Infeasible Paths Elimination by Symbolic Execution Techniques:

Proof of Correctness and Preservation of Paths

Romain Aissat and Frédéric Voisin and Burkhart Wolff

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

March 17, 2025

Abstract

TRACER [1] 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, 4, 5, 6]. 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.

The accompanying paper (published in ITP 2016) can be found at https://www.lri.fr/~wolff/papers/conf/2016-itp-InfPathsNSE.pdf, also appeared in[7].

Keywords: TRACER, CEGAR, Symbolic Executions, Feasible Paths, Control-Flow Graphs, Graph Transformation

Contents

1 Introduction

In this document, we formalize a method for pruning infeasible paths from control-flow graphs. The method formalized here is a graph-transformation approach based on *symbolic execution*. Since we consider programs with unbounded loops, symbolic execution is augmented by the detection of *subsumptions* in order to stop unrolling loops eventually. The method follows the *abstract-check-refine* paradigm. Abstractions are allowed in order to force subsumptions. But, since abstraction consists of loosing part of information at a given point, abstractions might introduce infeasible paths into the result. A counterexample guided refinement is used to rule out such abstractions.

This method takes a CFG G and a user given precondition and builds a new CFG G' that still over-approximates the set of feasible paths of G but contains less infeasible paths. It proceeds basically as follows (see [8] for more details). First, it starts by building a classical symbolic execution tree (SET) of the program under analysis. As soon as a cyclic path is detected, the algorithm searches for a subsumption of the point at the end of the cycle by one of its ancestors. When doing this, the algorithm is allowed to abstract the ancestor in order to force the subsumption. When a subsumption is established, the current symbolic execution halts along that path and a subsumption link is added to the SET, turning it into a symbolic execution graph (SEG). When an occurrence of a final location of the original CFG is reached, we check if abstractions that might have been performed along the current path did not introduce certain infeasible paths in the new representation. If no refinement is needed, symbolic execution resumes at the next pending point. Otherwise, the analysis restarts at the point where the "faulty" abstraction occurred, but now this point is strengthened with a *safeguard condition*: future abstractions occurring at this point will have to entail the safeguard condition, preventing the faulty abstraction to occur again. These safeguard conditions could be user-provided but are typically the result of a *weakest precondition calculus*. When the analysis is over, the SEG is turned into a new CFG.

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 [9]. If the probability of drawing a feasible path from such a transformed CFG was high, this would lead to an efficient statistical structural white-box testing method. With testing in mind, a crucial property that our approach must have, besides being correct, is to preserve the set of feasible paths of the original CFG. Our goal with this formalization is to establish correctness of the approach and the fact that it preserves the feasible paths of the original CFG, that is:

1. for every path in the new CFG, there exists a path with the same trace in the original CFG,

2. for every feasible path of the original CFG, there exists a path with the same trace in the new CFG.

We consider that our method is made of five graph-transformation operators and a set of heuristics. These five operators consist in:

1. adding an arc to the SEG as the result of a symbolic execution step in the original CFG,

2. adding a subsumption link to the SEG,

3. abstracting a node of the SEG,

4. marking a node as unsatisfiable,

5. labelling a node with a safeguard condition.

Heuristics control, for example, the order in which these operators are applied, which of the possible abstractions is selected, etc. These heuristics cannot interfer with the correctness of the approach or the preservation of feasible paths since they simply combine the five kernel transformations. In the following, we model the different data structures that our method performs on and formalize our five operators but completely skip the heuristics aspects of the approach. Thus, our results extend to a large family of algorithms that add specific heuristics in their goal to over-approximate the set of feasible paths of a CFG.

Due to the nature of the problem, symbolic execution in presence of unbounded loops, such algorithms might not terminate. In practice, this is handled using some kind of timeout condition. When such condition triggers, the SEG is only a partial unfolding of the original CFG. Thus, the resulting CFG cannot contains all feasible paths of the original one. In this situation, the only way to preserve the set of feasible paths is to "connect" the SEG to the original CFG. The SEG is the currently known over-approximating set of prefixes of feasible paths and the original CFG represents the unknown part of the set of feasible paths.

In the following, we use an adequate data structure that we call a *red-black graph*. Its *black part* is the original CFG: it represents the unknown part of the set of feasible paths and is never modified during the analysis. The *red part* represents the SEG: its vertices are occurrences of the vertices of the black part. Then, we define the five operators that will modify the red part as described previously. We only consider red-black graphs built using these five operators, starting from a red-black graph whose red part is empty. Paths of such structures are called *red-black paths*. Such paths start in the red part and might end in the black part: they are made of a red feasible prefix and a black prefix on which nothing is known about feasibility. Finally, we prove that, given any red-black graph built using our five operators and modulo a renaming of vertices, the set of red-black paths is a subset of the set of black paths and that the set of feasible black paths is a subset of the set of red-black paths.

In the following, we proceed as follows (see Figure 1 for the detailed hierarchy). First, we formalize all the aspects related to symbolic execution, subsumption and abstraction (`Aexp.thy`, `Bexp.thy`, `Store.thy`, `Conf.thy`, `Labels.thy`, `SymExec.thy`). Then, we formalize graphs and their paths (`Graph.thy`). Using extensible records allows us to model Labeled Transition Systems from graphs (`Lts.thy`). Since we are interested in paths going through subsumption links, we also define these notions for graphs equipped with subsumption relations (`SubRel.thy`) and prove a number of theorems describing how the set of paths of such graphs evolve when an arc (`ArcExt.thy`) or a subsumption link (`SubExt.thy`) is added. Finally, we formalize the notion of red-black graphs and prove the two properties we are mainly interested in (`RB.thy`).

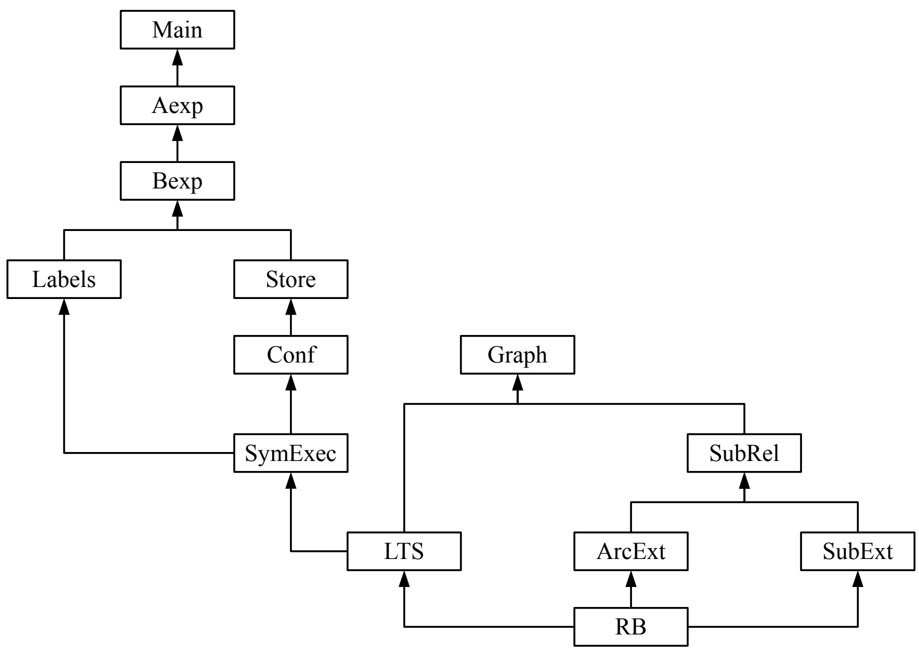

Figure 1: The hierarchy of theories.

theory *Graph*
imports *Main*
begin

2 Rooted Graphs

In this section, we model rooted graphs and their sub-paths and paths. We give a number of lemmas that will help proofs in the following theories, but that are very specific to our approach.

First, we will need the following simple lemma, which is not graph related, but that will prove useful when we will want to exhibit the last element of a non-empty sequence.

lemma *neq-Nil-conv2* :
  $xs \neq [] = (\exists\ x\ xs'.\ xs = xs'\ @\ [x])$
$\langle proof \rangle$

2.1 Basic Definitions and Properties

2.1.1 Edges

We model edges by a record *'v edge* which is parameterized by the type *'v* of vertices. This allows us to represent the red part of red-black graphs as well as the black part (i.e. LTS) using extensible records (more on this later). Edges have two components, *src* and *tgt*, which respectively give their source and target.

record *'v edge* =
  src :: *'v*
  tgt :: *'v*

2.1.2 Rooted graphs

We model rooted graphs by the record *'v rgraph*. It consists of two components: its root and its set of edges.

record *'v rgraph* =
  root :: *'v*
  edges :: *'v edge set*

2.1.3 Vertices

The set of vertices of a rooted graph is made of its root and the endpoints of its edges. Isabelle/HOL provides *extensible records*, i.e. it is possible to define records using existing records by adding components. The following definition suppose that g is of type $('v,'x)$ *rgraph-scheme*, i.e. an object that has at least all the components of a $'v$ *rgraph*. The second type parameter $'x$ stands for the hypothetical type parameters that such an object could have in addition of the type of vertices $'v$. Using $('v,'x)$ *rgraph-scheme* instead of $'v$ *rgraph* allows to reuse the following definition(s) for all type of objects that have at least the components of a rooted graph. For example, we will reuse the following definition to characterize the set of locations of a LTS (see `LTS.thy`).

definition *vertices* ::
  $('v,'x)$ *rgraph-scheme* $\Rightarrow$ $'v$ *set*
where
  $vertices\ g = \{root\ g\} \cup src\ `edges\ g \cup tgt\ `\ edges\ g$

2.1.4 Basic properties of rooted graphs

In the following, we will be only interested in loop free rooted graphs and in what we call *well formed rooted graphs*. A well formed rooted graph is rooted graph that has an empty set of edges or, if this is not the case, has at least one edge whose source is its root.

abbreviation *loop-free* ::
  $('v,'x)$ *rgraph-scheme* $\Rightarrow$ *bool*
where
$loop\text{-}free\ g \equiv \forall\ e \in edges\ g.\ src\ e \neq tgt\ e$

abbreviation *wf-rgraph* ::
  $('v,'x)$ *rgraph-scheme* $\Rightarrow$ *bool*
where
$wf\text{-}rgraph\ g \equiv root\ g \in src\ `\ edges\ g = (edges\ g \neq \{\})$

Even if we are only interested in this kind of rooted graphs, we will not assume the graphs are loop free or well formed when this is not needed.

2.1.5 Out-going edges

This abbreviation will prove handy in the following.

abbreviation *out-edges* ::
  $('v,'x)$ *rgraph-scheme* $\Rightarrow$ $'v \Rightarrow 'v$ *edge set*
where
  $out\text{-}edges\ g\ v \equiv \{e \in edges\ g.\ src\ e = v\}$

2.2 Consistent Edge Sequences, Sub-paths and Paths

2.2.1 Consistency of a sequence of edges

A sequence of edges *es* is consistent from vertex *v1* to another vertex *v2* if *v1* = *v2* if it is empty, or, if it is not empty:

- *v1* is the source of its first element, and
- *v2* is the target of its last element, and
- the target of each of its elements is the source of its follower.

fun *ces* ::
  'v ⇒ 'v edge list ⇒ 'v ⇒ bool
where
  ces v1 [] *v2* = (*v1* = *v2*)
| *ces v1* (*e*#*es*) *v2* = (*src e* = *v1* ∧ *ces* (*tgt e*) *es v2*)

2.2.2 Sub-paths and paths

Let *g* be a rooted graph, *es* a sequence of edges and *v1* and *v2* two vertices. *es* is a sub-path in *g* from *v1* to *v2* if:

- it is consistent from *v1* to *v2*,
- *v1* is a vertex of *g*,
- all of its elements are edges of *g*.

The second constraint is needed in the case of the empty sequence: without it, the empty sequence would be a sub-path of *g* even when *v1* is not one of its vertices.

definition *subpath* ::
  (*'v*,*'x*) *rgraph-scheme ⇒ 'v ⇒ 'v edge list ⇒ 'v ⇒ bool*
where
  subpath g v1 es v2 ≡ *ces v1 es v2* ∧ *v1* ∈ *vertices g* ∧ *set es* ⊆ *edges g*

Let *es* be a sub-path of *g* leading from *v1* to *v2*. *v1* and *v2* are both vertices of *g*.

lemma *fst-of-sp-is-vert* :
  assumes *subpath g v1 es v2*
  shows *v1* ∈ *vertices g*
⟨*proof*⟩

lemma *lst-of-sp-is-vert* :
  assumes *subpath g v1 es v2*
  shows $v2 \in vertices\ g$
$\langle proof \rangle$

The empty sequence of edges is a sub-path from *v1* to *v2* if and only if they are equal and belong to the graph.

The empty sequence is a sub-path from the root of any rooted graph.

lemma
  subpath g (root g) [] (root g)
$\langle proof \rangle$

In the following, we will not always be interested in the final vertex of a sub-path. We will use the abbreviation *subpath-from* whenever this final vertex has no importance, and *subpath* otherwise.

abbreviation *subpath-from* ::
  $('v,'x)$ *rgraph-scheme* $\Rightarrow$ $'v \Rightarrow$ $'v$ *edge list* $\Rightarrow$ *bool*
where
  subpath-from g v es $\equiv \exists\ v'.$ *subpath g v es* v'

abbreviation *subpaths-from* ::
  $('v,'x)$ *rgraph-scheme* $\Rightarrow$ $'v \Rightarrow$ $'v$ *edge list set*
where
  subpaths-from g v $\equiv$ {*es. subpath-from g v es*}

A path is a sub-path starting at the root of the graph.

abbreviation *path* ::
  $('v,'x)$ *rgraph-scheme* $\Rightarrow$ $'v$ *edge list* $\Rightarrow$ $'v \Rightarrow$ *bool*
where
  path g es v $\equiv$ *subpath g (root g) es v*

abbreviation *paths* ::
  $('a,'b)$ *rgraph-scheme* $\Rightarrow$ $'a$ *edge list set*
where
  paths g $\equiv$ {*es.* $\exists$ *v. path g es v*}

The empty sequence is a path of any rooted graph.

lemma
  $[] \in paths\ g$
$\langle proof \rangle$

Some useful simplification lemmas for *subpath*.

lemma *sp-one* :
  subpath g v1 [*e*] *v2* = (*src e* = *v1* ∧ *e* ∈ *edges g* ∧ *tgt e* = *v2*)
⟨*proof*⟩

lemma *sp-Cons* :
  subpath g v1 (*e*#*es*) *v2* = (*src e* = *v1* ∧ *e* ∈ *edges g* ∧ *subpath g* (*tgt e*) *es v2*)
⟨*proof*⟩

lemma *sp-append-one* :
  subpath g v1 (*es*@[*e*]) *v2* = (*subpath g v1 es* (*src e*) ∧ *e* ∈ *edges g* ∧ *tgt e* = *v2*)
⟨*proof*⟩

lemma *sp-append* :
  subpath g v1 (*es1*@*es2*) *v2* = (∃ *v*. *subpath g v1 es1 v* ∧ *subpath g v es2 v2*)
⟨*proof*⟩

A sub-path leads to a unique vertex.

lemma *sp-same-src-imp-same-tgt* :
  assumes *subpath g v es v1*
  assumes *subpath g v es v2*
  shows *v1* = *v2*
⟨*proof*⟩

In the following, we are interested in the evolution of the set of sub-paths of our symbolic execution graph after symbolic execution of a transition from the LTS representation of the program under analysis. Symbolic execution of a transition results in adding to the graph a new edge whose source is already a vertex of this graph, but not its target. The following lemma describes sub-paths ending in the target of such an edge.

