2016 IEEE International Conference on Software Quality, Reliability and Security

A Method for Pruning Infeasible Paths via Graph Transformations and Symbolic Execution

Romain Aissat, Marie-Claude Gaudel, Frédéric Voisin, Burkhart Wolff

LRI, Univ Paris-Sud, CNRS, CentraleSupélec, Université Paris-Saclay, France
aissat@lri.fr

***Abstract*—Path-biased random testing is an interesting alternative to classical path-based approaches faced to the explosion of the number of paths, and to the weak structural coverage of random methods based on the input domain only. Given a graph representation of the system under test a probability distribution on paths of a certain length is computed and then used for drawing paths. A limitation of this approach, similarly to other methods based on symbolic execution and static analysis, is the existence of infeasible paths that often leads to a lot of unexploitable drawings.**

We present a prototype for pruning some infeasible paths, thus eliminating useless drawings. It is based on graph transformations that have been proved to preserve the actual behaviour of the program. It is driven by symbolic execution and heuristics that use detection of subsumptions and the abstract-check-refine paradigm. The approach is illustrated on some detailed examples.

I. Introduction

White-box, path-based, methods are well-known techniques for testing programs. Given the control-flow graph (CFG) of the program under test, generation of a test suite is viewed as the process of (1) selecting a collection of *paths of interest*, then (2) providing, for each path in the collection, concrete values for the program parameters that will make the program follow exactly that path during a run.

Paths of interest are the ones that allow to fulfill some coverage criterion related to elements of the graph (vertices, branches, paths, etc). Paths can be selected according to a given distribution of probability over these elements. Both approaches can be combined as in *structural statistical testing* [1, 2].

The second step requires to produce for each path its *path predicate*, which is the conjunction of all the constraints over the input parameters that must hold for the system to run along that path. This is done using symbolic execution techniques [3]. Then, constraint-solving is used to compute concrete values to be used as inputs for testing the program. If for no values the path predicate evaluates to true, the path is infeasible. It is very common for a program to have infeasible paths and such paths can largely outnumber feasible paths. Infeasible paths selected during the first step do not contribute to the final test suite, and there is no better choice than to select another path, hoping for its feasibility. Handling infeasible paths is the serious limitation of structural methods since such methods can spend most of the time selecting useless paths.

In path-biased random testing [1], paths in the CFG are drawn according to a given distribution. This requires a global knowledge and combinatorial analysis of the graph, and disqualifies concolic-like methods (see Section VI). Path feasibility is checked in a second step. In [4], for each drawing yielding an infeasible path a new path was drawn, while trying to learn infeasibility patterns from the set of rejected paths. But the experimental results were not satisfactory. The prototype presented here improves the CFG of the program w.r.t. infeasibility: it builds a transformed graph which still over-approximates the set of feasible paths but includes fewer infeasible paths. Random drawing of paths is then performed on this new graph.

We use symbolic execution [3] and constraint solving to detect whether some paths are infeasible. To handle cyclic paths due to loops in the code we enrich symbolic execution with the detection of *subsumptions*: there is a subsumption when some node met during the analysis is a particular case of another node met previously. There is no need to explore its successors since they are subsumed by the successors of the subsumer.

The paper is organized as follows: Section II presents our graph transformations; Section III, our main algorithm and its heuristics; Section IV-A, an example; Section V, some experimental results. We discuss related works in Section VI and conclude in Section VII.

978-1-5090-4127-5/16 $31.00 © 2016 European Union
DOI 10.1109/QRS.2016.26

II. Background

A. Modelling Programs

Programs are modelled by *Labelled Transition Systems* (LTS). A LTS is a quadruple (L, l_0, Δ, F) where L is the set of program locations, $l_0 \in L$ is the entry point of the program and $F \subseteq L$ the set of final vertices, i.e. exit points. $\Delta \subseteq L \times \mathit{Labels} \times L$ is the transition relation, with Labels being a set of labels whose elements represent the basic operations that can occur in programs. In this paper, a label can be as follows:

- Skip, used for edges associated with break, continue or jump statements,
- Assume ϕ, where ϕ is a boolean expression over Vars, the set of program variables,
- Assign v e, where v is a program variable and e an arithmetic expression over elements of Vars.

Vertex l_0 has no incoming edges and elements in F have no outgoing edges. We suppose that all vertices are reachable from l_0 and reach an element in F. We write $l \overset{label}{\rightarrow} l'$ to denote the transition from $l \in L$ to $l' \in L$ executing the operation corresponding to $label \in \mathit{Labels}$.

