Generic and Extensible Automatic Test Data Generation for Safety Critical Software with CHR

Ralf Gerlich

BSSE, Auf dem Ruhbuehl 181, D-88090 Immenstaad, Germany,
ralf.gerlich@bsse.biz

Abstract. We present a new method for automatic test data generation (ATDG) applying to semantically annotated control-flow graphs (CFGs), covering both ATDG based on source code and assembly or virtual machine code. The method supports a generic set of test coverage criteria, including all structural coverage criteria currently in use in industrial software test for safety critical software.
Several known and new strategies are supported for avoiding infeasible paths, that is paths in the CFG for which no input exists leading to their execution. We describe the implementation of the method in $\mathrm{CHR}^{\vee}$ [1] and discuss difficulties and advantages of CHR in this context.

1 Introduction

Testing is one of the most important methods of analytical quality assurance of software-based systems, but also one of the most expensive, causing 50% of the effort for a typical software project [2] and nearly 80% for safety-critical software.

Software is categorised as safety-critical if its failure can lead to death or serious injury of humans, damage or loss of equipment or environmental harm. This kind of software is typically most complex as it has to handle and recover from many different types of failure, leading to individual components of hundreds of thousands or even millions lines of C or Ada code and thousands of interdependent functions.

Automatic Test-Data Generation (ATDG) aims to automate selection of test inputs and — if possible — the expected outputs. However, ATDG requires formal answers to two questions:

- Which criteria shall govern the selection of test data?
- How can we find samples that fulfill these criteria?

In practice the first question is typically answered by a list of well-known structural test criteria [3]. These criteria are defined based on the activation of specific portions of the control-flow graph (CFG) of the function under test. For example, the *all-nodes* criterion requires that for each node there is at least one test case by which the node is executed. Using the edges of the CFG, *all-edges* can be similarly defined.

Data-flow-based criteria are based on so-called definition-free paths. A path is *definition-free* regarding some variable v if no node in the path contains an assignment to v, except for the start and the end node. For example, the criterion *all-defs* requires that for each definition d of a variable v there must be at least one test case executing a definition-free path regarding v from d to a use u of v.

One approach to the second question is random testing [4], where inputs are selected randomly and the fulfillment of the criterion is checked afterwards [5]. Here statistical metrics on software quality can be derived. However, some portions of the CFG can only be reached for a small set of inputs and are therefore difficult to activate randomly.

Another approach is construction of some path in the CFG that fulfills the respective criterion, symbolically executing it to derive a set of equations and inequations and solving for the inputs [6]. However, in most cases there is a considerable set of so-called *infeasible paths* which cannot be activated by any input [7].

Gotlieb et al [8] propose a method for handling structured programs only consisting of `while`- and `if-else`-constructs. Multiple parts of an execution sequence can be processed in parallel, propagating information to be used for detecting and avoiding infeasible paths. Although a CFG can be emulated by a `while-if`-program, it is difficult to translate a CFG making proper use of the features of the method. Further, coverage goals have to be described by reference to the structured constructs, which specifically makes enforcing data-flow-based criteria tedious.

Godefroid et al [9] propose to randomly select an input and monitor the execution path for this input. Either the path matches the goal or there is a point in execution where a conflicting decision is made. The first conflicting decision is found and the path leading up to it is executed symbolically to derive a set of constraints. The constraints are then amended to enforce a decision matching the goal and the process is repeated with the solution of the constraint system as new input, if any. However it is possible that all paths with the selected prefix are either infeasible or do not match the goal.

We present an approach that overcomes these limitations and allows to compare different strategies, and discuss its implementation in $\mathrm{CHR}^{\vee}$.

This paper is organised as follows: In Sect. 2 we briefly describe the relational semantics of CFGs, which is used in Sect. 3 to formalise some structural test criteria and to introduce solution rules and strategies. In Sect. 4 we discuss some of the advantages and disadvantages of CHR which became visible during implementation. Some results obtained from the prototype implementation are shown in Sect. 5, followed by our conclusions in Sect. 6.

2 Semantics of Control-Flow Graphs

For a detailed description of the theoretical foundation please refer to [7].

A CFG is a directed graph consisting of a finite set of nodes N and edges $E \subseteq N \times N$, with two special nodes, the entry node s and the exit node e. The entry node s has no incoming edges, while the exit node e has no outgoing edges. Further, every node n is reachable from s and e is reachable from every node n. We define E^+ to be the transitive closure of E. See Fig. 1(a) for an example CFG.