Let *e* be an edge whose target has not out-going edges. A sub-path *es* containing *e* ends by *e* and this occurrence of *e* is unique along *es*.

lemma *sp-through-de-decomp* :
  assumes *out-edges g* (*tgt e*) = {}
  assumes *subpath g v1 es v2*
  assumes *e* ∈ *set es*
  shows ∃ *es*′. *es* = *es*′ @ [*e*] ∧ *e* ∉ *set es*′
⟨*proof*⟩

2.3 Adding Edges

This definition and the following lemma are here mainly to ease the definitions and proofs in the next theories.

abbreviation *add-edge* ::
  $('v,'x)$ *rgraph-scheme* $\Rightarrow$ $'v$ *edge* $\Rightarrow$ $('v,'x)$ *rgraph-scheme*
where
  add-edge g e $\equiv$ *rgraph.edges-update* $(\lambda$ *edges. edges* $\cup \{e\})$ *g*

Let *es* be a sub-path from a vertex other than the target of *e* in the graph obtained from *g* by the addition of edge *e*. Moreover, assume that the target of *e* is not a vertex of *g*. Then *e* is an element of *es*.

lemma *sp-ends-in-tgt-imp-mem* :
  assumes *tgt e* $\notin$ *vertices g*
  assumes $v \neq$ *tgt e*
  assumes *subpath* (*add-edge g e*) *v es* (*tgt e*)
  shows $e \in$ *set es*
⟨*proof*⟩

2.4 Trees

We define trees as rooted-graphs in which there exists a unique path leading to each vertex.

definition *is-tree* ::
  $('v,'x)$ *rgraph-scheme* $\Rightarrow$ *bool*
where
  is-tree g $\equiv \forall$ $l \in$ *Graph.vertices g*. $\exists!$ *p*. *Graph.path g p l*

The empty graph is thus a tree.

lemma *empty-graph-is-tree* :
  assumes *edges g* = {}
  shows *is-tree g*
⟨*proof*⟩

end
theory *Aexp*
imports *Main*
begin

3 Arithmetic Expressions

In this section, we model arithmetic expressions as total functions from valuations of program variables to values. This modeling does not take into consideration the syntactic aspects of arithmetic expressions. Thus, our modeling holds for any operator. However, some classical notions, like the set of variables occurring in a given expression for example, must be rethought and defined accordingly.

3.1 Variables and their domain

Note: in the following theories, we distinguish the set of *program variables* and the set of *symbolic variables*. A number of types we define are parameterized by a type of variables. For example, we make a distinction between expressions (arithmetic or boolean) over program variables and expressions over symbolic variables. This distinction eases some parts of the following formalization.

Symbolic variables. A *symbolic variable* is an indexed version of a program variable. In the following type-synonym, we consider that the abstract type *'v* represent the set of program variables. By set of program variables, we do not mean *the set of variables of a given program*, but *the set of variables of all possible programs*. This distinction justifies some of the modeling choices done later. Within Isabelle/HOL, nothing is known about this set. The set of program variables is infinite, though it is not needed to make this assumption. On the other hand, the set of symbolic variables is infinite, independently of the fact that the set of program variables is finite or not.

type-synonym *'v symvar* = *'v* × *nat*

lemma
  ¬ *finite* (*UNIV*::*'v symvar set*)
⟨*proof*⟩

The previous lemma has no name and thus cannot be referenced in the following. Indeed, it is of no use for proving the properties we are interested in. In the following, we will give other unnamed lemmas when we think they might help the reader to understand the ideas behind our modeling choices.

Domain of variables. We call D the domain of program and symbolic variables. In the following, we suppose that D is the set of integers.

3.2 Program and symbolic states

A state is a total function giving values in D to variables. The latter are represented by elements of type $'v$. Unlike in the $'v$ *symvar* type-synonym, here the type $'v$ can stand for program variables as well as symbolic variables. States over program variables are called *program states*, and states over symbolic variables are called *symbolic states*.

type-synonym $('v,'d)$ *state* $= \ 'v \Rightarrow \ 'd$

3.3 The *aexp* type-synonym

Arithmetic (and boolean, see `Bexp.thy`) expressions are represented by their semantics, i.e. total functions giving values in D to states. This way of representing expressions has the benefit that it is not necessary to define the syntax of terms (and formulae) appearing in program statements and path predicates.

type-synonym $('v,'d)$ *aexp* $= \ ('v,'d)$ *state* $\Rightarrow \ 'd$

In order to represent expressions over program variables as well as symbolic variables, the type synonym *aexp* is parameterized by the type of variables. Arithmetic and boolean expressions over program variables are used to express program statements. Arithmetic and boolean expressions over symbolic variables are used to represent the constraints occurring in path predicates during symbolic execution.

3.4 Variables of an arithmetic expression

Expressions being represented by total functions, one can not say that a given variable is occurring in a given expression. We define the set of variables of an expression as the set of variables that can actually have an influence on the value associated by an expression to a state. For example, the set of variables of the expression $\lambda\sigma.\ \sigma\ x - \sigma\ y$ is $\{x,\ y\}$, provided that x and y are distinct variables, and the empty set otherwise. In the second case, this expression would evaluate to 0 for any state σ. Similarly, an expression like $\lambda\sigma.\ \sigma\ x * 0$ is considered as having no variable as if a static evaluation of the multiplication had occurred.

definition *vars* ::
  $('v,'d)$ *aexp* $\Rightarrow \ 'v$ *set*
where
  vars $e = \{v.\ \exists\ \sigma\ val.\ e\ (\sigma(v := val)) \neq e\ \sigma\}$

lemma *vars-example-1* :
  fixes e::($'v$,*integer*) *aexp*
  assumes $e = (\lambda\ \sigma.\ \sigma\ x - \sigma\ y)$
  assumes $x \neq y$
  shows *vars* $e = \{x,y\}$
⟨*proof*⟩

lemma *vars-example-2* :
  fixes e::($'v$,*integer*) *aexp*
  assumes $e = (\lambda\ \sigma.\ \sigma\ x - \sigma\ y)$
  assumes $x = y$
  shows *vars* $e = \{\}$
⟨*proof*⟩

3.5 Fresh variables

Our notion of symbolic execution suppose *static single assignment form.* In order to symbolically execute an assignment, we require the existence of a fresh symbolic variable for the configuration from which symbolic execution is performed. We define here the notion of *freshness* of a variable for an arithmetic expression.

A variable is fresh for an expression if does not belong to its set of variables.

abbreviation *fresh* ::
  $'v \Rightarrow ('v,'d)$ *aexp* $\Rightarrow$ *bool*
where
  fresh $v\ e \equiv v \notin$ *vars* e

end
theory *Bexp*
imports *Aexp*
begin

4 Boolean Expressions

We proceed as in `Aexp.thy`.

4.1 Basic definitions

4.1.1 The *bexp* type-synonym

We represent boolean expressions, their set of variables and the notion of freshness of a variable in the same way than for arithmetic expressions.

type-synonym $('v,'d)$ *bexp* $=$ $('v,'d)$ *state* $\Rightarrow$ *bool*

definition *vars* ::
  $('v,'d)$ *bexp* $\Rightarrow$ $'v$ *set*
where
  $vars\ e = \{v.\ \exists\ \sigma\ val.\ e\ (\sigma(v := val)) \neq e\ \sigma\}$

abbreviation *fresh* ::
  $'v \Rightarrow ('v,'d)\ bexp \Rightarrow bool$
where
  $fresh\ v\ e \equiv v \notin vars\ e$

4.1.2 Satisfiability of an expression

A boolean expression e is satisfiable if there exists a state σ such that $e\ \sigma$ is *true*.

definition *sat* ::
  $('v,'d)$ *bexp* $\Rightarrow$ *bool*
where
  $sat\ e = (\exists\ \sigma.\ e\ \sigma)$

4.1.3 Entailment

A boolean expression φ entails another boolean expression ψ if all states making φ true also make ψ true.

definition *entails* ::
  $('v,'d)\ bexp \Rightarrow ('v,'d)\ bexp \Rightarrow bool$ (**infixl** $\langle \models_B \rangle$ *55*)
where
  $\varphi \models_B \psi \equiv (\forall\ \sigma.\ \varphi\ \sigma \longrightarrow \psi\ \sigma)$

4.1.4 Conjunction

In the following, path predicates are represented by sets of boolean expressions. We define the conjunction of a set of boolean expressions E as the expression that associates *true* to a state σ if, for all elements e of E, e associates *true* to σ.

definition *conjunct* ::
  $('v,'d)$ *bexp set* $\Rightarrow$ $('v,'d)$ *bexp*
where
  conjunct $E \equiv (\lambda\ \sigma.\ \forall\ e \in E.\ e\ \sigma)$

4.2 Properties about the variables of an expression

As said earlier, our definition of symbolic execution requires the existence of a fresh symbolic variable in the case of an assignment. In the following, a number of proof relies on this fact. We will show the existence of such variables assuming the set of symbolic variables already in use is finite and show that symbolic execution preserves the finiteness of this set, under certain conditions. This in turn requires a number of lemmas about the finiteness of boolean expressions. More precisely, when symbolic execution goes through a guard or an assignment, it conjuncts a new expression to the path predicate. In the case of an assignment, this new expression is an equality linking the new symbolic variable associated to the defined program variable to its symbolic value. In the following, we prove that:

1. the conjunction of a finite set of expressions whose sets of variables are finite has a finite set of variables,
2. the equality of two arithmetic expressions whose sets of variables are finite has a finite set of variables.

4.2.1 Variables of a conjunction

The set of variables of the conjunction of two expressions is a subset of the union of the sets of variables of the two sub-expressions. As a consequence, the set of variables of the conjunction of a finite set of expressions whose sets of variables are finite is also finite.

lemma *vars-of-conj* :
  vars $(\lambda\ \sigma.\ e1\ \sigma \wedge e2\ \sigma) \subseteq$ *vars e1* $\cup$ *vars e2*
(**is** *vars ?e* $\subseteq$ *vars e1* $\cup$ *vars e2*)
$\langle$*proof*$\rangle$

lemma *finite-conj* :

assumes *finite E*
assumes $\forall\ e \in E.$ *finite* (*vars e*)
shows *finite* (*vars* (*conjunct E*))
⟨*proof*⟩

4.2.2 Variables of an equality

We proceed analogously for the equality of two arithmetic expressions.

lemma *vars-of-eq-a* :
shows *vars* ($\lambda\ \sigma.\ e1\ \sigma = e2\ \sigma$) $\subseteq$ *Aexp.vars e1* $\cup$ *Aexp.vars e2*
(**is** *vars ?e* $\subseteq$ *Aexp.vars e1* $\cup$ *Aexp.vars e2*)
⟨*proof*⟩

lemma *finite-vars-of-a-eq* :
assumes *finite* (*Aexp.vars e1*)
assumes *finite* (*Aexp.vars e2*)
shows *finite* (*vars* ($\lambda\ \sigma.\ e1\ \sigma = e2\ \sigma$))
⟨*proof*⟩

end
theory *Labels*
imports *Aexp Bexp*
begin

5 Labels

In the following, we model programs by control flow graphs where edges (rather than vertices) are labelled with either assignments or with the condition associated with a branch of a conditional statement. We put a label on every edge : statements that do not modify the program state (like `jump`, `break`, etc) are labelled by a *Skip*.

datatype ('*v*,'*d*) *label* = *Skip* | *Assume* ('*v*,'*d*) *bexp* | *Assign* '*v* ('*v*,'*d*) *aexp*

We say that a label is *finite* if the set of variables of its sub-expression is finite (*Skip* labels are thus considered finite).

definition *finite-label* ::
('*v*,'*d*) *label* ⇒ *bool*
where
finite-label l ≡ *case l of*
Assume e ⇒ *finite* (*Bexp.vars e*)
| *Assign - e* ⇒ *finite* (*Aexp.vars e*)

| - ⇒ *True*

abbreviation *finite-labels* ::
('*v*,'*d*) *label list* ⇒ *bool*
where
finite-labels ls ≡ (∀ *l* ∈ *set ls*. *finite-label l*)

end
theory *Store*
imports *Aexp Bexp*
begin

6 Stores

In this section, we introduce the type of stores, which we use to link program variables with their symbolic counterpart during symbolic execution. We define the notion of consistency of a pair of program and symbolic states w.r.t. a store. This notion will prove helpful when defining various concepts and proving facts related to subsumption (see `Conf.thy`). Finally, we model substitutions that will be performed during symbolic execution (see `SymExec.thy`) by two operations: *adapt-aexp* and *adapt-bexp*.

6.1 Basic definitions

6.1.1 The *store* type-synonym

Symbolic execution performs over configurations (see `Conf.thy`), which are pairs made of:

- a *store* mapping program variables to symbolic variables,
- a set of boolean expressions which records constraints over symbolic variables and whose conjunction is the actual path predicate of the configuration.

We define stores as total functions from program variables to indexes.

type-synonym '*a store* = '*a* ⇒ *nat*

6.1.2 Symbolic variables of a store

The symbolic variable associated to a program variable v by a store s is the couple (v, s v).

definition *symvar* ::
$'a \Rightarrow 'a\ store \Rightarrow 'a\ symvar$
where
$symvar\ v\ s \equiv (v,s\ v)$

The function associating symbolic variables to program variables obtained from s is injective.

lemma
$inj\ (\lambda\ v.\ symvar\ v\ s)$
⟨*proof*⟩

The sets of symbolic variables of a store is the image set of the function *symvar*.

definition *symvars* ::
$'a\ store \Rightarrow 'a\ symvar\ set$
where
$symvars\ s = (\lambda\ v.\ symvar\ v\ s)\ `\ (UNIV::'a\ set)$

6.1.3 Fresh symbolic variables

A symbolic variable is said to be fresh for a store if it is not a member of its set of symbolic variables.

definition *fresh-symvar* ::
$'v\ symvar \Rightarrow 'v\ store \Rightarrow bool$
where
$fresh\text{-}symvar\ sv\ s = (sv \notin symvars\ s)$

6.2 Consistency

We say that a program state σ and a symbolic state σ_{sym} are *consistent* with respect to a store s if, for each variable v, the value associated by σ to v is equal to the value associated by σ_{sym} to the symbolic variable associated to v by s.

definition *consistent* ::
$('v,'d)\ state \Rightarrow ('v\ symvar,\ 'd)\ state \Rightarrow 'v\ store \Rightarrow bool$
where
$consistent\ \sigma\ \sigma_{sym}\ s \equiv (\forall\ v.\ \sigma_{sym}\ (symvar\ v\ s) = \sigma\ v)$

There always exists a couple of consistent states for a given store.

lemma
$\exists\ \sigma\ \sigma_{sym}.\ consistent\ \sigma\ \sigma_{sym}\ s$
⟨*proof*⟩

Moreover, given a store and a program (resp. symbolic) state, one can always build a symbolic (resp. program) state such that the two states are coherent wrt. the store. The four following lemmas show how to build the second state given the first one.

lemma *consistent-eq1* :
  $consistent\ \sigma\ \sigma_{sym}\ s = (\forall\ sv \in symvars\ s.\ \sigma_{sym}\ sv = \sigma\ (fst\ sv))$
$\langle proof \rangle$

lemma *consistent-eq2* :
  $consistent\ \sigma\ \sigma_{sym}\ store = (\sigma = (\lambda\ v.\ \sigma_{sym}\ (symvar\ v\ store)))$
$\langle proof \rangle$

lemma *consistentI1* :
  $consistent\ \sigma\ (\lambda\ sv.\ \sigma\ (fst\ sv))\ store$
$\langle proof \rangle$

lemma *consistentI2* :
  $consistent\ (\lambda\ v.\ \sigma_{sym}\ (symvar\ v\ store))\ \sigma_{sym}\ store$
$\langle proof \rangle$

6.3 Adaptation of an arithmetic expression to a store

Suppose that e is a term representing an arithmetic expression over program variables and let s be a store. We call *adaptation of* e *to* s the term obtained by substituting occurrences of program variables in e by their symbolic counterpart given by s. Since we model arithmetic expressions by total functions and not terms, we define the adaptation of such expressions as follows.

definition *adapt-aexp* ::
  $('v,'d)\ aexp \Rightarrow 'v\ store \Rightarrow ('v\ symvar,'d)\ aexp$
where
  $adapt\text{-}aexp\ e\ s = (\lambda\ \sigma_{sym}.\ e\ (\lambda\ v.\ \sigma_{sym}\ (symvar\ v\ s)))$

Given an arithmetic expression e, a program state σ and a symbolic state σ_{sym} coherent with a store s, the value associated to σ_{sym} by the adaptation of e to s is the same than the value associated by e to σ. This confirms the fact that *adapt-aexp* models the act of substituting occurrences of program variables by their symbolic counterparts in a term over program variables.

lemma *adapt-aexp-is-subst* :

assumes *consistent* σ σ_{sym} *s*
shows (*adapt-aexp e s*) σ_{sym} = *e* σ
⟨*proof*⟩

As said earlier, we will later need to prove that symbolic execution preserves finiteness of the set of symbolic variables in use, which requires that the adaptation of an arithmetic expression to a store preserves finiteness of the set of variables of expressions. We proceed as follows.