If-Then-Else and While-loops are encoded by edges with Assume labels. There is no block structure, assuming all variables to be defined at the topmost level. We call D the domain of program variables. A *program state* is a function $\sigma : \mathit{Vars} \to D$. In the examples, D is the set of integers. This could be extended to arrays, records and other constructs. Actually, the approach presented here can be combined with existing memory models such as [5, 6], yielding new kinds of formulae, but the notions of subsumption and abstraction will remain relevant. The real limitation comes from the constraint solver in use.

B. Symbolic Execution

A symbolic variable is an indexed program variable. The set $\mathit{Vars} \times \mathbb{N}$ of symbolic variables is noted $\mathit{SymVars}$. *Configurations* are pairs (s, π), where s, the *store*, is a function from program variables to indexes which maps program variables to symbolic variables, and π, the *path predicate*, is a formula over symbolic variables which is the conjunction of constraints met during symbolic execution. The set of configurations is noted by $\mathfrak{C}$. A configuration is satisfiable if and only if its path predicate is satisfiable.

We represent symbolic execution by a function $\mathrm{SE} : \mathfrak{C} \times \mathit{Labels} \to \mathfrak{C}$, and define it as follows:

$$\mathrm{SE}\ c\ l = \begin{cases} c & \text{if } l = \mathrm{Skip} \\ (s,\ \pi \wedge \phi_s) & \text{if } l = \mathrm{Assume}\ \phi \\ (s',\ \pi \wedge (v, s'(v)) = e_s) & \text{if } l = \mathrm{Assign}\ v\ e \end{cases}$$

where:

- e_s (resp. ϕ_s) is the expression obtained by substituting in e (resp. ϕ) every occurrence of any variable v by $(v, s(v))$,
- s' is obtained by updating s in such a way that the symbolic variable $(v, s'(v))$ is fresh for c.

C. Subsumption

Given a store s, a program state $\sigma : \mathit{Vars} \to D$ and a symbolic state $\sigma_{sym} : \mathit{SymVars} \to D$, σ and σ_{sym} are said to be consistent with s, noted $\mathit{cons}(s, \sigma, \sigma_{sym})$, if

$$\forall\ v \in \mathit{Vars}.\ \sigma(v) = \sigma_{sym}((v, s(v)))$$

Given an expression e over program (resp. symbolic) variables and a program state σ (resp. a symbolic state σ_{sym}), we write $e(\sigma)$ (resp. $e(\sigma_{sym})$) the evaluation of e in σ (resp. in σ_{sym}). The set of program states represented by a configuration $c = (s, \pi)$, is defined as:

$$\mathit{States}(c) = \{\sigma.\ \exists\ \sigma_{sym}.\ \mathit{cons}(s, \sigma, \sigma_{sym}) \wedge \pi(\sigma_{sym})\}$$

If the path predicate of a configuration is unsatisfiable, its set of states is empty. A configuration c is subsumed by another configuration c' if $\mathit{States}(c) \subseteq \mathit{States}(c')$.

Symbolic execution is monotonic with respect to this definition of subsumption. There is no need to explore the successors of a subsumed point: they are subsumed by the successors of the subsumer. It follows that the set of feasible paths starting at the subsumee is a subset of the set of feasible paths starting at the subsumer [7].

Since subsumption corresponds to a set inclusion, adding a subsumption to the symbolic execution tree often comes at the price of introducing infeasible paths into it. A challenge is thus to accept only subsumptions that introduce a reasonable number of infeasible paths.

D. Abstraction

Unfolding loops in a LTS might yield an infinite symbolic execution tree. The use of subsumptions allows to keep a finite representation. Every time a loop header is reached along a symbolic execution path, subsumptions are checked with previous occurrences of the same loop header on the path. The configurations at the subsumer and subsumee record changes of the (symbolic) values of variables along that path. Usually, the symbolic values of some variables differ between both configurations and subsumption is not possible without abstracting the configuration for the subsumer, that is forgetting part of the information about its state. There are various ways to weaken its path predicate: remove a set of conjuncts, or compute a weaker form of the path predicate that

would be implied by the current path predicates of both configurations [8, 9]. This amounts to compute some kind of invariant for that loop.

Once abstraction has been performed, the new configuration for the subsumer must be propagated in its subtree. This could rule out existing subsumptions involving some successors. We must also check that the abstraction propagated to the subsumee is still subsumed by the abstracted subsumer. When a conflict exists, we keep the existing subsumptions and discard the new abstraction. If no abstraction is retained with an occurrence on the path of the same loop header, the loop is unfolded by symbolic execution, hoping for a future subsumption.

Since abstraction discards part of information about the symbolic values of variables at a program point, it possibly adds infeasible paths with respect to the set of paths one would have with classical symbolic execution.