We extend a CFG to a program-flow graph (PFG) defining the semantics of the program. For this we first introduce the notion of memory state. A memory state represents the contents of memory relevant to the program at any given time during execution of a program. It can be represented, for example, as a tuple $(v_1, \ldots, v_n)$, where v_i represents the value of a variable V_i in that state. In the following, we will use S to designate the set of memory states.

During execution, the program will proceed through its nodes, modifying the memory state by the statements inside the nodes. After execution of a node, a decision is required at which node the execution shall continue. A successor is eligible if and only if the predicate attached to the edge leading to the successor is fulfilled by the current memory state. Further, the modification of the memory state can be described as a relation between the memory state as found on entry to the node and the memory state as found on exit to the node.

In the following, the predicate attached to edge (u, v) is called $\mathcal{C}(u, v)$ and the relation for node u is called $\mathcal{B}(u)$. We assume that the predicates and the relations are decidable.

A graphical representation of a PFG is given in Fig. 1(b). Here we define S as the set of binary tuples of natural numbers $\mathbb{N}^2$. In this example, the body of node 3 is to be understood as the relation $\mathcal{B}(3)$ with $(a_1, b_1)\ \mathcal{B}_3\ (a_2, b_2) \Leftrightarrow b_2 = b_1 \wedge a_2 = a_1 - b_1$.

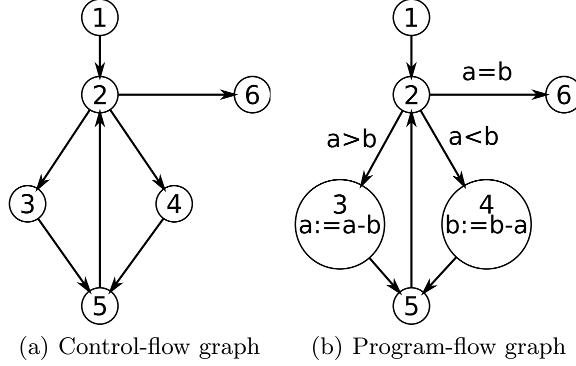

Fig. 1. Control- and Program-flow graphs for Euclid's algorithm

Given two nodes a and b we can define a relation $\mathcal{S}_{[a,b]} \subseteq S \times S$. For any two memory states $x, y \in S$, we have $x\ \mathcal{S}_{[a,b]}\ y$ if and only if input x to node a can be transformed to output y of node b along some execution path from a to b. We call $\mathcal{S}_{[a,b]}$ the *specification* of all paths from a to b.

Similarly, we can define a relation $\mathcal{I}_{[a,b]} \subseteq S \times S$ which relates the *outputs* of a to the *inputs* of b. We call $\mathcal{I}_{[a,b]}$ the *inner specification* of all paths from a to b, namely because it represents the transformations described by $\mathcal{S}_{[a,b]}$ minus the application of the bodies of a and b at the beginning respectively the end of the execution sequence.

The formalism can be used to model non-deterministic programs, but in practice non-deterministic programs are the exception.

3 Test-Data Generation

Using the relations defined in Sect. 2 we can now formalise several structural coverage criteria. For example, in order to cover some specific note $n \in N \setminus \{s, e\}$, we have to find some input $x \in S$ that will lead to execution of node n. This is only the case if there is some $y_1 \in S$ so that $x\ \mathcal{S}_{[s,n]}\ y_1$ holds. For deterministic PFGs, this necessary condition is also sufficient.

If we not only want to ensure the execution of n but also the completion of the program afterwards, we need $y_1, y_2, y_3 \in S$ so that $x\ \mathcal{S}_{[s,n]}\ y_1 \wedge y_1\ \mathcal{I}_{[n,e]}\ y_2 \wedge y_2\ \mathcal{B}(e)\ y_3$ hold.

Similarly, if we want to ensure execution of some edge (u, v), we need some $y_1 \in S$ so that $x\ \mathcal{S}_{[s,u]}\ y_1 \wedge y_1 \in \mathcal{C}(u, v)$ holds. Definition-free paths can be formalised using appropriate extensions of $\mathcal{I}_{[a,b]}$ and $\mathcal{S}_{[a,b]}$ [7].

We can see that these conditions resemble the declarative content of a CHR$^\vee$ goal. A constructive proof for satisfiability of such a goal will also yield candidate values for x and therefore the desired candidates for test inputs.

In this section we show a set of CHR$^\vee$ rules, which can be used to produce such a constructive proof or to show non-satisfiability. We will use the built-in constraints from Tab. 1. Note that $\exists y : x\ \mathcal{S}_{[s,e]}\ y$ is satisfiable for some fixed x if and only if the program terminates on x. Therefore our CHR$^\vee$-program cannot terminate on all possible goals.