First, we show that if *v* is a variable of an expression *e*, then the symbolic variable associated to *v* by a store is a variable of the adaptation of *e* to this store.

lemma *var-imp-symvar-var* :
assumes *v* ∈ *Aexp.vars e*
shows *symvar v s* ∈ *Aexp.vars* (*adapt-aexp e s*) (**is** *?sv* ∈ *Aexp.vars ?e′*)
⟨*proof*⟩

On the other hand, if *sv* is a symbolic variable in the adaptation of an expression to a store, then the program variable it represents is a variable of this expression. This requires to prove that the set of variables of the adaptation of an expression to a store is a subset of the symbolic variables of this store.

lemma *symvars-of-adapt-aexp* :
Aexp.vars (*adapt-aexp e s*) ⊆ *symvars s* (**is** *Aexp.vars ?e′* ⊆ *symvars s*)
⟨*proof*⟩

lemma *symvar-var-imp-var* :
assumes *sv* ∈ *Aexp.vars* (*adapt-aexp e s*) (**is** *sv* ∈ *Aexp.vars ?e′*)
shows *fst sv* ∈ *Aexp.vars e*
⟨*proof*⟩

Thus, we have that the set of variables of the adaptation of an expression to a store is the set of symbolic variables associated by this store to the variables of this expression.

lemma *adapt-aexp-vars* :
Aexp.vars (*adapt-aexp e s*) = (λ *v*. *symvar v s*) ‘ *Aexp.vars e*
⟨*proof*⟩

The fact that the adaptation of an arithmetic expression to a store preserves finiteness of the set of variables trivially follows the previous lemma.

lemma *finite-vars-imp-finite-adapt-a* :

assumes *finite* (*Aexp.vars e*)
shows *finite* (*Aexp.vars* (*adapt-aexp e s*))
⟨*proof*⟩

6.4 Adaptation of a boolean expression to a store

We proceed analogously for the adaptation of boolean expressions to a store.

definition *adapt-bexp* ::
$('v,'d)$ *bexp* $\Rightarrow$ *'v store* $\Rightarrow$ $('v\ symvar,'d)$ *bexp*
where
adapt-bexp e s $= (\lambda\ \sigma.\ e\ (\lambda\ x.\ \sigma\ (symvar\ x\ s)))$

lemma *adapt-bexp-is-subst* :
assumes *consistent* $\sigma\ \sigma_{sym}\ s$
shows (*adapt-bexp e s*) $\sigma_{sym} = e\ \sigma$
⟨*proof*⟩

lemma *var-imp-symvar-var2* :
assumes $v \in$ *Bexp.vars e*
shows *symvar v s* $\in$ *Bexp.vars* (*adapt-bexp e s*) (**is** *?sv* $\in$ *Bexp.vars ?e'*)
⟨*proof*⟩

lemma *symvars-of-adapt-bexp* :
Bexp.vars (*adapt-bexp e s*) $\subseteq$ *symvars s* (**is** *Bexp.vars ?e'* $\subseteq$ *?SV*)
⟨*proof*⟩

lemma *symvar-var-imp-var2* :
assumes *sv* $\in$ *Bexp.vars* (*adapt-bexp e s*) (**is** *sv* $\in$ *Bexp.vars ?e'*)
shows *fst sv* $\in$ *Bexp.vars e*
⟨*proof*⟩

lemma *adapt-bexp-vars* :
Bexp.vars (*adapt-bexp e s*) $= (\lambda\ v.\ symvar\ v\ s)$ ‘ *Bexp.vars e*
(**is** *Bexp.vars ?e'* $=$ *?R*)
⟨*proof*⟩

lemma *finite-vars-imp-finite-adapt-b* :
assumes *finite* (*Bexp.vars e*)

shows *finite* (*Bexp.vars* (*adapt-bexp e s*))
⟨*proof*⟩

end
theory *Conf*
imports *Store*
begin

7 Configurations, Subsumption and Symbolic Execution

In this section, we first introduce most elements related to our modeling of program behaviors. We first define the type of configurations, on which symbolic execution performs, and define the various concepts we will rely upon in the following and state and prove properties about them. Then, we introduce symbolic execution. After giving a number of basic properties about symbolic execution, we prove that symbolic execution is monotonic with respect to the subsumption relation, which is a crucial point in order to prove the main theorems of `RB.thy`. Moreover, Isabelle/HOL requires the actual formalization of a number of facts one would not worry when implementing or writing a sketch proof. Here, we will need to prove that there exist successors of the configurations on which symbolic execution is performed. Although this seems quite obvious in practice, proofs of such facts will be needed a number of times in the following theories. Finally, we define the feasibility of a sequence of labels.

7.1 Basic Definitions and Properties

7.1.1 Configurations

Configurations are pairs (*store*, *pred*) where:

- *store* is a store mapping program variable to symbolic variables,
- *pred* is a set of boolean expressions over program variables whose conjunction is the actual path predicate.

record (′*v*,′*d*) *conf* =
store :: ′*v store*
pred :: (′*v symvar*,′*d*) *bexp set*

7.1.2 Symbolic variables of a configuration.

The set of symbolic variables of a configuration is the union of the set of symbolic variables of its store component with the set of variables of its path predicate.

definition *symvars* ::
  $('v,'d)$ *conf* $\Rightarrow$ *'v symvar set*
where
  symvars c = *Store.symvars* (*store c*) $\cup$ *Bexp.vars* (*conjunct* (*pred c*))

7.1.3 Freshness.

A symbolic variable is said to be fresh for a configuration if it is not an element of its set of symbolic variables.

definition *fresh-symvar* ::
  'v symvar $\Rightarrow$ $('v,'d)$ *conf* $\Rightarrow$ *bool*
where
  fresh-symvar sv c = (*sv* $\notin$ *symvars c*)

7.1.4 Satisfiability

A configuration is said to be satisfiable if its path predicate is satisfiable.

abbreviation *sat* ::
  $('v,'d)$ *conf* $\Rightarrow$ *bool*
where
  sat c $\equiv$ *Bexp.sat* (*conjunct* (*pred c*))

7.1.5 States of a configuration

Configurations represent sets of program states. The set of program states represented by a configuration, or simply its set of program states, is defined as the set of program states such that consistent symbolic states wrt. the store component of the configuration satisfies its path predicate.

definition *states* ::
  $('v,'d)$ *conf* $\Rightarrow$ $('v,'d)$ *state set*
where
  states c = $\{\sigma.\ \exists\ \sigma_{sym}.$ *consistent* $\sigma\ \sigma_{sym}$ (*store c*) $\wedge$ *conjunct* (*pred c*) $\sigma_{sym}\}$

A configuration is satisfiable if and only if its set of states is not empty.

lemma *sat-eq* :
  sat c = (*states c* $\neq$ $\{\}$)
$\langle$*proof*$\rangle$

7.1.6 Subsumption

A configuration c_2 is subsumed by a configuration c_1 if the set of states of c_2 is a subset of the set of states of c_1.

definition *subsums* ::
  $('v,'d)$ *conf* $\Rightarrow$ $('v,'d)$ *conf* $\Rightarrow$ *bool* (**infixl** ‹$\sqsubseteq$› *55*)
where
  $c_2 \sqsubseteq c_1 \equiv (\mathit{states}\ c_2 \subseteq \mathit{states}\ c_1)$

The subsumption relation is reflexive and transitive.

lemma *subsums-refl* :
  $c \sqsubseteq c$
⟨*proof*⟩

lemma *subsums-trans* :
  $c1 \sqsubseteq c2 \Longrightarrow c2 \sqsubseteq c3 \Longrightarrow c1 \sqsubseteq c3$
⟨*proof*⟩

However, it is not anti-symmetric. This is due to the fact that different configurations can have the same sets of program states. However, the following lemma trivially follows the definition of subsumption.

lemma
  assumes $c1 \sqsubseteq c2$
  assumes $c2 \sqsubseteq c1$
  shows $\mathit{states}\ c1 = \mathit{states}\ c2$
⟨*proof*⟩

A satisfiable configuration can only be subsumed by satisfiable configurations.

lemma *sat-sub-by-sat* :
  assumes $\mathit{sat}\ c_2$
  and $c_2 \sqsubseteq c_1$
  shows $\mathit{sat}\ c_1$
⟨*proof*⟩

On the other hand, an unsatisfiable configuration can only subsume unsatisfiable configurations.

lemma *unsat-subs-unsat* :
  assumes $\neg\ \mathit{sat}\ c1$
  assumes $c2 \sqsubseteq c1$
  shows $\neg\ \mathit{sat}\ c2$
⟨*proof*⟩

7.1.7 Semantics of a configuration

The semantics of a configuration c is a boolean expression e over program states associating *true* to a program state if it is a state of c. In practice, given two configurations c_1 and c_2, it is not possible to enumerate their sets of states to establish the inclusion in order to detect a subsumption. We detect the subsumption of the former by the latter by asking a constraint solver if *sem* c_1 entails *sem* c_2. The following theorem shows that the way we detect subsumption in practice is correct.

definition *sem* ::
$('v,'d)\ conf \Rightarrow ('v,'d)\ bexp$
where
$sem\ c = (\lambda\ \sigma.\ \sigma \in states\ c)$

theorem
$c_2 \sqsubseteq c_1 \longleftrightarrow sem\ c_2 \models_B sem\ c_1$
⟨*proof*⟩

7.1.8 Abstractions

Abstracting a configuration consists in removing a given expression from its *pred* component, i.e. weakening its path predicate. This definition of abstraction motivates the fact that the *pred* component of configurations has been defined as a set of boolean expressions instead of a boolean expression.

definition *abstract* ::
$('v,'d)\ conf \Rightarrow ('v,'d)\ conf \Rightarrow bool$
where
$abstract\ c\ c_a \equiv c \sqsubseteq c_a$

7.1.9 Entailment

A configuration *entails* a boolean expression if its semantics entails this expression. This is equivalent to say that this expression holds for any state of this configuration.

abbreviation *entails* ::
$('v,'d)\ conf \Rightarrow ('v,'d)\ bexp \Rightarrow bool$ (**infixl** ‹$\models_c$› *55*)
where
$c \models_c \varphi \equiv sem\ c \models_B \varphi$

lemma
$sem\ c \models_B e \longleftrightarrow (\forall\ \sigma \in states\ c.\ e\ \sigma)$

⟨*proof*⟩

end
theory *SymExec*
imports *Conf Labels*
begin

7.2 Symbolic Execution

We model symbolic execution by an inductive predicate *se* which takes two configurations c_1 and c_2 and a label l and evaluates to *true* if and only if c_2 is a *possible result* of the symbolic execution of l from c_1. We say that c_2 is a possible result because, when l is of the form *Assign v e*, we associate a fresh symbolic variable to the program variable v, but we do no specify how this fresh variable is chosen (see the two assumptions in the third case). We could have model *se* (and *se-star*) by a function producing the new configuration, instead of using inductive predicates. However this would require to provide the two said assumptions in each lemma involving *se*, which is not necessary using a predicate. Modeling symbolic execution in this way has the advantage that it simplifies the following proofs while not requiring additional lemmas.

7.2.1 Definitions of *se* and *se_star*

Symbolic execution of *Skip* does not change the configuration from which it is performed.

When the label is of the form *Assume e*, the adaptation of e to the store is added to the *pred* component.

In the case of an assignment, the *store* component is updated such that it now maps a fresh symbolic variable to the assigned program variable. A constraint relating this program variable with its new symbolic value is added to the *pred* component.

The second assumption in the third case requires that there exists at least one fresh symbolic variable for c. In the following, a number of theorems are needed to show that such variables exist for the configurations on which symbolic execution is performed.

inductive *se* ::

$('v,'d)\ conf \Rightarrow ('v,'d)\ label \Rightarrow ('v,'d)\ conf \Rightarrow bool$
where
se c Skip c

| $se\ c\ (Assume\ e)\ (\!|\ store = store\ c,\ pred = pred\ c \cup \{adapt\text{-}bexp\ e\ (store\ c)\}\ |\!)$

| $fst\ sv = v \Longrightarrow$
$fresh\text{-}symvar\ sv\ c \Longrightarrow$
$se\ c\ (Assign\ v\ e)\ (\!|\ store = (store\ c)(v := snd\ sv),$
$pred = pred\ c \cup \{(\lambda\ \sigma.\ \sigma\ sv = (adapt\text{-}aexp\ e\ (store\ c))\ \sigma)\}\ |\!)$

In the same spirit, we extend symbolic execution to sequence of labels.

inductive $se\text{-}star :: ('v,'d)\ conf \Rightarrow ('v,'d)\ label\ list \Rightarrow ('v,'d)\ conf \Rightarrow bool$ **where**
se-star c [] c
| $se\ c1\ l\ c2 \Longrightarrow se\text{-}star\ c2\ ls\ c3 \Longrightarrow se\text{-}star\ c1\ (l\ \#\ ls)\ c3$

7.2.2 Basic properties of *se*

If symbolic execution yields a satisfiable configuration, then it has been performed from a satisfiable configuration.

lemma *se-sat-imp-sat* :
assumes $se\ c\ l\ c'$
assumes $sat\ c'$
shows $sat\ c$
⟨*proof*⟩

If symbolic execution is performed from an unsatisfiable configuration, then it will yield an unsatisfiable configuration.

lemma *unsat-imp-se-unsat* :
assumes $se\ c\ l\ c'$
assumes $\neg\ sat\ c$
shows $\neg\ sat\ c'$
⟨*proof*⟩

Given two configurations c and c' and a label l such that $se\ c\ l\ c'$, the three following lemmas express c' as a function of c.

lemma [*simp*] :
$se\ c\ Skip\ c' = (c' = c)$
⟨*proof*⟩

lemma *se-Assume-eq* :
$se\ c\ (Assume\ e)\ c' = (c' = (\!|\ store = store\ c,\ pred = pred\ c \cup \{adapt\text{-}bexp\ e\ (store\ c)\}\ |\!))$

$\langle proof \rangle$

lemma *se-Assign-eq* :
  $se\ c\ (Assign\ v\ e)\ c' =$
  $(\exists\ sv.\ fresh\text{-}symvar\ sv\ c$
    $\wedge\ fst\ sv = v$
    $\wedge\ c' = (\!|\ store = (store\ c)(v := snd\ sv),$
      $pred\ = insert\ (\lambda\sigma.\ \sigma\ sv = adapt\text{-}aexp\ e\ (store\ c)\ \sigma)\ (pred\ c)|\!))$
$\langle proof \rangle$

Given two configurations c and c' and a label l such that $se\ c\ l\ c'$, the two following lemmas express the path predicate of c' as a function of the path predicate of c when l models a guard or an assignment.

lemma *path-pred-of-se-Assume* :
  assumes $se\ c\ (Assume\ e)\ c'$
  shows $conjunct\ (pred\ c') =$
    $(\lambda\ \sigma.\ conjunct\ (pred\ c)\ \sigma \wedge adapt\text{-}bexp\ e\ (store\ c)\ \sigma)$
$\langle proof \rangle$

lemma *path-pred-of-se-Assign* :
  assumes $se\ c\ (Assign\ v\ e)\ c'$
  shows $\exists\ sv.\ conjunct\ (pred\ c') =$
    $(\lambda\ \sigma.\ conjunct\ (pred\ c)\ \sigma \wedge \sigma\ sv = adapt\text{-}aexp\ e\ (store\ c)\ \sigma)$
$\langle proof \rangle$

Let c and c' be two configurations such that c' is obtained from c by symbolic execution of a label of the form *Assume e*. The states of c' are the states of c that satisfy e. This theorem will help prove that symbolic execution is monotonic wrt. subsumption.

theorem *states-of-se-assume* :
  assumes $se\ c\ (Assume\ e)\ c'$
  shows $states\ c' = \{\sigma \in states\ c.\ e\ \sigma\}$
$\langle proof \rangle$

Let c and c' be two configurations such that c' is obtained from c by symbolic execution of a label of the form *Assign v e*. We want to express the set of states of c' as a function of the set of states of c. Since the proof requires a number of details, we split into two sub lemmas.