E. Limiting Abstractions

To prevent some abstractions, or to record abstraction banned by some kind of refinement [9], predicates can be attached to configurations at loop headers. They act as safeguard against abstractions: only abstract configurations that imply the additional predicate can be retained. In Section III we use a weakest-precondition calculus to obtain such predicates. To make sure that no feasible path is discarded by attaching the predicate to a configuration, the predicate must hold for all states described by the configuration. This predicate is not part of the configuration and is not propagated to successors.

III. Algorithm

Our algorithm transforms a given CFG into one with fewer infeasible paths. It takes a LTS S and an initial configuration c as inputs and builds an intermediate structure, we call it a *red-black graph*, which is turned back into a LTS when the analysis is over.

A. Red-Black Graphs

A red-black graph $\mathcal{RB}$ is a 6-uple $(\mathcal{B}, \mathcal{R}, \mathcal{S}, \mathcal{C}, \mathcal{M}, \Phi)$:

- $\mathcal{B} = (L, l_0, \Delta, F)$, the *black part*, is the input LTS. It is never modified during the analysis,
- $\mathcal{R} = (V, r, E)$, the *red part*, is a rooted graph (a LTS without labels) that represents the symbolic execution tree built so far; $V \subseteq L \times \mathbb{N}$ is the set of *red vertices*, which are indexed versions of elements of L representing occurrences of locations of $\mathcal{B}$ met during the analysis; $r \in V$ is its root; $E \subseteq V \times V$ is its set of edges.
- $\mathcal{S} \subseteq V \times V$ is the subsumption relation between red vertices computed so far,
- $\mathcal{C}$ is a function from red vertices to configuration stacks: a vertex can have multiple configurations (reflecting different choices of abstraction) during the analysis. Given a red vertex rv, we call *configuration of* rv the configuration currently on top of the stack associated with rv,
- $\mathcal{M}$, the marking, is a boolean function over red vertices recording partial information about unsatisfiability: $\mathcal{M}(rv)$ is $true$ only if the configuration of rv has been proved unsatisfiable. If $\mathcal{M}(rv)$ is $false$, nothing is known about the satisfiability of rv and it is treated as if it is satisfiable. Symbolic execution stops at vertices for which $\mathcal{M}$ holds.
- Φ is a function from red vertices to formulae over program variables recording the predicates used for limiting abstractions (see Section II-E).

The algorithm maintains a global data structure rvs of vertices to visit. In our prototype, rvs is a stack built according to a DFS traversal of the LTS. It starts with the original LTS and an initial configuration c whose store maps each program variable to a symbolic variable and whose path predicate is a user-provided formula (precondition of the program). All formulae must belong to the logic supported by the constraint solver.

B. Building the Red-Black Graph

We outline the algorithm that is fully described in [10]. Its main function is a loop that runs until there are no more red vertices to visit in the stack rvs.

The analysis starts with $\mathcal{RB}$ in the following state:

- $\mathcal{B}$ is S, the LTS under analysis,
- the root r of $\mathcal{R}$ is the couple $(l_0, 0)$, which is also the only element of V,
- E and $\mathcal{S}$ are empty, since the red graph is empty,
- $\mathcal{C}(r)$ is a one-element stack containing the initial configuration c, $\mathcal{M}(r)$ is *false* and $\Phi(r)$ is *true*

and rvs contains r only.

If rvs is not empty, its first element, called rv, is popped. If the configuration of rv is marked as unsatisfiable, the symbolic execution halts along that path. Otherwise, the path p from r to rv in $\mathcal{R}$ is recovered.

When none of the special cases below apply, a partial unfolding of $\mathcal{R}$ is performed by a symbolic execution step at rv; $\mathcal{M}$, $\mathcal{C}$ and Φ propagate to the immediate successors as follows. Let v be the black (i.e. unindexed) counterpart of rv: for every transitions $v \overset{label}{\rightarrow} l$ in Δ:

- the edge $(rv, (l, i))$ is added to E, where i is a fresh index for location l,
- the configuration SE $(\texttt{top}\ (\mathcal{C}\ rv))$ *label* (see Section II-B) is pushed on $\mathcal{C}$ (l, i),
- if *label* is of the form $\mathrm{Assume}\ \phi$, with ϕ being *false* (l, i) is marked in $\mathcal{M}$. If ϕ is neither *false* nor *true*, a constraint solver is called to check the satisfiability of the new configuration: (l, i) is marked in $\mathcal{M}$ only if the solver proves it is unsatisfiable,
- Φ is updated to associate *true* with (l, i).

Successors are then pushed onto rvs to be processed in the next iterations of the loop.

