Infeasible Paths Elimination by Symbolic Execution Techniques:
Proof of Correctness and Preservation of Paths

Romain Aissat, Frédéric Voisin, Burkhart Wolff

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

Abstract TRACER [8] is a tool for verifying safety properties of sequential C programs. TRACER attempts at building a finite symbolic execution graph which over-approximates the set of all concrete reachable states and the set of feasible paths.
We present an abstract framework for TRACER and similar CEGAR-like systems [2, 3, 5, 6, 9]. The framework provides 1) a graph-transformation based method for reducing the feasible paths in control-flow graphs, 2) a model for symbolic execution, subsumption, predicate abstraction and invariant generation. In this framework we formally prove two key properties: correct construction of the symbolic states and preservation of feasible paths. The framework focuses on core operations, leaving to concrete prototypes to "fit in" heuristics for combining them.
Keywords: TRACER, CEGAR, Symbolic Execution, Feasible Paths, Control-Flow Graphs, Graph Transformation

1 Introduction

TRACER [8] is a symbolic execution-based tool for verifying safety properties of imperative programs. TRACER tries to build from a program control-flow graph (CFG) a finite symbolic execution tree which over-approximates the set of reachable states. To this end, TRACER avoids the full enumeration of symbolic paths by learning from infeasible paths, i. e. from paths for which no input state exists allowing their execution. This learning phase uses *interpolants* for each program point. An interpolant is a formula characterizing a set of program states. If an interpolant allows to establish that a symbolic state is *subsumed* by a previous state in its path, TRACER annotates the symbolic execution tree by so-called subsumption links turning the tree into a graph. Thus, this annotated tree can represent infinite sets over-approximating the feasible paths.

Finding accurate approximations of the feasible paths of a program is of wide-spread interest for static analysis techniques, worst-time analyzers, code optimization and code-slicing techniques and structural test-case generation. Since in many programs the ratio of infeasible paths to feasible ones may be very high, a lot of computing power in a static analyser could be addressed at more rewarding targets, while dramatically improving the quality of results

by discarding information stemming from infeasible paths. Our motivation is in random-testing of imperative programs: there exist efficient algorithms that draw in a statistically uniform way long paths from very large graphs [4]. If the probability to find a feasible path from a (transformed) CFG is high, one could use these methods to randomly draw long paths, compute their path predicate, and test the program along an instance of the path predicate against a user-defined post-predicate (note that this method does not depend on user-defined loop-invariants). Thus, the method could be extended to an effective statistical structural (white-box) testing method.

When adapting TRACER mechanisms to our own purposes, we found that the presented proof sketches in the accompanying literature revealed a sensible gap to a formal proof development. We therefore built a formal theory in Isabelle/HOL of an abstract version of the TRACER algorithm, called ATRACER. ATRACER is a highly non-deterministic model of TRACER, consisting on five graph transformations of a so-called *red-black graph*, where the red part roughly corresponds to the analyzed symbolic execution tree gained by partial unfolding of the CFG and the black part is the initial CFG of the program. For ATRACER, two major theoretic results were established:

1. *correctness*: for every path in the new graph, there exists a path with the same trace in the original one, and
2. *preservation of feasible paths*: each transformation preserves the set of feasible paths. This very important property is often claimed in papers without a complete proof.

These results extend to an entire family of TRACER-like algorithms, which add to ATRACER specific heuristics in their goal to provide approximations of feasible paths of a given program. These heuristics fill in the non-deterministic "gaps" of ATRACER: which node to select, which interpolant to choose, which learned invariant to inject, etc. Note that our goal is not to provide a formal proof of TRACER implementation: heuristics aspects are not modeled, and ATRACER uses completely diferent data-structures. ATRACER is a rational reconstruction of TRACER identifying the core operations performed on symbolic execution graphs (SEGs) in order to prove the two above properties. In this paper, TRACER is essentially used as an instance of such systems.

This paper proceeds as follows: After providing a short introduction into Isabelle/HOL and the notations we need, we present in Sec. 3 ATRACER by a small example. Sec. 4.1 is devoted to the introduction of the formal machinery of red-black graphs and their symbolic execution. We present in Sec. 4.2 the formalization of graph-transformations. In Sec. 4.3 we state formally the correctness and preservation properties and outline the proofs. The entire formalization and proof effort in Isabelle/HOL consists of about one hundred definitions or abbreviations and two hundred lemmas, representing about 8k lines of code. All proofs were written using the Isar proof language in a structured manner. No fancy theorem proving technologies were needed: the most expensive tactic used is `force`. The sources are available under `https://www.lri.fr/~wolff/atracer.zip`.

2 Background: Isabelle and Isabelle/HOL

Isabelle/HOL [10] is an interactive theorem proving environment for Church's higher-order logic (HOL), a classical logic based on a simply typed λ-calculus extended by parametric polymorphism. HOL provides the usual logical connectives like $\_ \wedge \_$, $\_ \rightarrow \_$, $\neg \_$ as well as the object-logical quantifiers $\forall x.\, P\, x$ and $\exists x.\, P\, x$; in contrast to first-order logic, quantifiers may range over arbitrary types, including total functions $f :: \alpha \Rightarrow \beta$. HOL is centered around extensional equality $\_ = \_ :: \alpha \Rightarrow \alpha \Rightarrow$ bool.