Table 1. Built-in constraints

Constraint	Semantics
`edge(U,V)`	$(u, v) \in E$
`reachable(U,V)`	$(u, v) \in E^+$
`body(U,X,Y)`	$x \ \mathcal{B}(u) \ y$
`cond(U,V,X)`	$x \in \mathcal{C}(u, v)$
`deffree(U,W,V)`	all paths from u to w are definition-free regarding v
`onallpaths(U,W,V)`	all paths from u to w proceed from u via v to w
`value(X,Var,Val)`	`Val` is the value of variable `Var` in state `X`

First of all, we can construct $x \ \mathcal{S}_{[a,b]} \ y$ using $\mathcal{B}$ and $\mathcal{I}$, as already indicated in Sect. 2: For any given $x, y \in S$, $x \ \mathcal{S}_{[a,b]} \ y$ holds if and only if at least one of the following cases applies:

- $a = b \wedge x \ \mathcal{B}(a) \ y$.
- $\exists y_1, y_2 \in S : x \ \mathcal{B}(a) \ y_1 \wedge y_1 \ \mathcal{I}_{[a,b]} \ y_2 \wedge y_2 \ \mathcal{B}(b) \ y$.

This is implemented in Rule `spec_to_ispec` in Lst. 1, representing $x \ \mathcal{S}_{[a,b]} \ y$ as `spec(A,B,X,Y)` and $x \ \mathcal{I}_{[a,b]} \ y$ as `ispec(A,B,X,Y)`.

Now we can concentrate on solving $x \ \mathcal{I}_{[a,b]} \ z$, which holds if and only if at least one of the following cases applies:

- There is an edge from a to b and $x \in \mathcal{C}(a, b) \wedge x = z$ holds.
- The node n is a sucessor of a so that b can be reached from n and $\exists y \in S : x \in \mathcal{C}(a, n) \wedge x \ \mathcal{B}(n) \ y \wedge y \ \mathcal{I}_{[n,b]} \ z$ holds.

They are implemented in Rule `step_fwd`, providing a way of constructing a solution by iteratively stepping through the program in a forward direction. Analogously, Rule `step_bwd` implements iteratively stepping backwards through the program.

However, in some cases there are nodes which are traversed on every path from a to b. In the CFG shown in Fig. 1(a), for example, any path from node 3 to node 6 will traverse nodes 3, 5, 2 and 6. In case such a node n is known, we can split paths from a to b into two sub-paths from a to n and from n to b.

This split is implemented in Rule `split`. Note that for any pair of a and b there may be several different nodes n which are traversed on every path from a to b. However it can be shown that the choice of n is not relevant for the solution and a program consisting only of Rule `split` is confluent. In our prototype, we derive candidate split nodes using an efficient algorithm by Lengauer and Tarjan [10, 7].

The two subpaths introduced by `split` are not independent as the output of the first subpath is connected to the input of the second subpath via the body of n. However, we can exploit monotony and preservation properties of $\mathcal{I}_{[a,b]}$ to propagate information across these subpaths. One interesting property is the preservation of variable values. If we know that all paths from a to b are definition-free regarding some variable v, then we know that the value of v cannot change along any execution path from a to b. This notion is implemented in Rule `prop_var`.

Note that already a program consisting only of Rule `spec_to_ispec` and either Rule `step_fwd` or Rule `step_bwd` would implement the whole theory of $\mathcal{I}$ and $\mathcal{S}$. The Rules `prop_var` and `split` are mainly required for improved search performance.

Therefore, in our actual implementation we allowed selectively disabling any of the latter rules and chosing at least one of the stepping rules. Such a configuration is called a *strategy*.

The `body/3` and `cond/3` built-in constraints are implemented as Prolog clauses, in turn using a custom-built solver for finite domain constrains. The latter is implemented in CHR as well, combining classic domain-filtering solution strategies with

```
spec_to_ispec @ spec(U,W,X,Z) <=>
    (U=W, body(U,X,Z));
    (body(U,X,Y1), ispec(Y1,U,W,Y2), body(W,Y2,Z)).
prop_var @ ispec(U,W,X,Y) ==> reachable(U,W), deffree(U,W,V) |
    value(X,V,V1), value(Y,V,V2), V1=V2.
split @ ispec(X,U,W,Z) <=> reachable(U,W), onallpaths(U,W,V) |
    ispec(X,U,V,Y), body(V,Y,Z), ispec(Y,V,W,Z).
step_fwd @ ispec(X,U,W,Z) <=>
    (edge(U,W), X=Z, cond(U,W,X));
    (edge(U,V), reachable(V,W),
     cond(U,V,X), body(V,X,Y), ispec(V,W,Y,Z)).
step_bwd @ ispec(X,U,W,Z) <=>
    (edge(U,W), X=Z, cond(U,W,X));
    (edge(V,W), reachable(U,V),
     ispec(X,U,V,Z), body(V,Z,Y), cond(V,W,Z)).
```