Final locations and faulty abstractions: If rv corresponds to a final location of $\mathcal{B}$, the algorithm checks the feasibility of p from the initial configuration. If p turns out to be infeasible, this could come from an abstraction made along p, at some vertex rv': weakening its path predicate turns p to feasible. A "refine-and-restart" phase is triggered. The refine-part consists in searching back in p for some red vertex rv' whose stack of configurations contains two (consecutive) configurations c_i and c_j such that the suffix of p starting at rv' is infeasible from c_i but was feasible from c_j: configuration c_j corresponds to the faulty abstraction. Let p' be the subpath of p going from rv' to its first successor in p whose configuration is unsatisfiable, and ϕ the weakest precondition of *false* w.r.t. the trace of p'. ϕ is our limiting predicate and is joined to $\Phi(rv')$. Next, the configuration of rv' prior to the faulty abstraction is restored and its subtree is destroyed, i.e. $\mathcal{RB}$ is updated as follows:

- configuration stacks of successors of rv' are removed from $\mathcal{C}$,
- subsumptions involving rv' or any of its successor are removed from $\mathcal{S}$,
- successors of rv' marked in $\mathcal{M}$ are unmarked,
- entries of Φ involving successors of rv' are removed,
- every edge starting or ending in a successor of rv' is removed from E,
- and every successor of rv' is removed from rvs.

Finally, rv' is pushed on top of rvs so that the analysis restarts at rv', now strengthened with ϕ.

In our prototype, this refine-restart part is controled by a user switch: without it, when a final vertex is reached the algorithm simply selects the next vertex to visit in rvs; building the final LTS is faster but it often keeps many infeasible paths due to loose abstractions. With it, the risk is that no better abstraction is found unfolding the loop forever. Our current method of abstraction does not learn from the safeguard conditions: the subsumption is simply postponed in the hope that a more accurate abstraction can be found later. A compromise is to keep the refine-restart and bound the maximal length of paths or unfoldings, but experimenting with learning abstraction methods is definitely worth doing.

Finding abstraction: if rv is an occurrence of a loop header the algorithm tries to find a subsumption of rv with a red vertex, for the same black location, previously met along p. When such an occurrence rv' is found, the constraint solver is called to check if the configuration of rv' subsumes the one at rv. If yes, the subsumption is established and (rv, rv') is added to $\mathcal{S}$. If the answer is negative or *unknown*, we look for an abstraction for the configuration of rv' such that (1) it subsumes the configuration of rv and (2) it entails $\Phi(rv')$.

There are various ways to compute abstractions. Currently, we remove the first conjunct in the path predicate, check if $\Phi(rv')$ is entailed then check if the abstracted configuration subsumes the configuration of rv. If this is the case, the abstracted configuration is returned. Otherwise, we remove also the second conjunct, and so on. If $\Phi(rv')$ is no longer entailed, the search for an abstraction is cancelled. If there is no limiting predicate $\Phi(rv')$ and the path predicate becomes empty (equivalent to *true*), we end up with a trivial abstraction. This current greedy algorithm requires only a number of calls to the constraint solver that is linear in the initial number of conjuncts. In Section V, we present another method.

Once an abstraction is computed, we check that it can be propagated without invalidating existing subsumptions involving successors of rv'. If this is not the case, the subsumption at the loop header is cancelled and the search for an abstraction halts, since looser abstractions will not help. Otherwise, the abstraction is propagated by performing symbolic execution in the subtree rooted by rv' with the abstracted configuration pushed on its stack, and pushing newly computed configurations on the stacks of the successors of rv.

Depending on the order in which the LTS is traversed, a successor of rv' can have been already abstracted. To account for both abstractions, the path predicate of the configuration pushed on its stack is the conjunct of all sub-formulae common to the path predicates of both abstractions (the stores are identical). Propagation of an abstraction can also turn configurations from unsatisfiable to satisfiable. Corresponding vertices are unmarked and pushed back on the stack rvs. Once the abstraction has been propagated, the subsumption is added to $\mathcal{S}$.

This algorithm might not terminate. Our implementation takes the maximal red length, noted mrl, of symbolic paths as an additional parameter. Whenever the current symbolic path reaches this bound, the algorithm is not allowed to extend it further.

C. Building the new LTS

Once the analysis is over, $\mathcal{RB}$ is turned into a new LTS S' by removing from $\mathcal{R}$ the edges leading to marked vertices, replacing the targets of edges leading to subsumed vertices by their subsumers, then renaming vertices and labelling edges between red vertices with the label of the edge between their black counterparts. For red vertices where the analysis halted because of the mrl limit, if any, the edges whose target is not final are connected to the corresponding black vertex, i.e. to the original CFG. This trick and the fact that transformations on the red part never rule out (prefixes of) feasible paths ensure that S' preserves the feasible paths of S.