Isabelle/HOL comes with rich libraries for lists, typed sets, total and partial functions, etc. We introduce some library notations used throughout this paper: wrt. to sets, we use the usual `{x. P(x)}` for set comprehensions, `x` $\in$ `S` for inclusion, `A` $\cup$ `B, A` $\cap$ `B` for union and intersection, etc. Lists were built by the constructors `Nil` and `_ # _`. Of particular importance for this paper is the use of record notation; records are basically cartesian products where the components have a *tag-name*. As example, we declare a record by the specification construct:

```
record ('α, 'β) point = x :: "'α" y :: "'β"
```

This specification construct introduces a number of operations on record types (such as `('α, 'β)point`). For example, `P = (| x = 4, y = True |)` is a constructor of a record (of type `(int, bool)point`), where as `P' = P(x := 3)` is an update function of the record at the component `x`. The tag-names implicitly define selector functions on records; thus `y P'` is equivalent to `True`.

3 A Guided Run of ATRACER

TRACER avoids the full enumeration of symbolic paths by learning from infeasible paths and computing *interpolants* for program points. In this context, an interpolant is a logical formula associated with a program point that constraints a set of program states: whenever symbolic execution reaches that program point in a state satisfying the interpolant, it is ensured that the *final* program point can be reached from that point (without going through a given error statement). Once an interpolant has been synthesized for a program point, any symbolic execution path that reaches that point in a symbolic state satisfying the interpolant needs not to be extended further: it is ensured that it will reach the final program point. The new path is said to be *subsumed* by the previous one.

To avoid unrolling loops infinitely, when reaching a loop header TRACER checks if that program point can be subsumed by one of its prior occurrences on the path. Detection of subsumptions at loop headers is performed by computing abstraction between symbolic states, that is weakening constraints on the symbolic states for that point. Abstraction can be performed, for example, by removing or weakening (e.g. turning equalities into inequalities) constraints from the path predicates. Abstraction can be seen as a synthesis of a loop invariant. If the synthesized invariant is not strong enough, this can result in "false negatives" where paths that are infeasible in the original program are not ruled out

by the abstraction of the program states. Such abstractions have to be refined: Whenever a symbolic path leading to an error statement is produced from a point where an abstraction occurred, TRACER checks if that path exists without the abstraction. If this is the case, the error statement is truly reachable. Otherwise, information about the unfeasability is collected and an interpolant characterizing the unreachability of the error statement is attached to the node where the faulty abstraction was made. The analysis is restarted from that point, with its new interpolant serving as a safeguard for abstractions: abstractions at that point must now satisfy the interpolant. When it is not possible to find an abstraction between different occurrences of a loop header, the loop is unrolled one more time in the hope of later subsumptions. Absence of valid subsumptions leads TRACER to unroll loops infinitely. Otherwise, abstraction and subsumption result in a SEG that includes all the behaviors of the original program with respect to the reachability of error statements.

3.1 Presenting ATRACER by an Example

Our abstraction of the original TRACER is conceived as a set of graph transformations of an annotated CFG we call *red-black graph*. Its transitions are annotated with basic blocks of assignments, the skip-statement, or a guard that has to hold when executing this transition. A red-black graph represents the over-approximated set of feasible paths and is is made of two parts : the *black part* is the initial CFG and remains unchanged throughout the transformations; the *red part* consists initially of a single vertex and is extended by unfolding the initial CFG using our graph-transformation operators, i. e. by adding transitions that are symbolically executed, subsumption links, etc. The red and black parts are the *known* and *unknown* parts, respectively.

We illustrate ATRACER with the example drawn from Jaffar et al. [7]. The program in Fig. 1a implements a lock acquisition algorithm. The goal of the authors is to check that statement at line 8 is not reachable on any feasible path, ensuring that the lock is held when the execution exits the loop. The condition of statement `if (*)` at line 4 abstracts a call to an external condition (like a function or a system call) that returns true if the lock is held by another process. Hence at each traversal either branch of such a conditional can be taken independently of the current state of the execution. Doing so is equivalent to executing a `true`-guard. In Fig. 1b we give the CFG for the *lock* program.

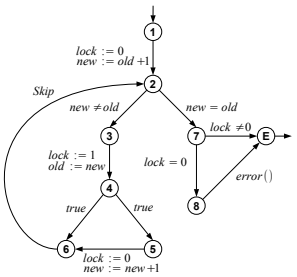
Since we are interested in illustrating the graph transformation operators, not in finding how to combine them in an actual system, in ATRACER we proceed as if we always guess correctly the next step. Thus, our sequence of elementary transformations differs from the one described by the authors in [7], whose order is controlled by several heuristics. However, we end up with the same final SEG as the original algorithm.

