Path Constraint Construction with Lookahead

Ralf Gerlich

ralf.gerlich@bsse.biz*

Abstract: One of the basic tasks of constraint-based testdata generation is the selection of paths to be executed and the construction and solution of the associated path constraint. Here the problem of infeasible paths is not negligible. In this paper a new constraint-based method applicable to general control-flow-graphs is proposed for selecting feasible paths and at the same time constructing the associated path constraint. The method also allows the flexible application of methods for predicting control flow, thereby enhancing the identification of infeasible paths in advance.

1 Introduction

When automating software test the same tasks have to be carried out as for manual software test, among them the selection of test data. While randomized techniques allow unbiased coverage of the input domain, they in some cases fail in achieving required code coverage. This problem arises mostly from the fact that some specific paths through the system-under-test (SUT) are related to only a small subset of the input domain, and therefore are difficult to hit randomly.

Gotlieb[GBR98] originally introduced a constraint-based concept for generation of test data covering a given statement in the SUT, based on the semantics of control-flow constructs, such as `while` and `if`. The method can examine sequential constructs in parallel, thereby predicting parts of the control flow and allowing for elimination of some infeasible paths in advance. The concept is Turing-complete in theory, but in practice a considerable amount of effort is necessary to extend it to languages used in industrial practice. Sy and Deville[SD01] modified this method so that it applies to control-flow-graphs (CFGs), however without path prediction.

An alternative method incorporating both applicability to CFGs and path prediction, is introduced in this paper. The paper has three further parts. In Section 2 the method is described, followed by a short report on a first implementation in Section 3. Finally, conclusions are drawn in Section 4.

*This work is supported by a doctorate scholarship of the University of Ulm, Germany

1.1 Constraint Handling Rules

Constraint Handling Rules (CHR) is a declarative high-level language for specifying customised constraint solvers[Frü98]. A CHR program consists of a set of guarded rules which are used to transform a given constraint goal consisting of user-defined and builtin constraints until it is solved.

The rule forms available in CHR are simplification, written as $H \Leftrightarrow G|C$, and propagation, written as $H \Rightarrow G|C$, with the *head* H being a conjunction of user-defined constraints, the *guard* G a conjunction of built-in constraints and the *body* C a conjunction of built-in and user-defined constraints.

Simplification rules model replacement of user-defined constraints by logically equivalent constraints. Propagation introduces new, logically redundant constraints, which may lead to further simplification, e.g. in case of $A \leq B, B \leq C \Rightarrow A \leq C$. The language CHR$^\vee$ extends the theory of CHR by search capabilities by introducing disjunction in rule bodies.

2 The Method

In terms of constraint programming semantics, the problem at hand is best described as constructively proving the existence of a feasible path through the CFG from some node a to some node b. A path is *feasible* if and only if there is some input to a which can lead to execution of the path, otherwise it is infeasible. Consider, for example, the following program operating on two input arrays a and b, both with length l:

```
equal:=true;
for i=1 to l do
        if a(i)!=b(i) then equal:=false; break; end if
end for
if equal then print "Arrays are equal!"; end if
```

Given that a path is desired in which the `print`-statement is executed, any path via the statement `equal:=false` is infeasible. As a, b and l are to be found according to the desired path, there is a potentially infinite set of infeasible paths. Large subsets of infeasible paths can be excluded by a proper search strategy.

The input to the algorithm shall be given in the form of a CFG, consisting of a set of nodes and a set of edges. In the following $a \rightarrow b$ shall express the fact that there is an edge from node a to node b. Further $a \rightarrow^+ b$ expresses that there is a non-empty sequence of edges from a to b. The nodes are labelled by their code text, and the edges are labelled by their associated conditions. An example for an implementation of the greatest-common-divisor-algorithm is shown in Figure 1.

The constraint $i\ B_n\ o$ shall be fulfilled if and only if the body of node n transforms the input i to the output o. Similarly, $o\ C_{a \rightarrow b}\ i$ is fulfilled if and only if the memory states o and i are equal and fulfill the condition for edge $a \rightarrow b$.

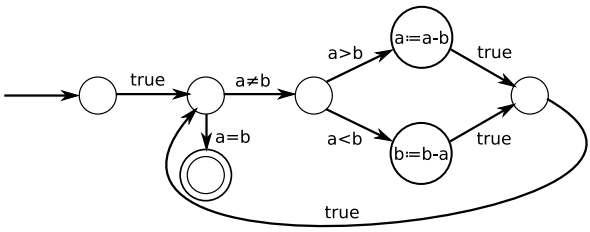

Figure 1: Labelled CFG of a gcd-Implementation

To express the feasibility constraint, a constraint $path\,(a, o_a, i_b, b)$ is introduced, which is fulfilled if and only if there is a path from a to b on which o_a, the output of a, is transformed to i_b, the input of b. By not using the input of a and the output of b directly, the calculations are simplified: o_a resp. o_b are easily found from i_a resp. i_b by the bodies of a resp. b.