Listing 1: $\text{CHR}^\vee$-Implementation

axiomatic rules. This way inconsistencies such as $a < b, b < a$ can be detected more efficiently using the transitivity and irreflexivity of $<$ than by pure domain filtering.

For testing deterministic selection of test inputs is not desirable. Therefore we implemented a probabilistic version of the program, in which in every step first the Rules `spec_to_ispec`, `split` and `prop_var` are applied exhaustively, if enabled. After that, stepping forward or backward is selected with probability p and completion of a subpath by direct edge traversal is selected with probability $1 - p$, if applicable. If both `step_fwd` and `step_bwd` are enabled — called a *mixed* configuration — they are applied with the same probability. The construction of a path now becomes a Bernoulli experiment, favouring shorter paths, but this bias can be at least partially compensated by varying p.

If stepping through a further node is selected, one of the alternative successors is selected according to a slightly skewed uniform distribution. Here nodes inside a loop are favoured over nodes by which the loop is exited in order to avoid constant and minimal mean iteration counts for inner loops [7].

4 Implementation Issues

Although the theoretical construction theorems are quite similar to the declarative semantics of CHR rules, it was not possible to transform them directly.

For example, the desired probabilistic behaviour as described in Sect. 3 can be modelled with Probabilistic CHR (PCHR) [11], but only by splitting up alternatives of Rules `step_fwd` and `step_bwd` into individual rules and thereby giving up the connection to the declarative semantics of $\text{CHR}^\vee$. Additionally, the second alternative of these rules has to be boxed into another constraint to delay the selection of the successor respectively predecessor node. Otherwise, each of the possible intermediate nodes would weigh in as an alternative to closing a subpath, making the probability of stepping dependent on the number of available intermediate nodes.

From a first look it seems that CHRiSM [12] provides a better integration of PCHR with $\text{CHR}^\vee$. Unfortunately, it was not available during implementation of the prototype.

Also, the first alternatives of Rules `step_fwd` and `step_bwd` are actually only present in the first of two cases of the stepping theorem, namely the case $(u, w) \in E$. So in these rules, `edge(U,W)` actually is a guard, but as $\text{CHR}^\vee$ does not allow individual guards for alternatives, the constraint had to be moved into the body

of the first alternative. Operationally, this does not make a difference as the first alternative does fail if there is no edge from u to v, just as if it had not been activated due to the guard.

Further, the second alternatives of Rule `step_fwd` and `step_bwd` actually correspond to a comprehensive union over all cases of $(u, v) \in E, (v, w) \in E^+$. As `edge/2` is a built-in constraint, search over its solutions is not supported by the declarative semantics of CHR$^\vee$. The program again is only operationally correct and only because the underlying host, SWI Prolog, allows search on `edge/2`, respectively our search extension for PCHR considers all applicable instances of a rule.

Still, these constraint solvers would be hardly manageable without CHR at all. At 26 constraints in 126 rules for the built-in solver and 45 constraints — many of them part of the probabilistic selection implementation or used for debugging purposes — in 74 rules for the path construction solver, a manual implementation is not feasible.

A CHR rule expresses interdependencies of many constraints. These interdependencies are difficult to handle with the classical "separation-of-concerns" approach to limiting complexity. Therefore a CHR compiler taking the burden of keeping an implementation of that size consistent is a huge relief for the developer.

5 Evaluation

We have applied the program to several example programs and determined the strategies performing best and worst as shown in Tab. 2. During the experiments, the length of the constructed path and the time required for construction were recorded. The best strategy was determined based on a fit of a second-order polynomial to the data as well as the observed scatter. The best overall strategy is marked with a †-symbol.