Notation: to distinguish the different *occurrences of program points* in the red part, we decorate the original location label (the line number) with a superscript. Superscripts start at 0 and increase with every further visit. Vertices labels

without superscript refer to the black part, those with a superscript to the red part. In Fig. 2a and latter, some red vertices are linked to their black counterpart by dotted edges. These links represent the continuation of the computation in the original program, i. e. parts that has not been symbolically executed yet.

```
1  lock = 0; new = old + 1;
2  while (new != old){
3      lock = 1; old = new;
4      if (*){
5          lock = 0; new = new + 1;}
6  }
7  if (lock == 0)
8      error ();
```

(a)



(b)

Fig. 1

In the red part (depicted with square vertices), vertices are implicitly annotated with *configurations*. Configurations are tuples: the first component, called the symbolic state, is a function associating symbolic variables with program variables; the second component is the path predicate, i. e. the conjunction of constraints over symbolic variables that are accumulated during symbolic execution of the current path up to that point. Path predicates are written under static single assignment form, introducing new symbolic variables for each assignment.

Initialization: We start the symbolic execution of the program in Fig. 1a with the configuration $(\{lock \mapsto lock_0, new \mapsto new_0, old \mapsto old_0\}, true)$ The red part consists of the single red vertex 1^0, corresponding to the entry program point in the black part and is linked to the latter (Fig. 2a).

Symbolic execution of assignments: from 1^0, we perform symbolic execution over the black transition leading from 1 to 2. This results in the addition of a red transition from 1^0 to 2^0. The symbolic state at 2^0 is obtained from the one at 1^0 by associating fresh symbolic variables with variables *lock* and *new* and adding two constraints to the path predicate. The configuration for 2^0 is: $(\{lock \mapsto lock_1, new \mapsto new_1, old \mapsto old_0\}, lock_1 = 0 \wedge new_1 = old_0 + 1)$.

Symbolic execution of guards: assuming the first symbolic path enters the loop, symbolic execution is performed from 2^0 over the transition from 2 to 3. The path predicate at 3^0 is the conjunct of path predicate at 2^0 with the constraint $new_1 \neq old_0$, obtained by substituting occurrences of program variables in the guard by the symbolic variables they are associated with in the symbolic state. Assuming we follow first the *then* branch of the inner conditional, i.e. statements at line 5, symbolic execution is performed

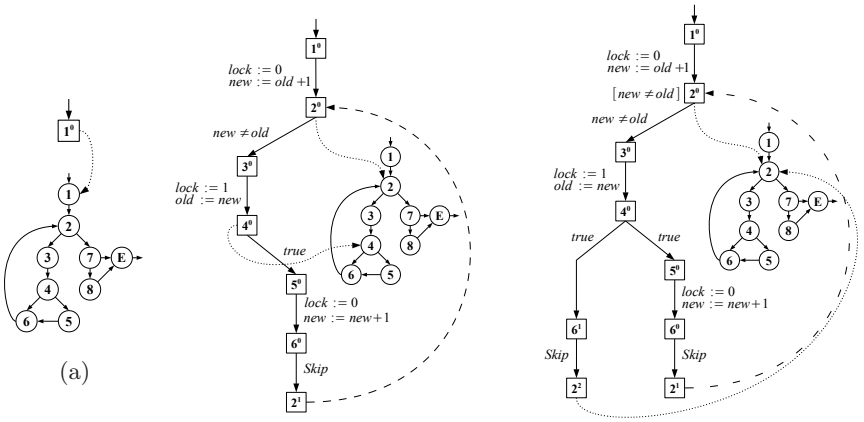


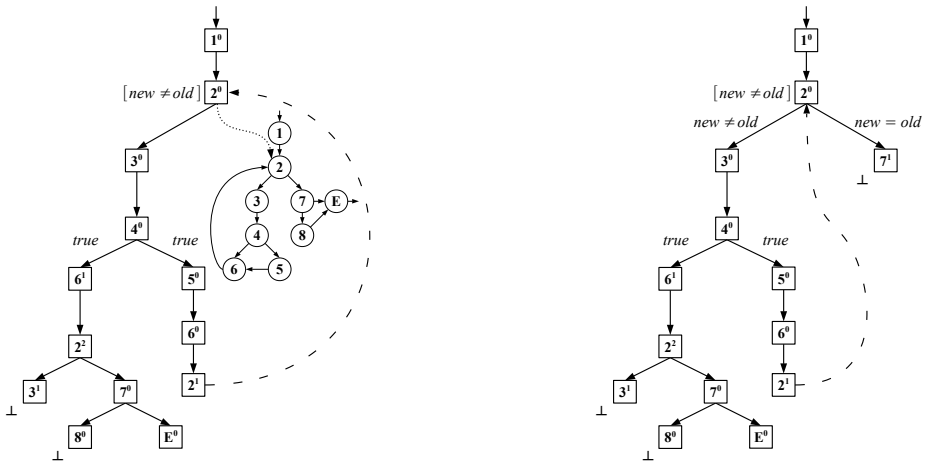
Fig. 2

until the statement at line 2 (the loop header) is reached for the second time, completing one loop iteration. The configuration for 2^1 is $(\{lock \mapsto lock_3, new \mapsto new_2, old \mapsto old_1\}, lock_1 = 0 \wedge new_1 = old_0 + 1 \wedge lock_2 = 1 \wedge old_1 = new_1 \wedge lock_3 = 0 \wedge new_2 = new_1 + 1)$.