First, we show that if σ' is a state of c', then it has been obtain from an adequate update of a state σ of c.

lemma *states-of-se-assign1* :
  assumes $se\ c\ (Assign\ v\ e)\ c'$
  assumes $\sigma' \in states\ c'$
  shows $\exists\ \sigma \in states\ c.\ \sigma' = (\sigma\ (v := e\ \sigma))$
$\langle proof \rangle$

Then, we show that if there exists a state σ of c from which σ' is obtained by an adequate update, then σ' is a state of c'.

lemma *states-of-se-assign2* :
  assumes $se\ c\ (Assign\ v\ e)\ c'$
  assumes $\exists\ \sigma \in states\ c.\ \sigma' = \sigma\ (v := e\ \sigma)$
  shows $\sigma' \in states\ c'$
$\langle proof \rangle$

The following theorem expressing the set of states of c' as a function of the set of states of c trivially follows the two preceding lemmas.

theorem *states-of-se-assign* :
  assumes $se\ c\ (Assign\ v\ e)\ c'$
  shows $states\ c' = \{\sigma\ (v := e\ \sigma) \mid \sigma.\ \sigma \in states\ c\}$
$\langle proof \rangle$

7.2.3 Monotonicity of *se*

We are now ready to prove that symbolic execution is monotonic with respect to subsumption.

theorem *se-mono-for-sub* :
  assumes $se\ c1\ l\ c1'$
  assumes $se\ c2\ l\ c2'$
  assumes $c2 \sqsubseteq c1$
  shows $c2' \sqsubseteq c1'$
$\langle proof \rangle$

A stronger version of the previous theorem: symbolic execution is monotonic with respect to states equality.

theorem *se-mono-for-states-eq* :
  assumes $states\ c1 = states\ c2$
  assumes $se\ c1\ l\ c1'$
  assumes $se\ c2\ l\ c2'$
  shows $states\ c2' = states\ c1'$
$\langle proof \rangle$

The previous theorem confirms the fact that the way the fresh symbolic variable is chosen in the case of symbolic execution of an assignment does not matter as long as the new symbolic variable is indeed fresh, which is more precisely expressed by the following lemma.

lemma *se-succs-states* :
 assumes *se c l c1*
 assumes *se c l c2*
 shows *states c1 = states c2*
⟨*proof*⟩

7.2.4 Basic properties of *se_star*

Some simplification lemmas for *se-star*.

lemma [*simp*] :
 se-star c [] *c′* = (*c′* = *c*)
⟨*proof*⟩

lemma *se-star-Cons* :
 se-star c1 (*l* # *ls*) *c2* = (∃ *c*. *se c1 l c* ∧ *se-star c ls c2*)
⟨*proof*⟩

lemma *se-star-one* :
 se-star c1 [*l*] *c2* = *se c1 l c2*
⟨*proof*⟩

lemma *se-star-append* :
 se-star c1 (*ls1* @ *ls2*) *c2* = (∃ *c*. *se-star c1 ls1 c* ∧ *se-star c ls2 c2*)
⟨*proof*⟩

lemma *se-star-append-one* :
 se-star c1 (*ls* @ [*l*]) *c2* = (∃ *c*. *se-star c1 ls c* ∧ *se c l c2*)
⟨*proof*⟩

Symbolic execution of a sequence of labels from an unsatisfiable configuration yields an unsatisfiable configuration.

lemma *unsat-imp-se-star-unsat* :
 assumes *se-star c ls c′*
 assumes ¬ *sat c*
 shows ¬ *sat c′*
⟨*proof*⟩

If symbolic execution yields a satisfiable configuration, then it has been performed from a satisfiable configuration.

lemma *se-star-sat-imp-sat* :
assumes *se-star c ls c′*
assumes *sat c′*
shows *sat c*
⟨*proof*⟩

7.2.5 Monotonicity of *se__star*

Monotonicity of *se* extends to *se-star*.

theorem *se-star-mono-for-sub* :
assumes *se-star c1 ls c1′*
assumes *se-star c2 ls c2′*
assumes $c2 \sqsubseteq c1$
shows $c2' \sqsubseteq c1'$
⟨*proof*⟩

lemma *se-star-mono-for-states-eq* :
assumes *states c1 = states c2*
assumes *se-star c1 ls c1′*
assumes *se-star c2 ls c2′*
shows *states c2′ = states c1′*
⟨*proof*⟩

lemma *se-star-succs-states* :
assumes *se-star c ls c1*
assumes *se-star c ls c2*
shows *states c1 = states c2*
⟨*proof*⟩

7.2.6 Existence of successors

Here, we are interested in proving that, under certain assumptions, there will always exist fresh symbolic variables for configurations on which symbolic execution is performed. Thus symbolic execution cannot "block" when an assignment is met. For symbolic execution not to block in this case, the configuration from which it is performed must be such that there exist fresh symbolic variables for each program variable. Such configurations are said to be *updatable*.

definition *updatable* ::
$('v,'d)$ *conf* $\Rightarrow$ *bool*
where
updatable $c \equiv \forall\ v.\ \exists\ sv.\ fst\ sv = v \wedge fresh\text{-}symvar\ sv\ c$

The following lemma shows that being updatable is a sufficient condition for a configuration in order for *se* not to block.

lemma *updatable-imp-ex-se-suc* :
assumes *updatable c*
shows $\exists\ c'.\ se\ c\ l\ c'$
⟨*proof*⟩

A sufficient condition for a configuration to be updatable is that its path predicate has a finite number of variables. The *store* component has no influence here, since its set of symbolic variables is always a strict subset of the set of symbolic variables (i.e. there always exist fresh symbolic variables for a store). To establish this proof, we need the following intermediate lemma.

We want to prove that if the set of symbolic variables of the path predicate of a configuration is finite, then we can find a fresh symbolic variable for it. However, we express this with a more general lemma. We show that given a finite set of symbolic variables SV and a program variable v such that there exist symbolic variables in SV that are indexed versions of v, then there exists a symbolic variable for v whose index is greater or equal than the index of any other symbolic variable for v in SV.

lemma *finite-symvars-imp-ex-greatest-symvar* :
fixes SV :: *'a symvar set*
assumes *finite* SV
assumes $\exists\ sv \in SV.\ fst\ sv = v$
shows $\exists\ sv\ \in \{sv \in SV.\ fst\ sv = v\}.$
$\forall\ sv' \in \{sv \in SV.\ fst\ sv = v\}.\ snd\ sv' \leq snd\ sv$
⟨*proof*⟩

Thus, a configuration whose path predicate has a finite set of variables is updatable. For example, for any program variable v, the symbolic variable $(v,i{+}1)$ is fresh for this configuration, where i is the greater index associated to v among the symbolic variables of this configuration. In practice, this is how we choose the fresh symbolic variable.

lemma *finite-pred-imp-se-updatable* :
assumes *finite* (*Bexp.vars* (*conjunct* (*pred c*))) (**is** *finite ?V*)
shows *updatable c*

⟨*proof*⟩

The path predicate of a configuration whose *pred* component is finite and whose elements all have finite sets of variables has a finite set of variables. Thus, this configuration is updatable, and it has a successor by symbolic execution of any label. The following lemma starts from these two assumptions and use the previous ones in order to directly get to the conclusion (this will ease some of the following proofs).

lemma *finite-imp-ex-se-succ* :
  assumes *finite* (*pred c*)
  assumes $\forall\ e \in$ *pred c*. *finite* (*Bexp.vars e*)
  shows $\exists\ c'$. *se c l* c'
⟨*proof*⟩

For symbolic execution not to block *along a sequence of labels*, it is not sufficient for the first configuration to be updatable. It must also be such that (all) its successors are updatable. A sufficient condition for this is that the set of variables of its path predicate is finite and that the sub-expression of the label that is executed also has a finite set of variables. Under these assumptions, symbolic execution preserves finiteness of the *pred* component and of the sets of variables of its elements. Thus, successors *se* are also updatable because they also have a path predicate with a finite set of variables. In the following, to prove this we need two intermediate lemmas:

- one stating that symbolic execution perserves the finiteness of the set of variables of the elements of the *pred* component, provided that the sub expression of the label that is executed has a finite set of variables,
- one stating that symbolic execution preserves the finiteness of the *pred* component.

lemma *se-preserves-finiteness1* :
  assumes *finite-label l*
  assumes *se c l* c'
  assumes $\forall\ e \in$ *pred c*. *finite* (*Bexp.vars e*)
  shows $\forall\ e \in$ *pred* c'. *finite* (*Bexp.vars e*)
⟨*proof*⟩

lemma *se-preserves-finiteness2* :
  assumes *se c l* c'
  assumes *finite* (*pred c*)

shows *finite* (*pred* c')
⟨*proof*⟩

We are now ready to prove that a sufficient condition for symbolic execution not to block along a sequence of labels is that the *pred* component of the "initial configuration" is finite, as well as the set of variables of its elements, and that the sub-expression of the label that is executed also has a finite set of variables.

lemma *finite-imp-ex-se-star-succ* :
assumes *finite* (*pred c*)
assumes ∀ *e* ∈ *pred c*. *finite* (*Bexp.vars e*)
assumes *finite-labels ls*
shows ∃ c'. *se-star c ls* c'
⟨*proof*⟩

7.3 Feasibility of a sequence of labels

A sequence of labels *ls* is said to be feasible from a configuration *c* if there exists a satisfiable configuration c' obtained by symbolic execution of *ls* from *c*.

definition *feasible* :: ($'v$,$'d$) *conf* ⇒ ($'v$,$'d$) *label list* ⇒ *bool* **where**
feasible c ls ≡ (∃ c'. *se-star c ls* c' ∧ *sat* c')

A simplification lemma for the case where *ls* is not empty.

lemma *feasible-Cons* :
feasible c (*l#ls*) = (∃ c'. *se c l* c' ∧ *sat* c' ∧ *feasible* c' *ls*)
⟨*proof*⟩

The following theorem is very important for the rest of this formalization. It states that, given two configurations *c1* and *c2* such that *c1* subsums *c2*, then any feasible sequence of labels from *c2* is also feasible from *c1*. This is a crucial point in order to prove that our approach preserves the set of feasible paths of the original LTS. This proof requires a number of assumptions about the finiteness of the sequence of labels, of the path predicates of the two configurations and of their states of variables. Those assumptions are needed in order to show that there exist successors of both configurations by symbolic execution of the sequence of labels.

lemma *subsums-imp-feasible* :
assumes *finite-labels ls*
assumes *finite* (*pred c1*)
assumes *finite* (*pred c2*)
assumes ∀ *e* ∈ *pred c1*. *finite* (*Bexp.vars e*)

assumes $\forall\ e \in pred\ c2.\ finite\ (Bexp.vars\ e)$
assumes $c2 \sqsubseteq c1$
assumes $feasible\ c2\ ls$
shows $feasible\ c1\ ls$
$\langle proof \rangle$

7.4 Concrete execution

We illustrate our notion of symbolic execution by relating it with *ce*, an inductive predicate describing concrete execution. Unlike symbolic execution, concrete execution describes program behavior given program states, i.e. concrete valuations for program variables. The goal of this section is to show that our notion of symbolic execution is correct, that is: given two configurations such that one results from the symbolic execution of a sequence of labels from the other, then the resulting configuration represents the set of states that are reachable by concrete execution from the states of the original configuration.

inductive ce ::
$('v,'d)\ state \Rightarrow ('v,'d)\ label \Rightarrow ('v,'d)\ state \Rightarrow bool$
where
$ce\ \sigma\ Skip\ \sigma$
$|\ e\ \sigma \Longrightarrow ce\ \sigma\ (Assume\ e)\ \sigma$
$|\ ce\ \sigma\ (Assign\ v\ e)\ (\sigma(v := e\ \sigma))$

inductive $ce\text{-}star :: ('v,'d)\ state \Rightarrow ('v,'d)\ label\ list \Rightarrow ('v,'d)\ state \Rightarrow bool$ **where**
$ce\text{-}star\ c\ []\ c$
$|\ ce\ c1\ l\ c2 \Longrightarrow ce\text{-}star\ c2\ ls\ c3 \Longrightarrow ce\text{-}star\ c1\ (l\ \#\ ls)\ c3$

lemma $[simp]$:
$ce\ \sigma\ Skip\ \sigma' = (\sigma' = \sigma)$
$\langle proof \rangle$

lemma $[simp]$:
$ce\ \sigma\ (Assume\ e)\ \sigma' = (\sigma' = \sigma \wedge e\ \sigma)$
$\langle proof \rangle$

lemma $[simp]$:
$ce\ \sigma\ (Assign\ v\ e)\ \sigma' = (\sigma' = \sigma(v := e\ \sigma))$
$\langle proof \rangle$

lemma $se\text{-}as\text{-}ce$:
assumes $se\ c\ l\ c'$
shows $states\ c' = \{\sigma'.\ \exists\ \sigma \in states\ c.\ ce\ \sigma\ l\ \sigma'\}$

⟨*proof*⟩

lemma [*simp*] :
  $\textit{ce-star}\ \sigma\ []\ \sigma' = (\sigma' = \sigma)$
⟨*proof*⟩

lemma *ce-star-Cons* :
  $\textit{ce-star}\ \sigma 1\ (l\ \#\ ls)\ \sigma 2 = (\exists\ \sigma.\ \textit{ce}\ \sigma 1\ l\ \sigma \wedge \textit{ce-star}\ \sigma\ ls\ \sigma 2)$
⟨*proof*⟩

lemma *se-star-as-ce-star* :
  assumes $\textit{se-star}\ c\ ls\ c'$
  shows $\textit{states}\ c' = \{\sigma'.\ \exists\ \sigma \in \textit{states}\ c.\ \textit{ce-star}\ \sigma\ ls\ \sigma'\}$
⟨*proof*⟩

end
theory *LTS*
imports *Graph Labels SymExec*
begin

8 Labelled Transition Systems

This theory is motivated by the need of an abstract representation of control-flow graphs (CFG). It is a refinement of the prior theory of (unlabelled) graphs and proceeds by decorating their edges with *labels* expressing assumptions and effects (assignments) on an underlying state. In this theory, we define LTSs and introduce a number of abbreviations that will ease stating and proving lemmas in the following theories.

8.1 Basic definitions

The labelled transition systems (LTS) we are heading for are constructed by extending *rgraph*'s by a labelling function of the edges, using Isabelle extensible records.

record $('vert,'var,'d)\ lts = 'vert\ rgraph\ +$
  $labelling :: 'vert\ edge \Rightarrow ('var,'d)\ label$

We call *initial location* the root of the underlying graph.

abbreviation *init* ::
$('vert,'var,'d,'x)\ \textit{lts-scheme} \Rightarrow 'vert$
where

init lts ≡ *root lts*

The set of labels of a LTS is the image set of its labelling function over its set of edges.

abbreviation *labels* ::
  (*'vert*,*'var*,*'d*,*'x*) *lts-scheme* ⇒ (*'var*,*'d*) *label set*
where
  labels lts ≡ *labelling lts* ‘ *edges lts*

In the following, we will sometimes need to use the notion of *trace* of a given sequence of edges with respect to the transition relation of an LTS.

abbreviation *trace* ::
  'vert edge list ⇒ (*'vert edge* ⇒ (*'var*,*'d*) *label*) ⇒ (*'var*,*'d*) *label list*
where
  trace as L ≡ *map L as*

We are interested in a special form of Labelled Transition Systems; the prior record definition is too liberal. We will constrain it to *well-formed labelled transition systems*.