In [7] we present a formal model of our graph transformations, in which we proved two key properties: the preservation of computations along the paths of the original CFG and the preservation of its feasible paths.

IV. Example

We illustrate our algorithm on a simplification of a *merging sort*: traversing two arrays a and b, filling a third array with their elements in order. Instructions that use the values of the arrays do not influence the feasibility of paths which only depends on how indexes vary. The LTS in Figure 1 keeps only code related to the indexes ia and ib and their bounds la and lb. Vertex 4 has two edges, each one incrementing an index. At each traversal of the loop, any of these two edges can be chosen non-deterministically, making our program a traversal of two arrays in an arbitrary order. The first loop iterates on both arrays until one of them is exhausted. The other array is processed by one of the last two loops.

Although simple, the LTS contains a lot of infeasible paths. There exists seven "groups" of infeasible paths. For instance, a feasible path enters at most one of the last two loops and not entering either loop is feasible only when both input arrays are empty. All infeasibilities rely on dependencies between the three loops: the first loop cannot exhaust both arrays simultaneously and the edge used to exit it forces which one of the other loops is entered. See [10] for a full presentation of the example.

Notation: we write l^i the red vertex (l, i). In Figures 2 and 3, subsumptions are represented with dotted edges and vertices marked as unsatisfiable with a $\bot$ symbol.

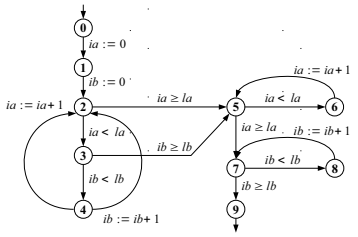

Figure 1: A simplified LTS for *merging sort*.

A. Merging Sort without Heuristic

Symbolic evaluation starts at 0^0 with configuration $(\{ia \mapsto ia_0, ib \mapsto ib_0, la \mapsto la_0, lb \mapsto lb_0\}, true)$ and proceeds up to 4^0 where rvs contains 2^1, 2^2, 5^1 and 5^0, with 2^1 on top. Two subsumptions occur: the first one between 2^1 and 2^0 requires an abstraction that removes $ia_1 = 0$ from 2^0. The abstraction is propagated from 2^0 to 2^1. Then 2^2 is subsumed by 2^0 after removing $ib_1 = 0$ from 2^0. Both abstractions are compatible, hence the new one is also propagated in the subtree rooted by 2^0.

Symbolic execution resumes at 5^1, pushing 6^0 and 7^0 on rvs. These vertices come from 3^0 itself linked to 2^0: their path predicate include $ia < la$. 7^0 is marked as unsatisfiable since the transition $5 \to 7$ adds $ia \geq la$ to its path predicate. Continuing with 6^0, we reach 5^2 (not in Figure 2 because it is later destroyed). Its subsumption by 5^1 removes $ia_1 < la_0$ from 5^1. Propagation of this abstraction removes $ia_1 < la_0$ from 5^1 and 7^0 making the latter satisfiable again. We unmark and visit it again, pushing its successors 8^0 and 9^0. Since 8^0 requires ib to be both lesser and greater or equal to lb, it is marked.

When reaching 9^0, a final location, we check if the unique (without subsumptions) path p in $\mathcal{R}$ from 0^0 to 9^0 is really feasible, or if feasibility is due to an abstraction: here, p becomes feasible by the abstraction at 5^1. It triggers a refine-and-restart phase. We extract from p the subpath p' starting at 5^1 and ending after the first infeasible step, namely $5^1 \cdot 7^0$. The weakest precondition of *false* along p', $ia < la$, is added as a limiting predicate for 5^1. At 5^1, the configuration before the abstraction is restored, its subtree destroyed (deleting 5^2) and 5^1 is pushed back on the stack. It is handled as before until the vertex (now named 5^3) where the abstraction needed for its subsumption by 5^1 is blocked by the new limiting predicate. The loop is unfolded again from 5^3 leading to 5^4. A subsumption $(5^4, 5^1)$ cannot be established but 5^4 is subsumed by 5^3 without abstraction: none of their configurations require ia to be lesser than la anymore. Final location 9^1 is

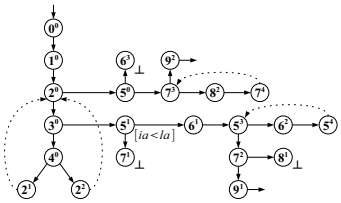

Figure 2: A first unfolding of the LTS of Figure 1.