Subsumption between loop headers: 2^0 and 2^1 are two occurrences on the current path of the same loop header. Given two occurrences v and v' of a same program point, v' can be subsumed by v if it is a particular case of v. When candidates for subsumption are discovered, in most cases the subsumption cannot occur directly. Subsuming a vertex often requires the configuration of the subsumer to be *abstracted*, that is relaxing some constraints of its path predicate, to force the subsumee to imply its subsumer. This is not needed here, since 2^1 and 2^0 can be shown to be logically equivalent. Vertex 2^1 is subsumed by 2^0 and the small dotted edge linking 2^1 to the black part is replaced by a subsumption link from 2^1 to 2^0 in Fig. 2b (depicted by the big dotted edge). After that subsumption, symbolic execution resumes at 4^0 and extends up to 2^2, a new target for a subsumption.

Limiting abstractions with interpolants: Before processing 2^2, the interpolant $new \neq old$ (written between brackets in Fig. 2c) is added at 2^0 to prevent the subsumption of 2^2 by 2^0. Labeling a vertex with an interpolant needs to show that the configuration entail the interpolant, which is the case here. The symbolic state at 2^2 is not a particular case of the one at 2^0 and abstraction is forbidden by the interpolant: subsumption cannot occur and the loop is unrolled, performing symbolic execution from 2^2 to 3^1.

Marking nodes as unsatisfiable: The path predicate at 3^1 is unsatisfiable, as it requires *new* and *old* to be both equal and different. We *mark* (with a



(a) (b)

Fig. 3

⊥ symbol, in Fig. 3a) nodes known as unsatisfiable. In practice, this would result from a call to a constraint solver: in ATRACER only configurations proved as unsatisfiable can be marked. As we do not expect the user to call a solver at every node, for performances reasons, we let it be an explicit action. We do not disallow to pursue from a marked configuration, but symbolic execution will carry the mark to its successors.

Symbolic execution resumes at node 2^2, and follows the exit branch of the loop. The exit point and the error location are reached, respectively at E^0 and 8^0, and the latter is marked since its path predicate is unsatisfiable. Symbolic execution now resumes at node 2^0, the last pending point, and exits the loop, reaching 7^1. As the configuration at 2^0 does not satisfy the exit condition, 7^1 is marked as unsatisfiable (Fig. 3b).

In Fig. 3b, every leaf of the red part is either marked as unsatisfiable, subsumed or an occurrence of the final black location. Every feasible path of the black part is contained in the red part! The error statement at line 8 is no longer reachable from a feasible symbolic path (from the red entry) either in the red part or in the black part (red vertices linked to the black part are all marked as unsatisfiable) which can therefore be pruned.

4 The Formalization of ATRACER

4.1 The Theory of Red-Black Graphs

In this section, we introduce the main concepts needed in order to formalize red-black graphs and to state and prove the main theorems we are interested

in. We first introduce the definitions we use to model graphs and CFGs, then present main definitions and facts about symbolic execution.

Basic Definitions about Graphs The notion of graph and associated concepts like sub-paths are central to our formalization because we need an abstraction of sharing in the abstract syntax as well as arbitrary cycles in the CFG. Moreover, we need to consider paths going through subsumption links, a notion that is specific to our approach. We start with a conventional definition of a graph over some type of vertices `'a` as a set of arcs linking the nodes:

```
record 'a arc = src :: "'a"          record 'a rgraph = root :: "'a"
                tgt :: "'a"                             arcs :: "'a arc set"
```

Our notion of graph assumes that they have one single `root` (which comes in handy when modeling CFG's). So far the definitions do neither imply that the graph is connected and that `root` has any connection to the arcs. These kind of side-conditions are captured by additional predicates, and sometimes managed by the Isabelle concept of a *locale*, covered by Ballarin in [1].

On this basis, a rich theory of auxiliary concepts must be developed. For instance, we need the concept of a "consistent arc sequence":

```
fun cas :: "'a ⇒ 'a arc list ⇒ 'a ⇒ bool"
where
  "cas v1 [] v2 = (v1 = v2)"
| "cas v1 (a#seq) v2 = (src a = v1 ∧ cas (tgt a) seq v2)"
```

which paves the way to the concept of a `subpath`:

```
"subpath g v1 seq v2 ≡ cas v1 seq v2 ∧ v1 ∈ verts g ∧ set seq ⊆ arcs g"
```

and `path` (a sub-path from the root). Both concepts were borrowed from Nochinski's Graph Library for Isabelle [11]. Here, we define as `vert` the nodes which are either root, source or target of an arc, and add the usual notions of in- or outgoing arcs. We add abstract operations on graphs (like adding arcs) and establish a number of properties wrt. vertices, paths, inarcs, and outarcs.