We first define an application that, given an LTS, returns its underlying graph.

abbreviation *graph* ::
  (*'vert*,*'var*,*'d*,*'x*) *lts-scheme* ⇒ *'vert rgraph*
where
  graph lts ≡ *rgraph.truncate lts*

An LTS is well-formed if its underlying *rgraph* is well-formed.

abbreviation *wf-lts* ::
  (*'vert*,*'var*,*'d*,*'x*) *lts-scheme* ⇒ *bool*
where
  wf-lts lts ≡ *wf-rgraph* (*graph lts*)

In the following theories, we will sometimes need to account for the fact that we consider LTSs with a finite number of edges.

abbreviation *finite-lts* ::
  (*'vert*,*'var*,*'d*,*'x*) *lts-scheme* ⇒ *bool*
where
  finite-lts lts ≡ ∀ *l* ∈ *range* (*labelling lts*). *finite-label l*

8.2 Feasible sub-paths and paths

A sequence of edges is a feasible sub-path of an LTS *lts* from a configuration *c* if it is a sub-path of the underlying graph of *lts* and if it is feasible from the configuration *c*.

abbreviation *feasible-subpath* ::
  $('vert,'var,'d,'x)$ *lts-scheme* $\Rightarrow$ $('var,'d)$ *conf* $\Rightarrow$ *'vert* $\Rightarrow$ *'vert edge list* $\Rightarrow$ *'vert* $\Rightarrow$ *bool*
where
  feasible-subpath lts pc l1 as l2 $\equiv$ *Graph.subpath lts l1 as l2*
    $\wedge$ *feasible pc* (*trace as* (*labelling lts*))

Similarly to sub-paths in rooted-graphs, we will not be always interested in the final vertex of a feasible sub-path. We use the following notion when we are not interested in this vertex.

abbreviation *feasible-subpath-from* ::
  $('vert,'var,'d,'x)$ *lts-scheme* $\Rightarrow$ $('var,'d)$ *conf* $\Rightarrow$ *'vert* $\Rightarrow$ *'vert edge list* $\Rightarrow$ *bool*
where
  feasible-subpath-from lts pc l as $\equiv$ $\exists$ *l'*. *feasible-subpath lts pc l as l'*

abbreviation *feasible-subpaths-from* ::
  $('vert,'var,'d,'x)$ *lts-scheme* $\Rightarrow$ $('var,'d)$ *conf* $\Rightarrow$ *'vert* $\Rightarrow$ *'vert edge list set*
where
  feasible-subpaths-from lts pc l $\equiv$ {*ts*. *feasible-subpath-from lts pc l ts*}

As earlier, feasible paths are defined as feasible sub-paths starting at the initial location of the LTS.

abbreviation *feasible-path* ::
  $('vert,'var,'d,'x)$ *lts-scheme* $\Rightarrow$ $('var,'d)$ *conf* $\Rightarrow$ *'vert edge list* $\Rightarrow$ *'vert* $\Rightarrow$ *bool*
where
  feasible-path lts pc as l $\equiv$ *feasible-subpath lts pc* (*init lts*) *as l*

abbreviation *feasible-paths* ::
  $('vert,'var,'d,'x)$ *lts-scheme* $\Rightarrow$ $('var,'d)$ *conf* $\Rightarrow$ *'vert edge list set*
where
  feasible-paths lts pc $\equiv$ {*as*. $\exists$ *l*. *feasible-path lts pc as l*}

end
theory *SubRel*
imports *Graph*
begin

9 Graphs equipped with a subsumption relation

In this section, we define subsumption relations and the notion of sub-paths in rooted graphs equipped with such relations. Sub-paths are defined in the same way than in `Graph.thy`: first we define the consistency of a sequence of edges in presence of a subsumption relation, then sub-paths. We are interested in subsumptions taking places between red vertices of red-black graphs (see `RB.thy`), i.e. occurrences of locations of LTSs. Here subsumptions are defined as pairs of indexed vertices of a LTS, and subsumption relations as sets of subsumptions. The type of vertices of such LTSs is represented by the abstract type $'v$ in the following.

9.1 Basic definitions and properties

9.1.1 Subsumptions and subsumption relations

Subsumptions take place between occurrences of the vertices of a graph. We represent such occurrences by indexed versions of vertices. A subsumption is defined as pair of indexed vertices.

type-synonym $'v\ \mathit{sub\text{-}t} = (('v \times \mathit{nat}) \times ('v \times \mathit{nat}))$

A subsumption relation is a set of subsumptions.

type-synonym $'v\ \mathit{sub\text{-}rel\text{-}t} = 'v\ \mathit{sub\text{-}t}\ \mathit{set}$

We consider the left member to be subsumed by the right one. The left member of a subsumption is called its *subsumee*, the right member its *subsumer*.

abbreviation *subsumee* ::
  $'v\ \mathit{sub\text{-}t} \Rightarrow ('v \times \mathit{nat})$
where
  $\mathit{subsumee}\ \mathit{sub} \equiv \mathit{fst}\ \mathit{sub}$

abbreviation *subsumer* ::
  $'v\ \mathit{sub\text{-}t} \Rightarrow ('v \times \mathit{nat})$
where
  $\mathit{subsumer}\ \mathit{sub} \equiv \mathit{snd}\ \mathit{sub}$

We will need to talk about the sets of subsumees and subsumers of a subsumption relation.

abbreviation *subsumees* ::
  $'v\ \mathit{sub\text{-}rel\text{-}t} \Rightarrow ('v \times \mathit{nat})\ \mathit{set}$
where
  $\mathit{subsumees}\ \mathit{subs} \equiv \mathit{subsumee}\ `\ \mathit{subs}$

abbreviation *subsumers* ::
$'v$ *sub-rel-t* $\Rightarrow$ $('v \times nat)$ *set*
where
subsumers subs $\equiv$ *subsumer* ‘ *subs*

The two following lemmas will prove useful in the following.

lemma *subsumees-conv* :
subsumees subs $= \{v.\ \exists\ v'.\ (v,v') \in subs\}$
$\langle proof \rangle$

lemma *subsumers-conv* :
subsumers subs $= \{v'.\ \exists\ v.\ (v,v') \in subs\}$
$\langle proof \rangle$

We call set of vertices of the relation the union of its sets of subsumees and subsumers.

abbreviation *vertices* ::
$'v$ *sub-rel-t* $\Rightarrow$ $('v \times nat)$ *set*
where
vertices subs $\equiv$ *subsumers subs* $\cup$ *subsumees subs*

9.2 Well-formed subsumption relation of a graph

9.2.1 Well-formed subsumption relations

In the following, we make an intensive use of *locales*. We use them as a convenient way to add assumptions to the following lemmas, in order to ease their reading. Locales can be built from locales, allowing some modularity in the formalization. The following locale simply states that we suppose there exists a subsumption relation called *subs*. It will be used later in order to constrain subsumption relations.

locale *sub-rel* =
fixes *subs* :: $'v$ *sub-rel-t* (**structure**)

We are only interested in subsumptions involving two different occurrences of the same LTS location. Moreover, once a vertex has been subsumed, there is no point in trying to subsume it again by another subsumer: subsumees must have a unique subsumer. Finally, we do not allow chains of subsumptions, thus the intersection of the sets of subsumers and subsumees must be empty. Such subsumption relations are said to be *well-formed*.

locale *wf-sub-rel* = *sub-rel* +

assumes *sub-imp-same-verts* :
  $sub \in subs \Longrightarrow fst\ (subsumee\ sub) = fst\ (subsumer\ sub)$

assumes *subsumed-by-one* :
  $\forall\ v \in subsumees\ subs.\ \exists!\ v'.\ (v,v') \in subs$

assumes *inter-empty* :
  $subsumers\ subs \cap subsumees\ subs = \{\}$

begin
  lemmas *wf-sub-rel* = *sub-imp-same-verts subsumed-by-one inter-empty*

A rephrasing of the assumption *subsumed-by-one*.

  lemma (**in** *wf-sub-rel*) *subsumed-by-two-imp* :
    assumes $(v,v1) \in subs$
    assumes $(v,v2) \in subs$
    shows $v1 = v2$
  ⟨*proof*⟩

A well-formed subsumption relation is equal to its transitive closure. We will see in the following one has to handle transitive closures of such relations.

  lemma *in-trancl-imp* :
    assumes $(v,v') \in subs^+$
    shows $(v,v') \in subs$
  ⟨*proof*⟩

  lemma *trancl-eq* :
    $subs^+ = subs$
  ⟨*proof*⟩
end

The empty subsumption relation is well-formed.

lemma
  wf-sub-rel $\{\}$
⟨*proof*⟩

9.2.2 Subsumption relation of a graph

We consider subsumption relations to equip rooted graphs. However, nothing in the previous definitions relates these relations to graphs: subsumptions relations involve objects that are of the type of indexed vertices, but that might to not be vertices of an actual graph. We equip graphs with subsumption relations using the notion of *sub-relation of a graph*. Such a relation must only involve vertices of the graph it equips.

locale *rgraph* =
  fixes *g* :: ('*v*,'*x*) *rgraph-scheme* (**structure**)

locale *sub-rel-of* = *rgraph* + *sub-rel* +
  assumes *related-are-verts* : *vertices subs* ⊆ *Graph.vertices g*
begin
  lemmas *sub-rel-of* = *related-are-verts*

The transitive closure of a sub-relation of a graph *g* is also a sub-relation of *g*.

  lemma *trancl-sub-rel-of* :
    sub-rel-of g ($subs^+$)
  ⟨*proof*⟩
end

The empty relation is a sub-relation of any graph.

lemma
  sub-rel-of g {}
⟨*proof*⟩

9.2.3 Well-formed sub-relations

We pack both previous locales into a third one. We speak about *well-formed sub-relations*.

locale *wf-sub-rel-of* = *rgraph* + *sub-rel* +
  assumes *sub-rel-of* : *sub-rel-of g subs*
  assumes *wf-sub-rel* : *wf-sub-rel subs*
begin
  lemmas *wf-sub-rel-of* = *sub-rel-of wf-sub-rel*
end

The empty relation is a well-formed sub-relation of any graph.

lemma
  wf-sub-rel-of g {}
⟨*proof*⟩

As previously, even if, in the end, we are only interested by well-formed sub-relations, we assume the relation is such only when needed.

9.3 Consistent Edge Sequences, Sub-paths

9.3.1 Consistency in presence of a subsumption relation

We model sub-paths in the same spirit than in `Graph.thy`, by starting with defining the consistency of a sequence of edges wrt. a subsumption relation. The idea is that subsumption links can "fill the gaps" between subsequent edges that would have made the sequence inconsistent otherwise. For now, we define consistency of a sequence wrt. any subsumption relation. Thus, we cannot account yet for the fact that we only consider relations without chains of subsumptions. The empty sequence is consistent wrt. to a subsumption relation from *v1* to *v2* if these two vertices are equal or if they belong to the transitive closure of the relation. A non-empty sequence is consistent if it is made of consistent sequences whose extremities are linked in the transitive closure of the subsumption relation.

fun *ces* :: $('v \times nat) \Rightarrow ('v \times nat)$ *edge list* $\Rightarrow ('v \times nat) \Rightarrow 'v$ *sub-rel-t* $\Rightarrow$ *bool*
where
ces v1 [] *v2 subs* = (*v1* = *v2* ∨ (*v1*,*v2*) ∈ $subs^+$)
| *ces v1* (*e*#*es*) *v2 subs* = ((*v1* = *src e* ∨ (*v1*,*src e*) ∈ $subs^+$) ∧ *ces* (*tgt e*) *es v2 subs*)

A consistent sequence from *v1* to *v2* without a subsumption relation is consistent between these two vertices in presence of any relation.

lemma
assumes *Graph.ces v1 es v2*
shows *ces v1 es v2 subs*
⟨*proof*⟩

Consistency in presence of the empty subsumption relation reduces to consistency as defined in `Graph.thy`.

lemma
assumes *ces v1 es v2* {}
shows *Graph.ces v1 es v2*
⟨*proof*⟩

Let (*v1*, *v2*) be an element of a subsumption relation, and *es* a sequence of edges consistent wrt. this relation from vertex *v2*. Then *es* is also consistent from *v1*. Even if this lemma will not be used much in the following, this is the base fact for saying that paths feasible from a subsumee are also feasible from its subsumer.

lemma *acas-imp-dcas* :
assumes (*v1*,*v2*) ∈ *subs*

assumes *ces v2 es v subs*
shows *ces v1 es v subs*
⟨*proof*⟩

Let *es* be a sequence of edges consistent wrt. a subsumption relation. Extending this relation preserves the consistency of *es*.

lemma *ces-Un* :
assumes *ces v1 es v2 subs1*
shows *ces v1 es v2* (*subs1* ∪ *subs2*)
⟨*proof*⟩

A rephrasing of the previous lemma.

lemma *cas-subset* :
assumes *ces v1 es v2 subs1*
assumes *subs1* ⊆ *subs2*
shows *ces v1 es v2 subs2*
⟨*proof*⟩

Simplification lemmas for *SubRel.ces*.

lemma *ces-append-one* :
ces v1 (*es* @ [*e*]) *v2 subs* = (*ces v1 es* (*src e*) *subs* ∧ *ces* (*src e*) [*e*] *v2 subs*)
⟨*proof*⟩

lemma *ces-append* :
ces v1 (*es1* @ *es2*) *v2 subs* = (∃ *v*. *ces v1 es1 v subs* ∧ *ces v es2 v2 subs*)
⟨*proof*⟩

Let *es* be a sequence of edges consistent from *v1* to *v2* wrt. a sub-relation *subs* of a graph *g*. Suppose elements of this sequence are edges of *g*. If *v1* is a vertex of *g* then *v2* is also a vertex of *g*.

lemma (**in** *sub-rel-of*) *ces-imp-ends-vertices* :
assumes *ces v1 es v2 subs*
assumes *set es* ⊆ *edges g*
assumes *v1* ∈ *Graph.vertices g*
shows *v2* ∈ *Graph.vertices g*
⟨*proof*⟩

9.3.2 Sub-paths

A sub-path leading from *v1* to *v2*, two vertices of a graph *g* equipped with a subsumption relation *subs*, is a sequence of edges consistent wrt. *subs* from *v1* to *v2* whose elements are edges of *g*. Moreover, we must assume that *subs* is a sub-relation of *g*, otherwise *es* could "exit" *g* through subsumption links.

definition *subpath* ::
$(('v \times nat),'x)$ *rgraph-scheme* $\Rightarrow ('v \times nat) \Rightarrow ('v \times nat)$ *edge list* $\Rightarrow ('v \times nat)$
$\Rightarrow (('v \times nat) \times ('v \times nat))$ *set* $\Rightarrow$ *bool*
where
subpath g v1 es v2 subs $\equiv$ *sub-rel-of g subs*
$\wedge$ *v1* $\in$ *Graph.vertices g*
$\wedge$ *ces v1 es v2 subs*
$\wedge$ *set es* $\subseteq$ *edges g*

Once again, in some cases, we will not be interested in the ending vertex of a sub-path.

abbreviation *subpath-from* ::
$(('v \times nat),'x)$ *rgraph-scheme* $\Rightarrow ('v \times nat) \Rightarrow ('v \times nat)$ *edge list* $\Rightarrow$ *'v sub-rel-t*
$\Rightarrow$ *bool*
where
subpath-from g v es subs $\equiv \exists\ v'$. *subpath g v es v' subs*

Simplification lemmas for *SubRel.subpath*.

lemma *Nil-sp* :
subpath g v1 [] *v2 subs* $\longleftrightarrow$ *sub-rel-of g subs*
$\wedge$ *v1* $\in$ *Graph.vertices g*
$\wedge$ $(v1 = v2 \vee (v1,v2) \in subs^+)$
⟨*proof*⟩

When the subsumption relation is well-formed (denoted by (*in wf-sub-rel*)), there is no need to account for the transitive closure of the relation.

lemma (**in** *wf-sub-rel*) *Nil-sp* :
subpath g v1 [] *v2 subs* $\longleftrightarrow$ *sub-rel-of g subs*
$\wedge$ *v1* $\in$ *Graph.vertices g*
$\wedge$ $(v1 = v2 \vee (v1,v2) \in subs)$
⟨*proof*⟩

Simplification lemma for the one-element sequence.

