test.

For example, in context of machine representations the condition `(u-l)<100` with two signed integers `u` and `l` may very well be true, while at the same time a loop

```
for (i=l;i<u;i++){...}
```

may take much more than 100 steps, for example if $l = -2^{31} + 1$ and $u = 1$. In this case, $u - l$ evaluates to 2^{31}. This is outside the range of `unsigned int` and thus according to the C-standard is converted to -2^{31}, which obviously is less than 100. Also, $l < u$ holds. The loop will iterate $2^{31} - 1$ times

Similarly, the expression `i+1` may very well be 0 even if $i \geq 0$ holds (as shown in the example below).

These are consequences of explicit modelling of two's-complement arithmetic of integers. Basically, the expression `z=i+1` in C is not equivalent to the expression `z=i+1` in classical integer arithmetic. Rather, it must be translated into Eqs. (10) and (11). The additional variable c represents the carry that occurs during overflow. Note that these equations exactly model the behaviour specified in the C-standard ISO/IEC 9899:2011.With the translations shown in Eqs. (10) and (11), it is easy to see that $i = 2^{32} - 1$ leads to $z = 0$, although $i \geq 0$ holds.

$$z = i + 1 + c2^{32} \quad (10$$

$$0 \leq z < 2^{32} \quad (11$$

This case was observed in a loop of the form

```
int getIncrement(...);
/*... */
unsigned int i,j;
i= getIncrement(...);
for (j=0;j<100;j+=i+1) {...}
```

The value returned from *`getIncrement`* was not checked for whether `i+1` would be zero in the memory representation, so it was possible to send the function into an endless loop.

This example demonstrates how the rather exotic value of $2^{32} - 1$ = 0xffffffff can occur for `i`: In this case it is a matter of (implicit) type casting between `signed` and `unsigned`. If the function returns -1 e.g. to flag an error while in normal case only non-negative return values are expected, `i` gets the highest value of *`unsigned int`*, so that adding 1 evaluates to 0. In terms of *`signed int`* this is also correct: -1 + 1 =0.

Similarly, a quasi-endless loop would occur for

```
for (j=0;j<i;j++) {...}
```

while at the first glance it could be expected from the returned value -1 that the loop never would be executed as 0 ≤ j < -1 is a contradiction. But the compiler interprets the value 0xffffffff differently, depending on the type.

7. OTHER POTENTIAL APPICATION AREAS

While in context of this paper we only discussed constraint programming in the context of constraint-based test-data generation, there is a large potential for other applications [8].

Solutions based on constraint programming are already in use for scheduling, not only in the computer-science sense but also, e.g. for fleet planning in logistics or aviation.

In a similar context they may be suitable for continuous mission replanning and on-board autonomy, e.g. for planning orbit manouvers for earth-observation based on requests coming from ground.

In operations research constraint programming is used for optimisation problems.

Also, constraint programming methods may be used for implementing design choices in code generation or for optimisation in compilers. One example for the latter is the use in parallelisation of algorithms or rescheduling of instructions.

8. CONCLUSIONS

While the original movement was already started in the early 1990s, the area of CBTDG has seen a surge of research activities in the last decade, some of which have led to actual implementations close to industrial requirements.

Effective use requires knowledge about the possibilities and the limitations at the same time. Some of the limitations highlighted in this paper do not only apply to constraint-based test data generation, but also to other formal and automated methods in software verification and validation, most notably to many forms of model checking and abstract interpretation.

Although there is still a lot of research potential left, most specifically in the area of solvers and the application to parallelised environments, CBTDG has the potential to become an important tool of the trade of software testing within the near future.

BSSE is also contributing to industry-ready solutions to many of these issues in the course of industrial research accompanying the development and maintenance of an industry-ready CBTDG tool integrated with its already existing tools for random, heuristic and grid-based testing.

9. REFERENCES

[1] D. Hoffman, "Using Oracles in Test Automation," in *Proceedings of the Pacific Northwest Software Quality Conference (PNSQC 2011)*, 2001.

[2] R. A. DeMillo and A. J. Offutt, "Constraint-Based Automatic Test Data Generation," *IEEE Transactions on Software Engineering*, vol. 17, no. 9, pp. 900-910, 1991.

[3] R. Ferguson and B. Korel, "The chaining approach for software test data generation," *ACM Transactions on Software Engineering Methodology*, vol. 5, no. 1, pp. 63-86, 1996.

[4] T. S. Nguyen and Y. Deville, "Automatic Test Data Generation for Programs with Integer and Float Variables," in *Procedings of 16th IEEE International Conference on Automated Software Engineering (ASE)*, 2001.

[5] S.-D. Gouraud, "AuGuSTe: a Tool for Statistical Testing - Experimental results," Université Paris Sud, 2005.

[6] R. Gerlich, *Verallgemeinertes Rahmenwerk zur constraintbasierten Testdatenerzeugung aus Programmflussgraphen*, Universität Ulm, 2009.

[7] A. Gotlieb, B. Botella and M. Rueher, "A CLP Framework for Computing Structural Test Data," in *CL '00 Proceedings of the First International Conference on Computational Logic*, 2000.

[8] T. Frühwirth and S. Abdennadher, Essentials of Constraint Programming, Springer, 2003.

[9] W. Pugh, "A practical algorithm for exact array dependence analysis," *Communications of the ACM*, vol. 35, no. 8, pp. 102-114, 1992.

[10] B. Botella, A. Gotlied and C. Michel, "Symbolic execution of floating-point computations," *Software Testing, Verification and Reliability*, vol. 16, no. 2, pp. 97-121, June 2006.

[11] B. Korel, "Automated Software Test Data Generation," *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 870-879, 1990.

[12] T. Denmat, A. Gotlieb and M. Ducassé, "Improving constraint-based testing with dynamic linear relaxations," in *Proceedings of the 18th IEEE International Symposium on Software Reliability Engineering*, 2007.

[13] R. Hamlet and R. Taylor, "Partition Testing does not inspire confidence," *IEEE Transactions on Software Engineering*, vol. 16, no. 12, pp. 206-215, 1990.

[14] A. Denise, M.-C. Gaudel and S.-D. Gouraud, "A generic method for statistical testing," in *Proceedings of the 15th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2004.

[15] R. Gerlich, R. Gerlich and C. Dietrich, "Fault Identification Strategies," in *DASIA 2009 DAta Systems In Aerospace*, 2009.

[16] A. Gotlied, "Exploiting Symmetries to Test Programs," in *Proceedings of the 14th International Symposium on Software Reliability Engineeering (ISSRE)*, 2003.

[17] J. Mayer and R. Guderlei, "Test Oracles using Statistical Methods," in *Testing of Component-Based Systems and Software Quality*, Gesellschaft für Informatik, e.V., 2004, pp. 179-189.

[18] P. Godefroid, "Random testing for security: blackbox vs. whitebox fuzzing," in *RT'07: Proceedings of the 2nd international workshop on Random testing*, Atlanta, 2007.

[19] A. Gotlieb, "INKA: An Automatic Software Test Data Generator," in *Proceedings of DASIA 2001 - DAta Systems in Aerospace*, Nice, 2001.

[20] A. Offutt, Z. Jin and J. Pan, "The dynamic domain reduction procedure for test data generation," *Software: Practice and Experience*, vol. 29, no. 2, pp. 167-193, 1997.

[21] A. Brillout, D. Kroening and T. Wahl, "Mixed Abstractions for Floating-Point Arithmetic," in *Formal Methods in Computer-Aided Design, 2009. FMCAD 2009*, 2009, pp. 69-76.