We then define graphs equipped with a subsumption relation. In the following, subsumptions only involve vertices of the red part that represent different occurrences of a same vertex of the black part. We represent subsumption relations by sets of pairs of indexed vertices:

```
type_synonym 'a sub_t = "(('a × nat) × ('a × nat))"
type_synonym 'a sub_rel_t = "'a sub_t set"
```

Again, paths and sub-paths in a graph equipped with such a relation are defined using the notion of consistency of an arc sequence. An arc sequence is consistent in presence of a subsumption relation if it is made of a number of consecutive consistent (without the subsumption relation) sequences whose extremities are linked throughout elements of the subsumption relation.

If we add an arc labeling function to graphs, we speak of *labeled transition systems* (lts). Their type is defined by the record-extension:

```
record ('a,'b,'c) lts = "'a rgraph" + labf :: "'a arc ⇒ ('b,'c) label"
```

where `'c` is the type of values taken by program variables. It enriches the `'a rgraph` with a labeling function; these labels turning an arbitrary graph into a CFG have a richer structure that we describe in the following.

Main Definitions and Facts about Symbolic Execution First, we define corresponding to program variables `'a` the symbolic variables with their superscripts (cf. Sec. 3.1) by a type synonym (pairing that program variable with an integer) and define the concept of a store for bookkeeping the current association of a program variable with its symbolic variable represented by its superscript.

```
type_synonym 'b symvar = "'b × nat"
type_synonym 'b store  = "'b ⇒ nat"
```

States are used to give values to variables. Arithmetic and boolean expressions are modeled in shallow embedding style, by total functions from variables to their domain and to boolean values, respectively.

```
type_synonym ('b,'c) state = "'b ⇒ 'c"
type_synonym ('b,'c) aexp  = "('b,'c) state ⇒ 'c"
type_synonym ('b,'c) bexp  = "('b,'c) state ⇒ bool"
```

This way of modeling expressions has the advantage that there is no need to formalize the different operators on expressions, which would have been necessary using a syntactic approach. Moreover, shallow embedding allows the use of the existing Isabelle notations and theorems about functions.

On the other hand, it makes it a bit harder to describe the set of variables of such expressions, which is needed when reasoning about the freshness of some symbolic variable for a configuration. We define the set of variables of an arithmetic (resp. boolean) expression as the set of variables that can actually have an influence over the value of this expression.

```
definition vars :: "('b,'c) aexp ⇒ 'b set" where
  "vars e = {v. ∃σ val. e (σ(v := val)) ≠ e σ}"
```

Since configurations and subsumption between configurations have been introduced in Section 3.1, we skip their formal definitions here and go directly to symbolic execution. We note $c \sqsubseteq c'$ the fact that configuration c is subsumed by configuration c'.

Symbolic execution is defined as an inductive predicate `se` that takes two configurations c and c' and a label l and evaluates to *true* if c' is a *result* of the symbolic execution of l from c. Results are defined up to the way fresh indexes are chosen in the case of Assign labels. We prove that fresh indexes exist when needed, assuming expressions in labels and configurations from which symbolic execution is performed have finite sets of variables.

Labels are either *Skip*, *Assume* ϕ, where ϕ is a boolean expression, or *Assign* v e where v is a program variable and e an arithmetic expression.

```
datatype ('b,'c) label =
  Skip | Assume "('b,'c) bexp" | Assign 'b "('b,'c) aexp"

inductive se :: "('b,'c) conf ⇒ ('b,'c) label ⇒ ('b,'c) conf ⇒ bool"
where
  "se c Skip c"
| "se c (Assume e) (| store c, pred = pred c ∪ {adapt_bexp e (store c)} |)"
| "fresh_symvar (v,i) c ⟹
   se c (Assign v e)
   (| store = (store c)(v := i),
     pred  = pred c ∪ {(λ σ. σ (v,i) = (adapt_aexp e (store c)) σ)} |)"
```

Here, `adapt_aexp e s` (resp. `adapt_bexp`) represent the expression obtained from the arithmetic (resp. boolean) expression e by substituting every occurrence of program variables by their symbolic counterpart given by s. It would have been possible to define `se` as a function, but the assumption about freshness in the case of an assignment would require a special treatment. This could be done in a number of ways. For example, `se` could be a partial function defined only in those cases where the new symbolic variable is indeed fresh.[1] In the end, we found that using a predicate was the simplest way to model `se`, and also yields simpler proofs in the rest of the formalization.

We extend symbolic execution to sequences of labels, and model it by an inductive predicate `se_star` that takes two configurations and a sequence of labels, and evaluates to *true* if the second configuration is a possible result of symbolic execution of the given sequence from the first configuration.

To prove the key properties of our approach, one first proves that symbolic execution is monotonic with respect to the previous definition of subsumption. We only state the theorem for `se`, a similar one holds for `se_star`.

```
theorem se_mono_for_sub :
assumes "se c1 l c1'"
assumes "se c2 l c2'"
assumes "c2 ⊑ c1"
shows   "c2' ⊑ c1'"
```

The proof is obtained by case distinction on l, expressing the states of $c1'$ and $c2'$ as functions of the states of $c1$ and $c2$, respectively. In the case of sequence of labels ls, the proof is obtained by induction on ls, using `se_mono_for_sub`.