The constructive proof is based on a divide-and-conquer strategy, where the path is either completed by a direct edge or split at some node n lying on some path from a to b. This strategy is described by the rules *Divide* and *Conquer* in the following CHR^v program. Note that these two rules actually have to be read as a disjunction for the program to be declaratively correct.

$$
\begin{array}{lcl}
\textbf{PredictVar} \mathbin{@} path\,(a, o_a, i_b, b) & \Rightarrow & v \in constvar\,(a, b) \mid o_a.v = i_b.v\ . \\
\textbf{PredictPath} \mathbin{@} path\,(a, o_a, i_b, b) & \Leftrightarrow & n \in allpaths\,(a, b) \\
 & \mid & path\,(a, o_a, i_n, n)\,, path\,(n, o_n, i_b, b)\,, \\
 & & i_n\ B_n\ o_n\ . \\
\textbf{Conquer} \mathbin{@} path\,(a, o_a, i_b, b) & \Leftrightarrow & a \rightarrow b \mid o_a\ C_{a \rightarrow b}\ i_b\ . \\
\textbf{Divide} \mathbin{@} path\,(a, o_a, i_b, b) & \Leftrightarrow & a \rightarrow^+ n, n \rightarrow^+ b \\
 & \mid & path\,(a, o_a, i_n, n)\,, path\,(n, o_n, i_b, b)\,, \\
 & & i_n\ B_n\ o_n\ .
\end{array}
$$

The rule *PredictVar* handles the case of variables which cannot be modified between on any path from a to b. This is the case when the definition nodes for v are either not reachable from a or cannot reach b, and is expressed by the constraint expression $v \in constvar\,(a, b)$.

The rule *PredictPath* predicts control flow by splitting the query at nodes which are traversed in any path from a to b, if any such node exists, using the builtin constraint $n \in allpaths\,(a, b)$. A special case of such nodes are the so-called dominators, which can be found efficiently from the CFG[LT79], and can be extended to the general case by using an appropriately chosen subset of the CFG with a as entry and b as exit node.

In the example, *PredictPath* can predict that the `if`-statement containing the `print`-statement must be executed. One of the predecessors of the `if` is infeasible due to the conflicting assignment, leaving only those paths exiting the loop normally, all of which are feasible.

3 Implementation

A prototype was implemented in Java for an imperative subset of Java. The builtin solver was implemented in JCHR[VWSD05] and specifically optimised for both solving and early inconsistency detection. To avoid recurring search of path subsets, *Divide* was refined to only select direct successors of a resp. predecessors of b.

Preliminary measurements on the generation rate of criteria-fulfilling test-inputs have been executed. They show that the new algorithm is about 10^5 times faster than random test-data generation for the hard-to-hit cases, while in simpler cases it is about 10^3 times slower.

4 Conclusions and Outlook

The algorithm presented is a generally applicable algorithm dealing directly with the CFG and performing control-flow prediction. However, to avoid bias introduced from the user-specified goals and to allow statistical evaluation of the results based on large test-input sets, it is recommended to use the algorithm only as a second complementary step to random test data generation.

Further work will focus on statistically controlling path length and finding mutually exclusive choices of path split-points, as well as on systematic performance measurement and extension of the supported language set in the prototype.

References

[Frü98] Thom Frühwirth. Theory and Practice of Constraint Handling Rules. *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, 37(1-3):95–138, October 1998.

[GBR98] Arnaud Gotlieb, Bernard Botella, and Michel Rueher. Automatic test data generation using constraint solving techniques. In *ISSTA '98: Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, 1998.

[LT79] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, 1979.

[SD01] Tran Nguyen Sy and Yves Deville. Automatic Test Data Generation for Programs with Integer and Float Variables. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE)*, 2001.

[VWSD05] Peter Van Weert, Tom Schrijvers, and Bart Demoen. K.U.Leuven JCHR: a user-friendly, flexible and efficient CHR system for Java. In Tom Schrijvers and Thom Frühwirth, editors, *Proceedings of the Second Workshop on Constraint Handling Rules (CHR 2005)*, pages 47–62, Oct 2005.