lemma *sp-one* :
shows *subpath g v1* [*e*] *v2 subs* $\longleftrightarrow$ *sub-rel-of g subs*
$\wedge$ $(v1 = src\ e \vee (v1,src\ e) \in subs^+)$
$\wedge$ $e \in edges\ g$
$\wedge$ $(tgt\ e = v2 \vee (tgt\ e,v2) \in subs^+)$
⟨*proof*⟩

Once again, when the subsumption relation is well-formed, the previous lemma can be simplified since, in this case, the transitive closure of the relation is the relation itself.

lemma (**in** *wf-sub-rel-of*) *sp-one* :
shows $subpath\ g\ v1\ [e]\ v2\ subs \longleftrightarrow sub\text{-}rel\text{-}of\ g\ subs$
$\wedge\ (v1 = src\ e \vee (v1, src\ e) \in subs)$
$\wedge\ e \in edges\ g$
$\wedge\ (tgt\ e = v2 \vee (tgt\ e, v2) \in subs)$
⟨*proof*⟩

Simplification lemma for the non-empty sequence (which might contain more than one element).

lemma *sp-Cons* :
shows $subpath\ g\ v1\ (e\ \#\ es)\ v2\ subs \longleftrightarrow sub\text{-}rel\text{-}of\ g\ subs$
$\wedge\ (v1 = src\ e\ \vee (v1, src\ e) \in subs^+)$
$\wedge\ e \in edges\ g$
$\wedge\ subpath\ g\ (tgt\ e)\ es\ v2\ subs$
⟨*proof*⟩

The same lemma when the subsumption relation is well-formed.

lemma (**in** *wf-sub-rel-of*) *sp-Cons* :
$subpath\ g\ v1\ (e\ \#\ es)\ v2\ subs \longleftrightarrow sub\text{-}rel\text{-}of\ g\ subs$
$\wedge\ (v1 = src\ e \vee (v1, src\ e) \in subs)$
$\wedge\ e \in edges\ g$
$\wedge\ subpath\ g\ (tgt\ e)\ es\ v2\ subs$
⟨*proof*⟩

Simplification lemma for *SubRel.subpath* when the sequence is known to end by a given edge.

lemma *sp-append-one* :
$subpath\ g\ v1\ (es\ @\ [e])\ v2\ subs \longleftrightarrow subpath\ g\ v1\ es\ (src\ e)\ subs$
$\wedge\ e \in edges\ g$
$\wedge\ (tgt\ e = v2 \vee (tgt\ e,\ v2) \in subs^+)$
⟨*proof*⟩

Simpler version in the case of a well-formed subsumption relation.

lemma (**in** *wf-sub-rel*) *sp-append-one* :
$subpath\ g\ v1\ (es\ @\ [e])\ v2\ subs \longleftrightarrow subpath\ g\ v1\ es\ (src\ e)\ subs$
$\wedge\ e \in edges\ g$
$\wedge\ (tgt\ e = v2 \vee (tgt\ e,\ v2) \in subs)$
⟨*proof*⟩

Simplification lemma when the sequence is known to be the concatenation of two sub-sequences.

lemma *sp-append* :
subpath g v1 (*es1* @ *es2*) *v2 subs* $\longleftrightarrow$
(∃ *v*. *subpath g v1 es1 v subs* ∧ *subpath g v es2 v2 subs*)
⟨*proof*⟩

Let *es* be a sub-path of a graph *g* starting at vertex *v1*. By definition of *SubRel.subpath*, *v1* is a vertex of *g*. Even if this is a direct consequence of the definition of *SubRel.subpath*, this lemma will ease the proofs of some goals in the following.

lemma *fst-of-sp-is-vert* :
assumes *subpath g v1 es v2 subs*
shows *v1* ∈ *Graph.vertices g*
⟨*proof*⟩

The same property (which also follows the definition of *SubRel.subpath*, but not as trivially as the previous lemma) can be established for the final vertex *v2*.

lemma *lst-of-sp-is-vert* :
assumes *subpath g v1 es v2 subs*
shows *v2* ∈ *Graph.vertices g*
⟨*proof*⟩

A sub-path ending in a subsumed vertex can be extended to the subsumer of this vertex, provided that the subsumption relation is a sub-relation of the graph it equips.

lemma *sp-append-sub* :
assumes *subpath g v1 es v2 subs*
assumes (*v2*,*v3*) ∈ *subs*
shows *subpath g v1 es v3 subs*
⟨*proof*⟩

Let *g* be a graph equipped with a well-formed sub-relation. A sub-path starting at a subsumed vertex *v1* whose set of out-edges is empty is either:

1. empty,
2. a sub-path starting at the subsumer *v2* of *v1*.

The third assumption represent the fact that, when building red-black graphs, we do not allow to build the successor of a subsumed vertex.

lemma (**in** *wf-sub-rel-of*) *sp-from-subsumee* :
assumes (*v1*,*v2*) ∈ *subs*
assumes *subpath g v1 es v subs*

assumes *out-edges g v1* = {}
shows *es* = [] ∨ *subpath g v2 es v subs*
⟨*proof*⟩

Note that it is not possible to split this lemma into two lemmas (one for each member of the disjunctive conclusion). Suppose *v* is *v1*, then *es* could be empty or it could also be a non-empty sub-path leading from *v2* to *v1*. If *v* is not *v1*, it could be *v2* and *es* could be empty or not.

A sub-path starting at a non-subsumed vertex whose set of out-edges is empty is also empty.

lemma *sp-from-de-empty* :
assumes *v1* ∉ *subsumees subs*
assumes *out-edges g v1* = {}
assumes *subpath g v1 es v2 subs*
shows *es* = []
⟨*proof*⟩

Let *e* be an edge whose target is not subsumed and has not out-going edges. A sub-path *es* containing *e* ends by *e* and this occurrence of *e* is unique along *es*.

lemma *sp-through-de-decomp* :
assumes *tgt e* ∉ *subsumees subs*
assumes *out-edges g* (*tgt e*) = {}
assumes *subpath g v1 es v2 subs*
assumes *e* ∈ *set es*
shows ∃ *es'*. *es* = *es'* @ [*e*] ∧ *e* ∉ *set es'*
⟨*proof*⟩

Consider a sub-path ending at the target of a recently added edge *e*, whose target did not belong to the graph prior to its addition. If *es* starts in another vertex than the target of *e*, then it contains *e*.

lemma (**in** *sub-rel-of*) *sp-ends-in-tgt-imp-mem* :
assumes *tgt e* ∉ *Graph.vertices g*
assumes *v* ≠ *tgt e*
assumes *subpath* (*add-edge g e*) *v es* (*tgt e*) *subs*
shows *e* ∈ *set es*
⟨*proof*⟩

end
theory *ArcExt*
imports *SubRel*
begin

10 Extending rooted graphs with edges

In this section, we formalize the operation of adding to a rooted graph an edge whose source is already a vertex of the given graph but not its target. We call this operation an extension of the given graph by adding an edge. This corresponds to an abstraction of the act of adding an edge to the red part of a red-black graph as a result of symbolic execution of the corresponding transition in the LTS under analysis, where all details about symbolic execution would have been abstracted. We then state and prove a number of facts describing the evolution of the set of paths of the given graph, first without considering subsumption links then in the case of rooted graph equipped with a subsumption relation.

10.1 Definition and Basic properties

Extending a rooted graph with an edge consists in adding to its set of edges an edge whose source is a vertex of this graph but whose target is not.

abbreviation *extends* ::
  $('v,'x)$ *rgraph-scheme* $\Rightarrow$ $'v$ *edge* $\Rightarrow$ $('v,'x)$ *rgraph-scheme* $\Rightarrow$ *bool*
where
  extends g e g' $\equiv$ *src e* $\in$ *Graph.vertices g*
    $\wedge$ *tgt e* $\notin$ *Graph.vertices g*
    $\wedge$ $g' =$ (*add-edge g e*)

After such an extension, the set of out-edges of the target of the new edge is empty.

lemma *extends-tgt-out-edges* :
  assumes *extends g e g'*
  shows *out-edges g'* (*tgt e*) $= \{\}$
⟨*proof*⟩

Consider a graph equipped with a sub-relation. This relation is also a sub-relation of any extension of this graph.

lemma (**in** *sub-rel-of*)
  assumes *extends g e g'*
  shows *sub-rel-of g' subs*
⟨*proof*⟩

Extending a graph with an edge preserves the existing sub-paths.

lemma *sp-in-extends* :
  assumes *extends g e g'*

assumes *Graph.subpath g v1 es v2*
shows *Graph.subpath g′ v1 es v2*
⟨*proof*⟩

10.2 Extending trees

We show that extending a rooted graph that is already a tree yields a new tree. Since the empty rooted graph is a tree, all graphs produced using only the extension by edge are trees.

lemma *extends-is-tree* :
assumes *is-tree g*
assumes *extends g e g′*
shows *is-tree g′*
⟨*proof*⟩

10.3 Properties of sub-paths in an extension

Extending a graph by an edge preserves the existing sub-paths.

lemma *sp-in-extends-w-subs* :
assumes *extends g a g′*
assumes *subpath g v1 es v2 subs*
shows *subpath g′ v1 es v2 subs*
⟨*proof*⟩

In an extension, the target of the new edge has no out-edges. Thus sub-paths of the extension starting and ending in old vertices are sub-paths of the graph prior to its extension.

lemma (**in** *sub-rel-of*) *sp-from-old-verts-imp-sp-in-old* :
assumes *extends g e g′*
assumes $v1 \in$ *Graph.vertices g*
assumes $v2 \in$ *Graph.vertices g*
assumes *subpath g′ v1 es v2 subs*
shows *subpath g v1 es v2 subs*
⟨*proof*⟩

For the same reason, sub-paths starting at the target of the new edge are empty.

lemma (**in** *sub-rel-of*) *sp-from-tgt-in-extends-is-Nil* :
assumes *extends g e g′*
assumes *subpath g′* (*tgt e*) *es v subs*
shows $es = []$
⟨*proof*⟩

Moreover, a sub-path *es* starting in another vertex than the target of the new edge *e* but ending in this target has *e* as last element. This occurrence of *e* is unique among *es*. The prefix of *es* preceding *e* is a sub-path leading at the source of *e* in the original graph.

lemma (**in** *sub-rel-of*) *sp-to-new-edge-tgt-imp* :
  assumes *extends g e g′*
  assumes *subpath g′ v es* (*tgt e*) *subs*
  assumes $v \neq tgt\ e$
  shows $\exists\ es'.\ es = es' @ [e] \wedge e \notin set\ es' \wedge subpath\ g\ v\ es'\ (src\ e)\ subs$
⟨*proof*⟩

end
theory *SubExt*
imports *SubRel*
begin

11 Extending subsomption relations

In this section, we are interested in the evolution of the set of sub-paths of a rooted graph equipped with a subsumption relation after adding a subsumption to this relation. We are only interested in adding subsumptions such that the resulting relation is a well-formed sub-relation of the graph (provided the original relation was such). As for the extension by edges, a number of side conditions must be met for the new subsumption to be added.

11.1 Definition

Extending a subsumption relation *subs* consists in adding a subsumption *sub* such that:

- the two vertices involved are distinct,
- they are occurrences of the same vertex,
- they are both vertices of the graph,
- the subsumee must not already be a subsumer or a subsumee,
- the subsumer must not be a subsumee (but it can already be a subsumer),

- the subsumee must have no out-edges.

Once again, in order to ease proofs, we use a predicate stating when a subsumpion relation is the extension of another instead of using a function that would produce the extension.

abbreviation *extends* ::
  $(('v \times nat),'x)$ *rgraph-scheme* $\Rightarrow$ *'v sub-rel-t* $\Rightarrow$ *'v sub-t* $\Rightarrow$ *'v sub-rel-t* $\Rightarrow$ *bool*
where
  extends g subs sub subs' $\equiv$ (
    subsumee sub $\neq$ *subsumer sub*
  $\wedge$ *fst* (*subsumee sub*) = *fst* (*subsumer sub*)
  $\wedge$ *subsumee sub* $\in$ *Graph.vertices g*
  $\wedge$ *subsumee sub* $\notin$ *subsumers subs*
  $\wedge$ *subsumee sub* $\notin$ *subsumees subs*
  $\wedge$ *subsumer sub* $\in$ *Graph.vertices g*
  $\wedge$ *subsumer sub* $\notin$ *subsumees subs*
  $\wedge$ *out-edges g* (*subsumee sub*) = {}
  $\wedge$ *subs'* = *subs* $\cup$ {*sub*})

11.2 Properties of extensions

First, we show that such extensions yield sub-relations (resp. well-formed relations), provided the original relation is a sub-relation (resp. well-formed relation).

Extending the sub-relation of a graph yields a new sub-relation for this graph.

lemma (**in** *sub-rel-of*)
  assumes *extends g subs sub subs'*
  shows *sub-rel-of g subs'*
⟨*proof*⟩

Extending a well-formed relation yields a well-formed relation.

lemma (**in** *wf-sub-rel*) *extends-imp-wf-sub-rel* :
  assumes *extends g subs sub subs'*
  shows *wf-sub-rel subs'*
⟨*proof*⟩

Thus, extending a well-formed sub-relation yields a well-formed sub-relation.

lemma (**in** *wf-sub-rel-of*) *extends-imp-wf-sub-rel-of* :
  assumes *extends g subs sub subs'*
  shows *wf-sub-rel-of g subs'*
⟨*proof*⟩

11.3 Properties of sub-paths in an extension

Extending a sub-relation of a graph preserves the existing sub-paths.

lemma *sp-in-extends* :
  assumes *extends g subs sub subs'*
  assumes *subpath g v1 es v2 subs*
  shows *subpath g v1 es v2 subs'*
⟨*proof*⟩

We want to describe how the addition of a subsumption modifies the set of sub-paths in the graph. As in the previous theories, we will focus on a small number of theorems expressing sub-paths in extensions as functions of sub-paths in the graphs before extending them (their subsumption relations). We first express sub-paths starting at the subsumee of the new subsumption, then the sub-paths starting at any other vertex.

First, we are interested in sub-paths starting at the subsumee of the new subsumption. Since such vertices have no out-edges, these sub-paths must be either empty or must be sub-paths from the subsumer of this subsumption.

lemma (**in** *wf-sub-rel-of*) *sp-in-extends-imp1* :
  assumes *extends g subs (v1,v2) subs'*
  assumes *subpath g v1 es v subs'*
  shows $es = [] \vee$ *subpath g v2 es v subs'*
⟨*proof*⟩

After an extension, sub-paths starting at any other vertex than the new subsumee are either:

- sub-paths of the graph before the extension if they do not "use" the new subsumption,
- made of a finite number of sub-paths of the graph before the extension if they use the new subsumption.

In order to state the lemmas expressing these facts, we first need to introduce the concept of *usage* of a subsumption by a sub-path.