4.2 Graph-Transformations on Red-Black Graphs

We are ready to give the structure of `('a,'b,'c) pre_RedBlack` before defining what means to be a `('a,'b,'c) RedBlack` graph.

```
record ('a,'b,'c) pre_RedBlack =
  red           :: "('a × nat) rgraph"
```

[1] Given an arbitrary configuration, there is no guarantee that there exists a fresh symbolic variable for a given program variable, since expressions are defined as total functions.

```
  black          :: "('a,'b,'c) lts"
  subs           :: "(('a × nat) × ('a ×  nat)) set"
  init_conf      :: "'b conf"
  confs          :: "('a × nat) ⇒ ('b,'c) conf"
  marked         :: "('a × nat) ⇒ bool"
  strengthenings :: "('a × nat) ⇒ ('b,'c) bexp"
```

The fields `red` and `black` represent the red and black parts, respectively. `init_conf` is the configuration initially chosen to start the analysis. `subs` is the subsumption relation which contains the subsumption links between the vertices of `red`. Finally, `confs`, `marked` and `strengthenings` are functions associating to the vertices of `red` their current configuration, the fact that they are marked as unsatisfiable or not, and their current interpolant, respectively.

We now specify what we call the GT-calculus, i.e. the five graph transformations and the set of reachable red-black graphs from an initial configuration containing just a black part and an empty well-formed red part.[2] The construction proceeds per inductive definition as follows:

```
inductive RedBlack :: "('a,'b,'c) pre_RedBlack ⇒ bool" where
  init :
    "fst (root (red rb)) = init (black rb)          ⟹
     arcs (red rb) = {}                             ⟹
     subs rb = {}                                   ⟹
     (confs rb) (root (red rb)) = init_conf rb      ⟹
     marked rb = (λ rv. False)                      ⟹
     strengthenings rb = (λ rv. (λ σ. True))        ⟹ RedBlack rb"
| se_step :
    "RedBlack rb ⟹ se_extends rb ra c' rb'          ⟹ RedBlack rb'"
| mark_step :
    "RedBlack rb ⟹ mark_extends rb rv rb'           ⟹ RedBlack rb'"
| subsum_step :
    "RedBlack rb ⟹ subsum_extends rb sub rb'        ⟹ RedBlack rb'"
| abstract_step :
    "RedBlack rb ⟹ abstract_extends rb rv e rb'     ⟹ RedBlack rb'"
| strengthen_step :
    "RedBlack rb ⟹ strengthen_extends rb rv e rb'  ⟹RedBlack rb'"
```

where operations `se_extends`, `mark_extends`, `subsum_extends`, `abstract_extends` and `strengthen_extends` are abbreviations (macros) for a number of constraints necessary to describe, one by one, the graph transformations informally introduced in Sec. 3.1. We pick the graph transformation `se_extends` as example:

```
abbreviation se_extends ::
  "('a,'b,'c) pre_RedBlack ⇒('a × nat) arc ⇒ ('b,'c) conf ⇒
   ('a,'b,'c) pre_RedBlack ⇒bool"
where
  "se_extends rb ra c' rb' ≡
     ui_arc ra ∈ arcs (black rb)                                  (* 1 *)
```

[2] This is ensured by a number of constraints on the free variable `rb` forcing the root of the red part to be the initial location of the black part, etc.

```
∧ ArcExt.extends (red rb) ra (red rb')                          (* 2 *)
∧ src ra ∉ subsumees (subs rb)                                  (* 3 *)
∧ se (confs rb (src ra)) (labf (black rb)(ui_arc ra)) c'        (* 4 *)
∧ rb' = (| red       = red rb',
           black     = black rb,
           subs      = subs rb,
           init_conf = init_conf rb,
           confs     = (confs rb) (tgt ra := c'),
           marked    = (marked rb)(tgt ra := marked rb (src ra)),
           strengthenings = strengthenings rb |)                (* 5 *)"
```

The constraints describe formally the following side-conditions (we follow the labels in comments above):

1. `ui_arc ra`, the (unindexed) black counterpart of red arc ra must exist in the black graph,
2. `ArcExt.extends` is an abbreviation that states that the source of ra must be an existing vertex of the red graph, but not its target, and that the new red graph is obtained by adding ra to the arcs of the old one,
3. the source of ra is is not already subsumed,[3]
4. c' is the new configuration obtained by symbolic execution of ra
5. the new red-black graph rb' is constructed from the old one by the following updates:
   - ra is added to the red graph
   - the new configuration is added at the target of ra
   - the satisfiability-flag of the target of ra is set to the one of its source. Recall that we want registration of unsatisfiablity to be an explicit action.

The amount of detail that must be added when reasoning precisely over the correctness issues of these type of graph-based static analysis algorithms is quite substantial and makes it a valuable target for a machine-checked analysis.

From now on, we call red-black graphs the set of `pre_RedBlack` reachable by the predicate `RedBlack`.