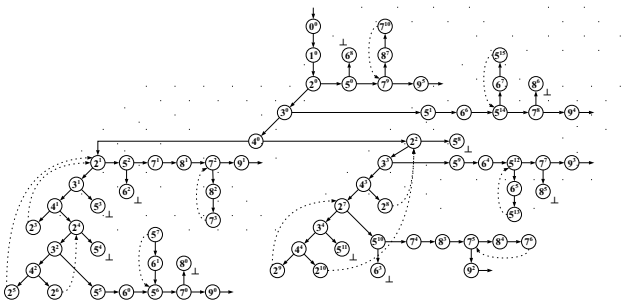

Figure 3: Final unfolding of the LTS of Figure 1

reached without refinement: the abstractions at 2^0 did not introduce any infeasible path. Finally the subtree rooted by 5^0 is built: subsumption $(7^4, 7^3)$ stops unfolding the third loop without abstraction of 7^3. Our traversal ends with the visit to 9^2.

In the LTS of Figure 2, two groups of infeasible paths have been eliminated: paths going through both ending loops, and paths exiting the first loop through $(3, ib \geq lb, 5)$ without going at least once through the second loop.

B. Merging Sort with Feasible Path Sets Comparisons

Subsumption is defined as an inclusion between sets of program states represented by configurations. Usually this inclusion is strict, and subsumptions add infeasible paths into the graph. The example shows that the first subsumption that can be established is not always the best one in terms of pruning infeasible paths. Besides, the refine-and-restart mechanism help us detect faulty abstractions only when it is the shortest path to a given final location that is made feasible by some abstraction.

Requiring equality instead of inclusion is not realistic but we can ask for inclusion *plus* equality of the sets of feasible paths *up to a certain depth*. In the example, constraining subsumptions to satisfy equality of sets of feasible paths up to a lookahead of two edges gives the LTS of Figure 3, with no infeasible paths. No refine-and-restart is needed for obtaining this new LTS. Whenever the algorithm attempts to subsume two occurrences of black vertex 2, comparing sets of feasible paths up to a depth of two suffices to deduce which index was incremented last in the main loop and to decide if the subsumption is precise enough. When attempting to subsume occurrences of 5 (resp. 7), this lookahead mechanism also prevents the abstraction to forget that ia (resp. ib) is lesser than la (resp. lb).

V. Results

We now present experimental results obtained for three programs: *merging sort*, *bubble sort* and *substring search* [10]. These examples are simple but they all have at least two unbounded loops with dependencies between the loops. *Merging sort* has three successive loops; the inner loop in *bubble sort* is always executed the same number of times for each traversal of the outer loop; *substring search* has two nested loops related by still another kind of dependency.

For each program, we build LTSs with different parameters: the way abstractions are computed, the depth of the lookahead and allowing restarts or not. Then, for each LTS we compare the number of paths of a given maximal length l with the corresponding number of feasible paths. Paths are counted and drawn using the Rukia library [1]. Given an LTS S with n paths of length at most l, we draw uniformly at random n distinct paths of length at most l, then count the number of feasible paths.

The second column of the tables shows which method for finding abstraction is used: with $a = 1$, abstractions are computed as shown in Section III; with $a = 2$, abstraction associates fresh symbolic values to program variables defined between the two candidates for subsumption, until the subsumption is stated. Abstracting configurations in this way requires to combine them appropriately at loop headers, during their propagation.

A mark in the third column indicates that restarts are enabled. Next columns give the number of paths of length at most l (all eliminated paths are infeasible) and the number of such feasible paths.

A. Merging Sort

Table I shows the results for *merging sort* using a lookahead of two. The first line gives the results for the LTS S from Figure 1. The second line shows the results for the LTS S' built from the red-black graph in Figure 3: no infeasible paths remain with the first method for finding abstractions. With the second method of abstraction and restarts enabled, the analysis does not terminate without bounding mrl. For *merging sort*, abstractions

Table I: Paths (P) and feasible paths (FP) in *merging sort*.

	a	r	$l=30$		$l=50$		$l=100$	
			P	FP	P	FP	P	FP
S			1224	140	24434	2300	~25.8M	~2.3M
S'	1	✓	140		2300		~2.3M	
	2	✓	140		4652		~23.9M	
	2		210		3271		~3.2M	

computed this way are too crude to not trigger a restart. Since our algorithm does not learn yet from safeguard conditions, restarts postpone subsumptions but unfolding the loop makes no good. Learning from the safeguard-conditions when computing abstractions would rule out such chains of restarts.