The idea is that, if a sequence of edges that uses a subsumption *sub* is consistent wrt. a subsumption relation *subs*, then *sub* must occur in the transitive closure of *subs* i.e. the consistency of the sequence directly (and partially) depends on *sub*. In the case of well-formed subsumption relations, whose transitive closures equal the relations themselves, the dependency of the consistency reduces to the fact that *sub* is a member of *subs*.

fun *uses-sub* ::
$('v \times nat) \Rightarrow ('v \times nat)$ *edge list* $\Rightarrow ('v \times nat) \Rightarrow (('v \times nat) \times ('v \times nat)) \Rightarrow$ *bool*
where
uses-sub v1 [] *v2 sub* $= (v1 \neq v2 \wedge sub = (v1,v2))$
| *uses-sub v1* (*e#es*) *v2 sub* $= (v1 \neq src\ e \wedge sub = (v1,src\ e) \vee$ *uses-sub* (*tgt e*) *es v2 sub*)

In order for a sequence *es* using the subsumption *sub* to be consistent wrt. to a subsumption relation *subs*, the subsumption *sub* must occur in the transitive closure of *subs*.

lemma
assumes *uses-sub v1 es v2 sub*
assumes *ces v1 es v2 subs*
shows $sub \in subs^+$
⟨*proof*⟩

This reduces to the membership of *sub* to *subs* when the latter is well-formed.

lemma (**in** *wf-sub-rel*)
assumes *uses-sub v1 es v2 sub*
assumes *ces v1 es v2 subs*
shows $sub \in subs$
⟨*proof*⟩

Sub-paths prior to the extension do not use the new subsumption.

lemma *extends-and-sp-imp-not-using-sub* :
assumes *extends g subs* (v,v') *subs'*
assumes *subpath g v1 es v2 subs*
shows $\neg$ *uses-sub v1 es v2* (v,v')
⟨*proof*⟩

Suppose that the empty sequence is a sub-path leading from *v1* to *v2* after the extension. Then, the empty sequence is a sub-path leading from *v1* to *v2* in the graph before the extension if and only if (*v1*, *v2*) is not the new subsumption.

lemma (**in** *wf-sub-rel-of*) *sp-Nil-in-extends-imp* :
assumes *extends g subs* (v,v') *subs'*
assumes *subpath g v1* [] *v2 subs'*
shows *subpath g v1* [] *v2 subs* $\longleftrightarrow (v1 \neq v \vee v2 \neq v')$
⟨*proof*⟩

Thus, sub-paths after the extension that do not use the new subsumption are also sub-paths before the extension.

lemma (**in** *wf-sub-rel-of*) *sp-in-extends-not-using-sub* :
assumes *extends g subs* (v,v') *subs'*
assumes *subpath g v1 es v2 subs'*
assumes ¬ *uses-sub v1 es v2* (v,v')
shows *subpath g v1 es v2 subs*
⟨*proof*⟩

We are finally able to describe sub-paths starting at any other vertex than the new subsumee after the extension. Such sub-paths are made of a finite number of sub-paths before the extension: the usage of the new subsumption between such (sub-)sub-paths makes them sub-paths after the extension. We express this idea as follows. Sub-paths starting at any other vertex than the new subsumee are either:

- sub-paths of the graph before the extension,
- made of a non-empty prefix that is a sub-path leading to the new subsumee in the original graph and a (potentially empty) suffix that is a sub-path starting at the new subsumer after the extension.

For the second case, the lemma `sp_in_extends_imp1` as well as the following lemma could be applied to the suffix in order to decompose it into sub-paths of the graph before extension (combined with the fact that we only consider finite sub-paths, we indirectly obtain that sub-paths after the extension are made of a finite number of sub-paths before the extension, that are made consistent with the new relation by using the new subsumption).

lemma (**in** *wf-sub-rel-of*) *sp-in-extends-imp2* :
assumes *extends g subs* (v,v') *subs'*
assumes *subpath g v1 es v2 subs'*
assumes $v1 \neq v$

shows *subpath g v1 es v2 subs* ∨ (∃ *es1 es2*. *es* = *es1* @ *es2*
∧ *es1* ≠ []
∧ *subpath g v1 es1 v subs*
∧ *subpath g v es2 v2 subs'*)
(**is** *?P es v1*)
⟨*proof*⟩

end
theory *RB*
imports *LTS ArcExt SubExt*
begin

12 Red-Black Graphs

In this section we define red-black graphs and the five operators that perform over them. Then, we state and prove a number of intermediate lemmas about red-black graphs built using only these five operators, in other words: invariants about our method of transformation of red-black graphs.

Then, we define the notion of red-black paths and state and prove the main properties of our method, namely its correctness and the fact that it preserves the set of feasible paths of the program under analysis.

12.1 Basic Definitions

12.1.1 The type of Red-Black Graphs

We represent red-black graph with the following record. We detail its fields:

- *red* is the red graph, called *red part*, which represents the unfolding of the black part. Its vertices are indexed black vertices,
- *black* is the original LTS, the *black part*,
- *subs* is the subsumption relation over the vertices of *red*,
- *init-conf* is the initial configuration,
- *confs* is a function associating configurations to the vertices of *red*,
- *marked* is a function associating truth values to the vertices of *red*. We use it to represent the fact that a particular configuration (associated to a red location) is known to be unsatisfiable,
- *strengthenings* is a function associating boolean expressions over program variables to vertices of the red graph. Those boolean expressions can be seen as invariants that the configuration associated to the "strengthened" red vertex has to model.

We are only interested by red-black graphs obtained by the inductive relation *RedBlack*. From now on, we call "red-black graphs" the *pre-RedBlack*'s obtained by *RedBlack* and "pre-red-black graphs" all other ones.

record (*'vert*,*'var*,*'d*) *pre-RedBlack* =
  red :: (*'vert* × *nat*) *rgraph*
  black :: (*'vert*,*'var*,*'d*) *lts*
  subs :: *'vert sub-rel-t*

init-conf :: ('*var*,'*d*) *conf*
confs :: ('*vert* × *nat*) ⇒ ('*var*,'*d*) *conf*
marked :: ('*vert* × *nat*) ⇒ *bool*
strengthenings :: ('*vert* × *nat*) ⇒ ('*var*,'*d*) *bexp*

We call *red vertices* the set of vertices of the red graph.

abbreviation *red-vertices* ::
('*vert*,'*var*,'*d*,'*x*) *pre-RedBlack-scheme* ⇒ ('*vert* × *nat*) *set*
where
red-vertices lts ≡ *Graph.vertices* (*red lts*)

ui-edge is the operation of "unindexing" the ends of a red edge, thus giving the corresponding black edge.

abbreviation *ui-edge* ::
('*vert* × *nat*) *edge* ⇒ '*vert edge*
where
ui-edge e ≡ (| *src* = *fst* (*src e*), *tgt* = *fst* (*tgt e*) |)

We extend this idea to sequences of edges.

abbreviation *ui-es* ::
('*vert* × *nat*) *edge list* ⇒ '*vert edge list*
where
ui-es es ≡ *map ui-edge es*

12.1.2 Well-formed and finite red-black graphs

locale *pre-RedBlack* =
fixes *prb* :: ('*vert*,'*var*,'*d*) *pre-RedBlack* (**structure**)

A pre-red-black graph is well-formed if :

- its red and black parts are well-formed,
- the root of its red part is an indexed version of the root of its black part,
- all red edges are indexed versions of black edges.

locale *wf-pre-RedBlack* = *pre-RedBlack* +
assumes *red-wf* : *wf-rgraph* (*red prb*)
assumes *black-wf* : *wf-lts* (*black prb*)
assumes *consistent-roots* : *fst* (*root* (*red prb*)) = *root* (*black prb*)
assumes *ui-re-are-be* : *e* ∈ *edges* (*red prb*) ⟹ *ui-edge e* ∈ *edges* (*black prb*)
begin

lemmas *wf-pre-RedBlack* = *red-wf black-wf consistent-roots ui-re-are-be*
end

We say that a pre-red-black graph is finite if :

- the path predicate of its initial configuration contains a finite number of constraints,
- each of these constraints contains a finite number of variables,
- its black part is finite (cf. definition of *finite-lts*.).

locale *finite-RedBlack* = *pre-RedBlack* +
assumes *finite-init-pred* : *finite* (*pred* (*init-conf prb*))
assumes *finite-init-pred-symvars* : ∀ *e* ∈ *pred* (*init-conf prb*). *finite* (*Bexp.vars* *e*)
assumes *finite-lts* : *finite-lts* (*black prb*)
begin
lemmas *finite-RedBlack* = *finite-init-pred finite-init-pred-symvars finite-lts*
end

12.2 Extensions of Red-Black Graphs

We now define the five basic operations that can be performed over red-black graphs. Since we do not want to model the heuristics part of our prototype, a number of conditions must be met for each operator to apply. For example, in our prototype abstractions are performed at nodes that actually have successors, and these abstractions must be propagated to these successors in order to keep the symbolic execution graph consistent. Propagation is a complex task, and it is hard to model in Isabelle/HOL. This is partially due to the fact that we model the red part as a graph, in which propagation might not terminate. Instead, we suppose that abstraction must be performed only at leaves of the red part. This is equivalent to implicitly assume the existence of an oracle that would tell that we will need to abstract some red vertex and how to abstract it, as soon as this red vertex is added to the red part.

As in the previous theories, we use predicates instead of functions to model these transformations to ease writing and reading definitions, proofs, etc.

12.2.1 Extension by symbolic execution

The core abstract operation of symbolic execution: take a black edge and turn it red, by symbolic execution of its label. In the following abbreviation, *re* is the red edge obtained from the (hypothetical) black edge *e* that we want to symbolically execute and *c* the configuration obtained by symbolic execution of the label of *e*. Note that this extension could have been defined as a predicate that takes only two *pre-RedBlack*s and evaluates to *true* if and only if the second has been obtained by adding a red edge as a result of symbolic execution. However, making the red edge and the configuration explicit allows for lighter definitions, lemmas and proofs in the following.

abbreviation *se-extends* ::
  ('vert,'var,'d) pre-RedBlack
   ⇒ ('vert × nat) edge
   ⇒ ('var,'d) conf
   ⇒ ('vert,'var,'d) pre-RedBlack ⇒ bool
where
  se-extends prb re c prb' ≡
     ui-edge re ∈ *edges* (*black prb*)
   ∧ *ArcExt.extends* (*red prb*) *re* (*red prb'*)
   ∧ *src re* ∉ *subsumees* (*subs prb*)
   ∧ *se* (*confs prb* (*src re*)) (*labelling* (*black prb*) (*ui-edge re*)) *c*
   ∧ *prb'* = ⦇ *red* = *red prb'*,
        black = *black prb*,
        subs = *subs prb*,
        init-conf = *init-conf prb*,
        confs = (*confs prb*) (*tgt re* := *c*),
        marked = (*marked prb*)(*tgt re* := *marked prb* (*src re*)),
        strengthenings = *strengthenings prb* ⦈

Hiding the new red edge (using an existential quantifier) and the new configuration makes the following abbreviation more intuitive. However, this would require using `obtain` or `let ... = ... in ...` constructs in the following lemmas and proofs, making them harder to read and write.

abbreviation *se-extends2* ::
  ('vert,'var,'d) pre-RedBlack ⇒ ('vert,'var,'d) pre-RedBlack ⇒ bool
where
  se-extends2 prb prb' ≡
   ∃ *re* ∈ *edges* (*red prb'*).
    ui-edge re ∈ *edges* (*black prb*)
   ∧ *ArcExt.extends* (*red prb*) *re* (*red prb'*)
   ∧ *src re* ∉ *subsumees* (*subs prb*)
   ∧ *se* (*confs prb* (*src re*)) (*labelling* (*black prb*) (*ui-edge re*)) (*confs prb'* (*tgt re*))
   ∧ *black prb'* = *black prb*
   ∧ *subs prb'* = *subs prb*
   ∧ *init-conf prb'* = *init-conf prb*
   ∧ *confs prb'* = (*confs prb*) (*tgt re* := *confs prb'* (*tgt re*))

$\wedge$ *marked prb'* $=$ (*marked prb*)(*tgt re* := *marked prb* (*src re*))
$\wedge$ *strengthenings prb'* $=$ *strengthenings prb*

12.2.2 Extension by marking

The abstract operation of mark-as-unsat. It manages the information - provided, for example, by an external automated prover -, that the configuration of the red vertex *rv* has been proved unsatisfiable.

abbreviation *mark-extends* ::
('*vert*,'*var*,'*d*) *pre-RedBlack* $\Rightarrow$ ('*vert* $\times$ *nat*) $\Rightarrow$ ('*vert*,'*var*,'*d*) *pre-RedBlack* $\Rightarrow$ *bool*
where
mark-extends prb rv prb' $\equiv$
rv $\in$ *red-vertices prb*
$\wedge$ *out-edges* (*red prb*) *rv* $=$ {}
$\wedge$ *rv* $\notin$ *subsumees* (*subs prb*)
$\wedge$ *rv* $\notin$ *subsumers* (*subs prb*)
$\wedge$ $\neg$ *sat* (*confs prb rv*)
$\wedge$ *prb'* $=$ ⦇ *red* $=$ *red prb*,
black $=$ *black prb*,
subs $=$ *subs prb*,
init-conf $=$ *init-conf prb*,
confs $=$ *confs prb*,
marked $=$ (λ *rv'*. *if rv'* $=$ *rv then True else marked prb rv'*),
strengthenings $=$ *strengthenings prb*,
... $=$ *more prb* ⦈

12.2.3 Extension by subsumption

The abstract operation of introducing a subsumption link.

abbreviation *subsum-extends* ::
('*vert*,'*var*,'*d*) *pre-RedBlack* $\Rightarrow$ '*vert sub-t* $\Rightarrow$ ('*vert*,'*var*,'*d*) *pre-RedBlack* $\Rightarrow$ *bool*
where
subsum-extends prb sub prb' $\equiv$
SubExt.extends (*red prb*) (*subs prb*) *sub* (*subs prb'*)
$\wedge$ $\neg$ *marked prb* (*subsumer sub*)
$\wedge$ $\neg$ *marked prb* (*subsumee sub*)
$\wedge$ *confs prb* (*subsumee sub*) $\sqsubseteq$ *confs prb* (*subsumer sub*)
$\wedge$ *prb'* $=$ ⦇ *red* $=$ *red prb*,
black $=$ *black prb*,
subs $=$ *insert sub* (*subs prb*),
init-conf $=$ *init-conf prb*,
confs $=$ *confs prb*,

$marked = marked\ prb,$
$strengthenings = strengthenings\ prb,$
$\ldots = more\ prb\ \|)$

12.2.4 Extension by abstraction

This operation replaces the configuration of a red vertex *rv* by an abstraction of this configuration. The way the abstraction is computed is not specified. However, besides a number of side conditions, it must subsume the former configuration of *rv* and must entail its safeguard condition, if any.

abbreviation *abstract-extends* ::
$('vert,'var,'d)\ pre\text{-}RedBlack$
$\Rightarrow ('vert \times nat)$
$\Rightarrow ('var,'d)\ conf$
$\Rightarrow ('vert,'var,'d)\ pre\text{-}RedBlack$
$\Rightarrow bool$
where
$abstract\text{-}extends\ prb\ rv\ c_a\ prb' \equiv$
$rv \in red\text{-}vertices\ prb$
$\wedge \neg\ marked\ prb\ rv$
$\wedge\ out\text{-}edges\ (red\ prb)\ rv = \{\}$
$\wedge\ rv \notin subsumees\ (subs\ prb)$
$\wedge\ abstract\ (confs\ prb\ rv)\ c_a$
$\wedge\ c_a \models_c (strengthenings\ prb\ rv)$
$\wedge\ finite\ (pred\ c_a)$
$\wedge\ (\forall\ e \in pred\ c_a.\ finite\ (vars\ e))$
$\wedge\ prb' = (\!|\ red = red\ prb,$
$black = black\ prb,$
$subs = subs\ prb,$
$init\text{-}conf = init\text{-}conf\ prb,$
$confs = (confs\ prb)(rv := c_a),$
$marked = marked\ prb,$
$strengthenings = strengthenings\ prb,$
$\ldots = more\ prb\ |\!)$

12.2.5 Extension by strengthening

This operation consists in labeling a red vertex with a safeguard condition. It does not actually change the red part, but model the mechanism of preventing too crude abstractions.

abbreviation *strengthen-extends* ::
$('vert,'var,'d)\ pre\text{-}RedBlack$
$\Rightarrow ('vert \times nat)$

$\Rightarrow$ $('var,'d)$ *bexp*
$\Rightarrow$ $('vert,'var,'d)$ *pre-RedBlack*
$\Rightarrow$ *bool*
where
strengthen-extends prb rv e prb' $\equiv$
rv $\in$ *red-vertices prb*
$\wedge$ *rv* $\notin$ *subsumees* (*subs prb*)
$\wedge$ *confs prb rv* $\models_c$ *e*
$\wedge$ *prb'* = $(\!|$ *red* = *red prb*,
black = *black prb*,
subs = *subs prb*,
init-conf = *init-conf prb*,
confs = *confs prb*,
marked = *marked prb*,
strengthenings = (*strengthenings prb*)(*rv* := (λ σ. (*strengthenings prb rv*) σ $\wedge$ *e* σ)),
... = *more prb* $|\!)$

12.3 Building Red-Black Graphs using Extensions

Red-black graphs are pre-red-black graphs built with the following inductive relation, i.e. using only the five previous pre-red-black graphs transformation operators, starting from an empty red part.