4.3 Main Theorems of ATRACER

Relation between Red Vertices In the case of a classical symbolic execution tree, one would prove that, given one sub-path in the tree, the symbolic state at its end has been obtained by symbolic execution of its trace from the symbolic state at its beginning. This property is too strong for red graphs obtained by the GT-calculus. We must handle sub-paths that go through subsumption links and configurations along these sub-paths may have been abstracted, both inducing a loss of information about program states.[4] In ATRACER, the configuration at

[3] The conjunction of 2 and 3 is equivalent to say that the source of ra is a pending point in the analysis.

[4] Note that in the second assumption of `gt_calc_se_rel`, unlike in Sec. 4.1, `subpath` has a fifth parameter: the subsumption relation of rb.

the end of a sub-path merely subsumes the one obtained by classical symbolic execution. This is expressed by the following theorem.

```
theorem gt_calc_se_rel :
assumes "RedBlack rb"
assumes "subpath (red rb) r1 s r2 (subs rb)"
assumes "se_star (confs rb r1) (trace (ui_as s) (labelling (black rb))) c"
shows   "c ⊑ (confs rb r2)"
```

The proof is obtained by rule induction on `RedBlack`, i.e. the five transformation operators maintain the property. All cases are quite straightforward, except for adding a subsumption link. The details of its proof are quite tedious and numerous, so we skip them here and just give the main idea. The problem is that we do not know how many times the considered sub-path goes through the new subsumption, if it does. But as we consider finite sub-paths only, this number is finite: the proof is obtained by a backward induction on s, using the fact that subsumption between configurations is a partial ordering for which symbolic execution is monotonic.

Red-black Sub-paths and Paths Before stating our two main theorems, we formalize the notion of sub-path of a red-black graph and its set of paths. Given a vertex rv of the red graph, we first define the set of red-black sub-paths starting from rv as the union of the two following sets:

- the sets of black sub-paths entirely represented in the red graph by sub-paths starting at rv and ending in a non-marked red vertex,
- the sets of black sub-paths that have a prefix represented in the red graph leading to a non-marked red vertex rv', which is not subsumed and from which there exist black arcs that have not been symbolically executed yet. Moreover, the remaining black suffix must have no (non-empty) prefixes represented in the red graph (starting at rv').

As in Sec. 4.1, we define the set of red-black paths as the set of red-black sub-paths starting at the root of the red graph. This complex definition is needed to ensure that what we call the set of red-black paths is not simply the set of paths of the black graph.

Correctness of the GT-Calculus Our first main theorem states that red-black paths all come from paths of the black part. More precisely, every red-black sub-path starting at some red vertex rv is also a sub-path starting at the black vertex represented by rv in the black graph. Thus, our approach is correct in the sense that it does not introduce new paths in the red-black graph and preserve program behavior.

```
theorem gt_calc_correct :
assumes "RedBlack rb"
shows   "RedBlack_subpaths_from rb rv
         ⊆Graph.subpaths_from (black rb) (fst rv)"
```

The theorem relies on the fact that arcs added to the red part are simply indexed versions of black arcs, and that subsumption links only involve different occurrences of the same black vertices.

Preservation of Feasible Paths Finally, we prove that the original set of feasible paths is contained in the red-black graph. Our main theorem is the following: given a red vertex *rv*, every feasible black sub-path *bs* starting at the black vertex represented by *rv* from the configuration associated to *rv* is a red-black sub-path starting at *rv*.

```
theorem gt_calc_preserves :
assumes "RedBlack rb"
assumes "rv ∈ red_vertices rb"
assumes "bs ∈ feasible_subpaths_from (black rb) (confs rb rv) (fst rv)"
shows   "bs ∈ RedBlack_subpaths_from rb rv"
```

As for correctness, the proof (which is 2.3k loc, and can be found in file `RB.thy`) is obtained by rule induction on `RedBlack`. For each operator, we use the induction hypothesis to get that *bs* is also a red-black sub-path of the old red-graph, before proving that it is not ruled out by the current transformation. The initial case is trivial, as well as those of abstracting a configuration and adding an interpolant, since the former only makes the set of red-black sub-paths larger and the latter does not modify the graph structure but only prevents future abstractions. The case of adding a red arc is simple but tedious as one needs to treat the numerous sub-cases. Marking a red vertex as unsatisfiable is proved using the fact that the vertices that *bs* goes through cannot be marked, otherwise *bs* would not be feasible. The case of adding a subsumption link is the difficult one, for the same reasons as previously. Again, the proof is obtained by a backward induction on the considered sub-path before proceeding by case distinction.

We rephrase our main theorem in more readable way:

```
theorem gt_calc_preserves2 :
assumes "RedBlack rb"
shows   "feasible_paths (black rb) (init_conf rb) ⊆ RedBlack_paths rb"
```

It is proved using the fact that the initial configuration of a red-black graph is subsumed by the one associated with the root of its red part, hence the set of feasible paths starting from the former is a subset of the set of the latter.