Table 2. Comparison of Strategies

Program	Goal	best		worst
		without `split`	with `split`	
Fibonacci	feasible path	`step_bwd`†	`step_bwd`	mixed+`split`
Selection Sort	feasible path	`step_fwd`†	n/a	`step_fwd+split`
`strcmp` without `break`	result = 0	`step_bwd`†	n/a	mixed+`split`
`strcmp` with `break`	result = 0	mixed	`step_bwd`†	mixed+`split`
Insert into array	cover node	mixed	`step_bwd`†	`step_fwd+split`

Several of the strategies showed a notable scatter in runtime due to backtracking. In some cases, the runtime impact of backtracking is so high that only very short paths can be constructed in an acceptable time, as shown in Fig. 2(b).

6 Conclusions and Outlook

We have presented a novel approach to ATDG and its implementation in CHR$^\vee$. The approach supports several combinable path construction strategies, none of which is optimal, and a generic set of structural coverage criteria, including all industrial criteria for safety-critical software.

Unfortunately, none of the currently available CHR compilers is formally qualified according to industry standards. Such a qualification is necessary for acceptance of development tools in the context of safety-critical applications. In contrast, the

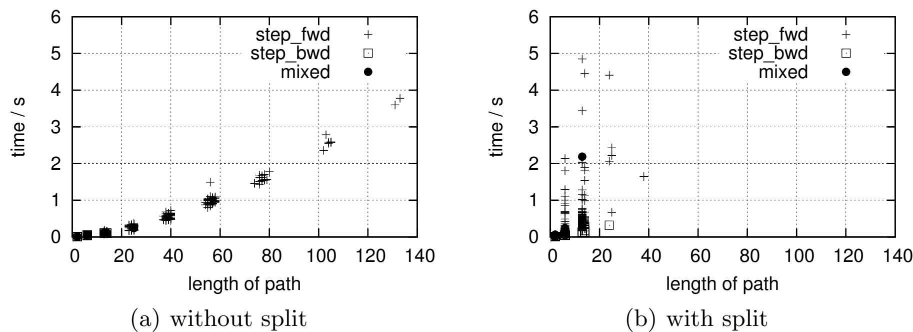

Fig. 2. Performance measurements for Selection Sort

argument in favour of CHR-based tools is strongly supported by the close connection of theory and practice in CHR and most of its variants.

Further research will focus on the possible application of CHRiSM as well as on integration with results from static program analysis such as abstract interpretation. Also, several projects for application on real-life safety-critical software are already defined and a toolchain for the language C based on the method is currently under development.

References

1. Frühwirth, T., Abdennadher, S.: Essentials of Constraint Programming. Springer Verlag (2003)
2. Frederick P. Brooks, J.: The Mythical Man-Month: Essays on Software Engineering. Addison-Wesley (1995)
3. Rapps, S., Weyuker, E.J.: Data flow analysis techniques for test data selection. In: ICSE '82: Proceedings of the 6th international conference on Software engineering, IEEE Computer Society Press (1982) 272–278
4. Hamlet, R.: Random testing. In Marciniak, J., ed.: Encyclopedia of Software Engineering. Wiley (1994) 970–978
5. Gerlich, R., Gerlich, R., Boll, T.: Random testing: From the classical approach to a global view and full test automation. In: RT '07: Proceedings of the 2nd international workshop on Random testing, ACM (2007) 30–37
6. Denise, A., Gaudel, M.C., Gouraud, S.D.: A generic method for statistical testing. In: Proceedings of the 15th IEEE International Symposium on Software Reliability Engineering (ISSRE), IEEE (2004) 25–34
7. Gerlich, R.: Verallgemeinertes Rahmenwerk zur constraintbasierten Testdatenerzeugung aus Programmflussgraphen. PhD thesis, Faculty of Engineering and Computer Science, University of Ulm, Germany (2009)
8. Gotlieb, A., Botella, B., Rueher, M.: Automatic test data generation using constraint solving techniques. SIGSOFT Softw. Eng. Notes **23**(2) (1998) 53–62
9. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, ACM (2005) 213–223
10. Lengauer, T., Tarjan, R.E.: A fast algorithm for finding dominators in a flowgraph. ACM Trans. Program. Lang. Syst. **1**(1) (1979) 121–141
11. Frühwirth, T., Pierro, A.D., Wiklicky, H.: Probabilistic Constraint Handling Rules. In: WFLP 2002, 11th International Workshop on Functional and (Constraint) Logic Programming. Volume 76 of Electronic Notes in Theoretical Computer Science., Elsevier (November 2002) 1–16
12. Sneyers, J., Meert, W., Vennekens, J.: CHRiSM: CHance Rules induce Statistical Models. In: Proceedings of the Sixth International Workshop on Constraint Handling Rules (CHR'09). (July 2009) 62–76