The 3rd line gives the results with the second method of abstraction and $mrl = 30$. The red part is no longer complete and red vertices that were not expanded are linked to the black part. Paths of length at most 30 are entirely in the red part. Longer paths can end in the black part. The 4th line shows the results for the LTS obtained with the second method of abstraction, disabling restarts: a fair number of infeasible paths are detected.

B. Bubble Sort

Table II gives the results for *bubble sort* (discriminating feasible paths for $l = 100$ was not tractable here. Additional column la gives the level of lookahead). With a lookahead of 0 or 1 and either method of abstraction, the only infeasible path removed is the one that does not enter the outer loop. With a lookahead of at least 2, S' better approximates the set of feasible paths. From 2 to 7, the algorithm produces the same LTSs, but at 8, more dependencies about traversals of the inner loop are discovered. The value of the lookahead cannot be increased too much in practice as comparing the sets of feasible paths of candidates is exponential.

Method of abstraction 1 performs worst here. In *bubble sort*, a variable keeps track whether any permutation occurs during a traversal of the inner loop. The value of this variable can be lost when abstracting the configuration at the entry of the inner loop, introducing infeasible paths. This does not happen with method 2. With both methods, the analysis terminates without bounding mrl. There remains a fair number of infeasible paths in S'. Unlike *merging sort*, the set of feasible paths of *bubble sort* is not a regular language because of the dependency between the number of traversal of its two loops. We believe that the results in Table II can be improved with

Table II: Paths and feasible paths in *bubble sort*.

	a	r	la	$l=30$		$l=50$		$l=100$
				P	FP	P	FP	P
S				1474	20	643692	217	~2.3×10^{12}
S'	1	✓	2	741		321962		~1.2×10^{12}
	2	✓		203		44504		~2.9×10^{11}
	1	✓	8	285		69457		~6.6×10^{10}
	2	✓		103		13249		~6.4×10^{9}

Table III: Paths and feasible paths in *substring*.

	a	r	la	$l=30$		$l=50$		$l=100$
				P	FP	P	FP	P
S				1433	87	195874	2108	~4.2×10^{10}
S'	1	✓	0	143		4180		9227464
	2	✓						
	1	✓	10	130		3803		8395424
	2	✓		98		2818		6217117

more accurate abstractions, but *bubble sort* clearly shows some limitations of our approach in its current state.

C. Substring

Table III shows the results for *substring*. The function takes as input strings s_1 and s_2, and returns *true* if s_2 is a substring of s_1. Loops present some dependencies: for example, let s be a (strict) prefix of s_2 of length l_s found to be a substring of s_1. Then s_2 has length at least l_s, and no latter iteration of the outer loop can return *true* without doing at least l_s comparisons. The set of feasible paths of *substring* is not a regular language.

With a lookahead of 0, both methods of abstraction produces the same LTS: the path that returns *false* when s_2 is empty is ruled out. With a lookahead of 2, the algorithm discovers the above property for $l_s = 1$. New LTSs are produced with a lookahead of 10: the algorithm discovers the property for $l_s = 1$ and $l_s = 2$. No bound is needed for mrl.

VI. Related work

The abundant literature on unfeasible paths is not reviewed here due to lack of space. For a recent account of issues in this area see [3, 11, 12].

A notable advance is concolic testing [13, 14] where actual execution of the program under test is coupled with symbolic execution. It reduces the detection of infeasible paths to those paths that go one branch further than some feasible one, alleviating the load of the solver and decreasing the number of paths to be considered.

This leads to coverage of all feasible paths. Some randomness can be introduced in the choice of the next branch as mentioned in [14], but the resulting distribution on paths suffers from the drawback of isotropic random walks, yielding unbalanced coverage of paths.

To ensure uniform random coverage of paths (and more generally a maximum minimal probability of covering components of a coverage criterion [1]), a global knowledge of the graph is needed. Concolic or similar dynamic approaches cannot be used: a global static analysis is required. It is the application scenario that motivated the work presented here.

We have taken inspiration from [8] and [15], where subsumptions, abstractions and interpolation are used to verify unreachability of selected error locations. Here, the problem we address is to preserve feasibility rather than infeasibility. This requires specific strategies for subsumptions and abstractions.

The problem of unbounded loops is a general issue for methods based on symbolic execution [9, 16]. It is generally treated, as we do, by searching for subsumptions, which doesn't always terminate. The red-black graph data structure we have defined makes it possible to deal with these non terminating cases.

Other potential application scenarios of the graph transformation proposed here include paths selection for satisfying coverage criteria of elements of a graph like branches [12] or mutation points [17].

VII. Conclusion

We address the problem of graph transformations that discard infeasible paths, preserving the behaviour of the program, with path-biased random testing in mind.