4.4 Summary

The formalization of ATRACER presented a number of challenges. We first attempted to formalize the whole TRACER's algorithm, heuristics aspects included. At this time, the SEG was modeled as a tree, whose nodes and leaves could have different types: *simple*, *unsatisfiable*, *subsumed*, *subsumer*, and were decorated with much information, like configurations, the identity of the subsumer, etc. We then faced major difficulties. First, this structure is not suitable

to describe inductively how its set of paths evolves after adding a new node, a subsumption, etc. Our current modeling of graphs equipped with subsumption relations makes this task far more easy. Second, it is very difficult to model in details the heuristics aspects, like graph traversals, how subsumptions are detected, or how abstractions are refined in practice, for example. We finally chose to "break" TRACER's algorithm into pieces in order to identify and formalize the core operations it performs on SEG, and to give up the heuristics aspects, since they have no influence on the preservation of feasible paths. Finally, due to the nature of the problem - symbolic execution in presence of unbounded loops, TRACER-like algorithms might not terminate. In practice, this is handled using some kind of time-out condition. When such conditions are triggered, the only way to preserve all feasible paths is to "plug" the actual SEGs into the original graph. In ATRACER, this is represented by the black part and the complex definition of red-black paths. This is also what motivates identifying the core operations, since the problem of preservation is reduced to showing that each operator never rules out feasible paths.

5 Conclusion

Related Work. Our work is inspired by Tracer [8] and the more wider class of Cegar-like systems [2, 3, 5, 6, 9] based on predicate abstraction. However, we did not attempt any code-verification of these systems and rather opted for their rational reconstruction allowing for a clean separation of heuristics and fundamental parts. Moreover, our treatment of `Assume` and `Assign`-labels is based on shallow encodings for reasons of flexibility and model simplification, which these systems lack. There is a substantial amount of formal developments of graph-theories in HOL, most closest is perhaps by Lars Noschinski [11] in the Isabelle AFP. However, we do not use any deep graph-theory in our work; graphs were just used as a kind of abstract syntax allowing sharing and arbitrary cycles in the control-flow. And there are a large number of works representing programming languages, be it by shallow or deep embedding; on the Isabelle system alone, there is most notably the works on Ninja, NanoJava, IMP, etc. However, these works represent the underlying abstract syntax by a free datatype and are not concerned with the introduction of sharing in the program presentation; to our knowledge, our work is the first approach that describes optimizations by a series of graph transformations on CFG's in HOL.

Summary. We formally proved the correctness of a set of graph transformations used by systems that compute approximaztions of sets of (feasible) paths by building symbolic evaluation graphs. Formalizing all the details needed for a machine-checked proof was a substantial work. To our knowledge, such formalization was not done before.

The ATRACER model separates the fundamental aspects and the heuristic parts of the algorithm. Additional graph transformations for restricting abstrac-

tions or for computing interpolants or invariants can be added to the current framework, reusing the existing machinery for graphs, paths, configurations, etc.

Future Work. Currently, we are implementing a prototype "by hand" that must not only preserve feasible paths but heuristically generate abstractions and subsumptions. It would be possible to generate the core operations on red-black graphs by the Isabelle code-generator, by introducing un-interpreted function symbols for concrete heuristic functions that were mapped to implementations written by hand. This represents a substantial, albeit rewarding effort that has not yet been undertaken.

Acknowledgement We thank Marie-Claude Gaudel for her support while doing this work and for her remarks on this article.

References

[1] Clemens Ballarin. Locales: A Module System for Mathematical Theories. *J. Autom. Reasoning*, 52(2):123–153, 2014.

[2] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The Software Model Checker Blast. *STTT*, 9(5-6):505–525, 2007.

[3] Edmund M. Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: SAT-Based Predicate Abstraction for ANSI-C. In *Proceedings of TACAS '05*, pages 570–574, 2005.

[4] Alain Denise, Marie-Claude Gaudel, Sandrine-Dominique Gouraud, Richard Lassaigne, Johan Oudinet, and Sylvain Peyronnet. Coverage-biased Random Exploration of large Models and Application to Testing. *International Journal on Software Tools for Technology Transfer*, 14(1):73–93, 2011. ISSN 1433-2787.

[5] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing Software Verifiers from Proof Rules. In *Proceedings of PLDI '12*, pages 405–416, 2012.

[6] Franjo Ivancic, Zijiang Yang, Malay K. Ganai, Aarti Gupta, Ilya Shlyakhter, and Pranav Ashar. F-Soft: Software Verification Platform. In *Proceedings of CAV '05*, pages 301–306, 2005.

[7] Joxan Jaffar, Jorge A. Navas, and Andrew E. Santosa. Unbounded Symbolic Execution for Program Verification. In *Proceedings of RV '11*.

[8] Joxan Jaffar, Vijayaraghavan Murali, Jorge A. Navas, and Andrew E. Santosa. TRACER: A Symbolic Execution Tool for Verification. In *Proceedings of CAV '12*, pages 758–766, 2012.

[9] Kenneth L. McMillan. *Proceedings of CAV '06*, chapter Lazy Abstraction with Interpolants, pages 123–136. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[10] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*, volume 2283. 2002.

[11] Lars Noschinski. A Graph Library for Isabelle. *Mathematics in Computer Science*, 9(1):23–39, 2015. ISSN 1661-8289. doi: 10.1007/s11786-014-0183-z. URL `http://dx.doi.org/10.1007/s11786-014-0183-z`.