The size of the resulting graph and the length of paths are not a problem for drawing since the Rukia library we use for drawing paths scales up extremely well [1]. We expect the time of construction of the transformed graph to remain reasonable, thank to the progresses of symbolic execution tools and constraint solvers. The first results are encouraging since, on the current examples, the cost of this preprocessing phase is not an issue, and quite a significant number of infeasible paths are discarded, even with a basic set of heuristics.

We plan various improvements of the prototype aiming at the proportions of infeasible paths in the transformed graph. We are investigating better control of abstractions, taking advantage of safeguard conditions and interpolant propagations in the spirit of [9]. Using existing methods based on abstract interpretation and dataflow analysis [18, 19] will help finding abstractions by providing ranges of values for some variables, i.e. some kind of additional invariants. Besides, we plan to extend the range of application of our approach using memory models and additional language constructs.

References

[1] A. Denise, M.-C. Gaudel, S.-D. Gouraud, R. Lassaigne, J. Oudinet, and S. Peyronnet, "Coverage-biased random exploration of large models and application to testing," *STTT*, vol. 14, no. 1, pp. 73–93, 2011.

[2] P. Thévenod-Fosse and H. Waeselynck, "An investigation of statistical software testing," *Softw. Test., Verif. Reliab.*, vol. 1, no. 2, pp. 5–25, 1991.

[3] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *CACM*, vol. 56, pp. 82–90, 2013.

[4] S.-D. Gouraud, "Utilisation des structures combinatoires pour le test statistique," Ph.D. dissertation, Univ. Paris-Sud 11, 2004.

[5] M. Trtík and J. Strejček, "Symbolic memory with pointers," in *ATVA 2014*. Springer, 2014, pp. 380–395.

[6] F. Besson, S. Blazy, and P. Wilke, "A Precise and Abstract Memory Model for C Using Symbolic Values," in *12th Asian Symp. on Prog. Lang. and Systems*, ser. LNCS, vol. 8858. Springer, 2014, pp. 449 – 468.

[7] R. Aissat, F. Voisin, and B. Wolff, "Infeasible paths elimination by symbolic execution techniques: proof of correctness and preservation of paths," in *ITP'16*, ser. LNCS, vol. 9807, 2016.

[8] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan, "Abstractions from proofs," in *POPL'04*. ACM, pp. 232–244.

[9] K. L. McMillan, "Lazy annotation for program testing and verification," in *CAV'2010*, ser. LNCS, vol. 6174, pp. 104–118.

[10] R. Aissat, M.-C. Gaudel, F. Voisin, and B. Wolff, "Pruning infeasible paths via graph transformations and symbolic execution: a method and a tool," L.R.I., Univ. Paris-Sud, Tech. Rep. 1588, 2016. [Online]. Available: https://www.lri.fr/srubrique.php?news=33

[11] S. Bardin, M. Delahaye, R. David, N. Kosmatov, M. Papadakis, Y. L. Traon, and J. Marion, "Sound and quasi-complete detection of infeasible test requirements," in *8th ICST*. IEEE, 2015.

[12] M. Papadakis and N. Malevris, "A symbolic execution tool based on the elimination of infeasible paths," in *5th Int. Conf. on Soft. Eng.Advances, ICSEA 2010*. IEEE, 2010, pp. 435–440.

[13] N. Williams, B. Marre, P. Mouy, and M. Roger, "Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis," in *EDCC-5*, ser. LNCS, vol. 3463. Springer, 2005, pp. 281–292.

[14] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *ASE'2008*. IEEE, 2008, pp. 443–446.

[15] J. Jaffar, V. Murali, J. A. Navas, and A. E. Santosa, "TRACER: A symbolic execution tool for verification," in *CAV 2012*, ser. LNCS, vol. 7358. Springer, 2012, pp. 758–766.

[16] J. Jaffar, J. A. Navas, and A. E. Santosa, "Unbounded symbolic execution for program verification," in *RV'11*, ser. LNCS, vol. 7358. Springer, 2011, pp. 396–411.

[17] M. Papadakis and N. Malevris, "Mutation based test case generation via a path selection strategy," *Information & Software Technology*, vol. 54, no. 9, pp. 915–932, 2012.

[18] R. Bodík, R. Gupta, and M. L. Soffa, "Refining data flow information using infeasible paths," in *6th ESEC/5th ACM FSE Symp.* Springer-Verlag New York, 1997, pp. 361–377.

[19] V. Raychev, M. Musuvathi, and T. Mytkowicz, "Parallelizing user-defined aggregations using symbolic execution," in *25th SOSP*. New York: ACM, 2015, pp. 153–167.