inductive *RedBlack* ::
$('vert,'var,'d)$ *pre-RedBlack* $\Rightarrow$ *bool*
where
base :
fst (*root* (*red prb*)) = *init* (*black prb*) $\Longrightarrow$
edges (*red prb*) = {} $\Longrightarrow$
subs prb = {} $\Longrightarrow$
(*confs prb*) (*root* (*red prb*)) = *init-conf prb* $\Longrightarrow$
marked prb = (λ *rv*. *False*) $\Longrightarrow$
strengthenings prb = (λ *rv*. (λ σ. *True*)) $\Longrightarrow$ *RedBlack prb*

| *se-step* :
RedBlack prb $\Longrightarrow$
se-extends prb re p' prb' $\Longrightarrow$ *RedBlack prb'*

| *mark-step* :
RedBlack prb $\Longrightarrow$
mark-extends prb rv prb' $\Longrightarrow$ *RedBlack prb'*

| *subsum-step* :
RedBlack prb $\Longrightarrow$

subsum-extends prb sub prb' $\Longrightarrow$ *RedBlack prb'*

| *abstract-step* :
RedBlack prb $\Longrightarrow$
abstract-extends prb rv c_a *prb'* $\Longrightarrow$ *RedBlack prb'*

| *strengthen-step* :
RedBlack prb $\Longrightarrow$
strengthen-extends prb rv e prb' $\Longrightarrow$ *RedBlack prb'*

12.4 Properties of Red-Black-Graphs

12.4.1 Invariants of the Red-Black Graphs

The red part of a red-black graph is loop free.

lemma
assumes *RedBlack prb*
shows *loop-free* (*red prb*)
⟨*proof*⟩

A red edge can not lead to the (red) root.

lemma
assumes *RedBlack prb*
assumes *re* $\in$ *edges* (*red prb*)
shows *tgt re* $\neq$ *root* (*red prb*)
⟨*proof*⟩

Red edges are specific versions of black edges.

lemma *ui-re-is-be* :
assumes *RedBlack prb*
assumes *re* $\in$ *edges* (*red prb*)
shows *ui-edge re* $\in$ *edges* (*black prb*)
⟨*proof*⟩

The set of out-going edges from a red vertex is a subset of the set of out-going edges from the black location it represents.

lemma *red-OA-subset-black-OA* :
assumes *RedBlack prb*
shows *ui-edge* ‘ *out-edges* (*red prb*) *rv* $\subseteq$ *out-edges* (*black prb*) (*fst rv*)
⟨*proof*⟩

The red root is an indexed version of the black initial location.

lemma *consistent-roots* :

assumes *RedBlack prb*
shows *fst* (*root* (*red prb*)) = *init* (*black prb*)
⟨*proof*⟩

The red part of a red-black graph is a tree.

lemma
assumes *RedBlack prb*
shows *is-tree* (*red prb*)
⟨*proof*⟩

A red-black graph whose black part is well-formed is also well-formed.

lemma
assumes *RedBlack prb*
assumes *wf-lts* (*black prb*)
shows *wf-pre-RedBlack prb*
⟨*proof*⟩

Red locations of a red-black graph are indexed versions of its black locations.

lemma *ui-rv-is-bv* :
assumes *RedBlack prb*
assumes *rv* ∈ *red-vertices prb*
shows *fst rv* ∈ *Graph.vertices* (*black prb*)
⟨*proof*⟩

The subsumption of a red-black graph is a sub-relation of its red part.

lemma *subs-sub-rel-of* :
assumes *RedBlack prb*
shows *sub-rel-of* (*red prb*) (*subs prb*)
⟨*proof*⟩

The subsumption relation of red-black graph is well-formed.

lemma *subs-wf-sub-rel* :
assumes *RedBlack prb*
shows *wf-sub-rel* (*subs prb*)
⟨*proof*⟩

Using the two previous lemmas, we have that the subsumption relation of a red-black graph is a well-formed sub-relation of its red-part.

lemma *subs-wf-sub-rel-of* :
assumes *RedBlack prb*
shows *wf-sub-rel-of* (*red prb*) (*subs prb*)
⟨*proof*⟩

Subsumptions only involve red locations representing the same black location.

lemma *subs-to-same-BL* :
  assumes *RedBlack prb*
  assumes *sub* ∈ *subs prb*
  shows *fst* (*subsumee sub*) = *fst* (*subsumer sub*)
⟨*proof*⟩

If a red edge sequence *res* is consistent between red locations *rv1* and *rv2* with respect to the subsumption relation of a red-black graph, then its unindexed version is consistent between the black locations represented by *rv1* and *rv2*.

lemma *rces-imp-bces* :
  assumes *RedBlack prb*
  assumes *SubRel.ces rv1 res rv2* (*subs prb*)
  shows *Graph.ces* (*fst rv1*) (*ui-es res*) (*fst rv2*)
⟨*proof*⟩

The unindexed version of a subpath in the red part of a red-black graph is a subpath in its black part. This is an important fact: in the end, it helps proving that set of paths we consider in red-black graphs are paths of the original LTS. Thus, the same states are computed along these paths.

theorem *red-sp-imp-black-sp* :
  assumes *RedBlack prb*
  assumes *subpath* (*red prb*) *rv1 res rv2* (*subs prb*)
  shows *Graph.subpath* (*black prb*) (*fst rv1*) (*ui-es res*) (*fst rv2*)
⟨*proof*⟩

Any constraint in the path predicate of a configuration associated to a red location of a red-black graph contains a finite number of variables.

lemma *finite-pred-constr-symvars* :
  assumes *RedBlack prb*
  assumes *finite-RedBlack prb*
  assumes *rv* ∈ *red-vertices prb*
  shows ∀ *e* ∈ *pred* (*confs prb rv*). *finite* (*Bexp.vars e*)
⟨*proof*⟩

The path predicate of a configuration associated to a red location of a red-black graph contains a finite number of constraints.

lemma *finite-pred* :
  assumes *RedBlack prb*
  assumes *finite-RedBlack prb*

assumes *rv* ∈ *red-vertices prb*
shows *finite* (*pred* (*confs prb rv*))
⟨*proof*⟩

Hence, for a red location *rv* of a red-black graph and any label *l*, there exists a configuration that can be obtained by symbolic execution of *l* from the configuration associated to *rv*.

lemma (**in** *finite-RedBlack*) *ex-se-succ* :
assumes *RedBlack prb*
assumes *rv* ∈ *red-vertices prb*
shows ∃ *c'*. *se* (*confs prb rv*) *l c'*
⟨*proof*⟩

Generalization of the previous lemma to a list of labels.

lemma (**in** *finite-RedBlack*) *ex-se-star-succ* :
assumes *RedBlack prb*
assumes *rv* ∈ *red-vertices prb*
assumes *finite-labels ls*
shows ∃ *c'*. *se-star* (*confs prb rv*) *ls c'*
⟨*proof*⟩

Hence, for any red sub-path, there exists a configuration that can be obtained by symbolic execution of its trace from the configuration associated to its source.

lemma (**in** *finite-RedBlack*) *sp-imp-ex-se-star-succ* :
assumes *RedBlack prb*
assumes *subpath* (*red prb*) *rv1 res rv2* (*subs prb*)
shows ∃ *c*. *se-star*
(*confs prb rv1*)
(*trace* (*ui-es res*) (*labelling* (*black prb*)))
c
⟨*proof*⟩

The configuration associated to a red location *rl* is update-able.

lemma (**in** *finite-RedBlack*)
assumes *RedBlack prb*
assumes *rv* ∈ *red-vertices prb*
shows *updatable* (*confs prb rv*)
⟨*proof*⟩

The configuration associated to the first member of a subsumption is subsumed by the configuration at its second member.

lemma *sub-subsumed* :

assumes *RedBlack prb*
assumes *sub* ∈ *subs prb*
shows *confs prb* (*subsumee sub*) ⊑ *confs prb* (*subsumer sub*)
⟨*proof*⟩

12.4.2 Simplification lemmas for sub-paths of the red part.

lemma *rb-Nil-sp* :
assumes *RedBlack prb*
shows *subpath* (*red prb*) *rv1* [] *rv2* (*subs prb*) =
(*rv1* ∈ *red-vertices prb* ∧ (*rv1* = *rv2* ∨ (*rv1*,*rv2*) ∈ (*subs prb*)))
⟨*proof*⟩

lemma *rb-sp-one* :
assumes *RedBlack prb*
shows *subpath* (*red prb*) *rv1* [*re*] *rv2* (*subs prb*) =
(*sub-rel-of* (*red prb*) (*subs prb*)
∧ (*rv1* = *src re* ∨ (*rv1*, *src re*) ∈ (*subs prb*))
∧ *re* ∈ *edges* (*red prb*) ∧ (*tgt re* = *rv2* ∨ (*tgt re*, *rv2*) ∈ (*subs prb*)))
⟨*proof*⟩

lemma *rb-sp-Cons* :
assumes *RedBlack prb*
shows *subpath* (*red prb*) *rv1* (*re* # *res*) *rv2* (*subs prb*) =
(*sub-rel-of* (*red prb*) (*subs prb*)
∧ (*rv1* = *src re* ∨ (*rv1*, *src re*) ∈ *subs prb*)
∧ *re* ∈ *edges* (*red prb*)
∧ *subpath* (*red prb*) (*tgt re*) *res rv2* (*subs prb*))
⟨*proof*⟩

lemma *rb-sp-append-one* :
assumes *RedBlack prb*
shows *subpath* (*red prb*) *rv1* (*res* @ [*re*]) *rv2* (*subs prb*) =
(*subpath* (*red prb*) *rv1 res* (*src re*) (*subs prb*)
∧ *re* ∈ *edges* (*red prb*)
∧ (*tgt re* = *rv2* ∨ (*tgt re*, *rv2*) ∈ *subs prb*))
⟨*proof*⟩

12.5 Relation between red-vertices

The following key-theorem describes the relation between two red locations that are linked by a red sub-path. In a classical symbolic execution tree, the configuration at the end should be the result of symbolic execution of the trace of the sub-path from the configuration at its source. Here, due to the facts that abstractions might have occurred and that we consider sub-paths going through subsumption links, the configuration at the end subsumes the configuration one would obtain by symbolic execution of the trace. Note however that this is only true for configurations computed during the analysis: concrete execution of the sub-paths would yield the same program states than their counterparts in the original LTS.

thm *RedBlack.induct*[*of x P*]

theorem (**in** *finite-RedBlack*) *SE-rel* :
  assumes *RedBlack prb*
  assumes *subpath* (*red prb*) *rv1 res rv2* (*subs prb*)
  assumes *se-star* (*confs prb rv1*) (*trace* (*ui-es res*) (*labelling* (*black prb*))) *c*
  shows $c \sqsubseteq$ (*confs prb rv2*)
⟨*proof*⟩

12.6 Properties about marking.

A configuration which is indeed satisfiable can not be marked.

lemma *sat-not-marked* :
  assumes *RedBlack prb*
  assumes $rv \in$ *red-vertices prb*
  assumes *sat* (*confs prb rv*)
  shows ¬ *marked prb rv*
⟨*proof*⟩

On the other hand, a red-location which is marked unsat is indeed logically unsatisfiable.

lemma
  assumes *RedBlack prb*
  assumes $rv \in$ *red-vertices prb*
  assumes *marked prb rv*
  shows ¬ *sat* (*confs prb rv*)
⟨*proof*⟩

Red vertices involved in subsumptions are not marked.

lemma *subsumee-not-marked* :
  assumes *RedBlack prb*
  assumes $sub \in$ *subs prb*
  shows ¬ *marked prb* (*subsumee sub*)
⟨*proof*⟩

lemma *subsumer-not-marked* :
assumes *RedBlack prb*
assumes *sub* ∈ *subs prb*
shows ¬ *marked prb* (*subsumer sub*)
⟨*proof*⟩

If the target of a red edge is not marked, then its source is also not marked.

lemma *tgt-not-marked-imp* :
assumes *RedBlack prb*
assumes *re* ∈ *edges* (*red prb*)
assumes ¬ *marked prb* (*tgt re*)
shows ¬ *marked prb* (*src re*)
⟨*proof*⟩

Given a red subpath leading from red location *rv1* to red location *rv2*, if *rv2* is not marked, then *rv1* is also not marked (this lemma is not used).

lemma
assumes *RedBlack prb*
assumes *subpath* (*red prb*) *rv1 res rv2* (*subs prb*)
assumes ¬ *marked prb rv2*
shows ¬ *marked prb rv1*
⟨*proof*⟩

12.7 Fringe of a red-black graph

We have stated and proved a number of properties of red-black graphs. In the end, we are mainly interested in proving that the set of paths of such red-black graphs are subsets of the set of feasible paths of their black part. Before defining the set of paths of red-black graphs, we first introduce the intermediate concept of *fringe* of the red part. Intuitively, the fringe is the set of red vertices from which we can approximate more precisely the set of feasible paths of the black part. This includes red vertices that have not been subsumed yet, that are not marked and from which some black edges have not been yet symbolically executed (i.e. that have no red counterpart from these red vertices).

12.7.1 Definition

The fringe is the set of red locations from which there exist black edges that have not been followed yet.

definition *fringe* ::
  $('vert, 'var, 'd, 'x)$ *pre-RedBlack-scheme* $\Rightarrow$ $('vert \times nat)$ *set*
where
  fringe prb $\equiv$ $\{rv \in$ *red-vertices prb*.
  $rv \notin$ *subsumees* (*subs prb*) $\wedge$
  $\neg$ *marked prb rv* $\wedge$
  ui-edge ‘ *out-edges* (*red prb*) *rv* $\subset$ *out-edges* (*black prb*) (*fst rv*)$\}$

12.7.2 Fringe of an empty red-part

At the beginning of the analysis, i.e. when the red part is empty, the fringe consists of the red root.

lemma *fringe-of-empty-red1* :
  assumes *edges* (*red prb*) = {}
  assumes *subs prb* = {}
  assumes *marked prb* = (λ *rv*. *False*)
  assumes *out-edges* (*black prb*) (*fst* (*root* (*red prb*))) $\neq$ {}
  shows *fringe prb* = {*root* (*red prb*)}
⟨*proof*⟩

12.7.3 Evolution of the fringe after extension

Simplification lemmas for the fringe of the new red-black graph after adding an edge by symbolic execution. If the configuration from which symbolic execution is performed is not marked yet, and if there exists black edges going out of the target of the executed edge, the target of the new red edge enters the fringe. Moreover, if there still exist black edges that have no red counterpart yet at the source of the new edge, then its source was and stays in the fringe.

lemma *seE-fringe1* :
  assumes *sub-rel-of* (*red prb*) (*subs prb*)
  assumes *se-extends prb re c′ prb′*
  assumes $\neg$ *marked prb* (*src re*)
  assumes *ui-edge* ‘ (*out-edges* (*red prb′*) (*src re*)) $\subset$ *out-edges* (*black prb*) (*fst* (*src re*))
  assumes *out-edges* (*black prb*) (*fst* (*tgt re*)) $\neq$ {}
  shows *fringe prb′* = *fringe prb* $\cup$ {*tgt re*}
⟨*proof*⟩

On the other hand, if all possible black edges have been executed from the source of the new edge after the extension, then the source is removed from the fringe.

lemma *seE-fringe4* :
assumes *sub-rel-of* (*red prb*) (*subs prb*)
assumes *se-extends prb re c′ prb′*
assumes ¬ *marked prb* (*src re*)
assumes ¬ (*ui-edge* ‘ (*out-edges* (*red prb′*) (*src re*)) ⊂ *out-edges* (*black prb*) (*fst* (*src re*)))
assumes *out-edges* (*black prb*) (*fst* (*tgt re*)) ≠ {}
shows *fringe prb′* = *fringe prb* − {*src re*} ∪ {*tgt re*}
⟨*proof*⟩

If the source of the new edge is marked, then its target does not enter the fringe (and the source was not part of it in the first place).

lemma *seE-fringe2* :
assumes *se-extends prb re c prb′*
assumes *marked prb* (*src re*)
shows *fringe prb′* = *fringe prb*
⟨*proof*⟩

If there exists no black edges going out of the target of the new edge, then this target does not enter the fringe.

lemma *seE-fringe3* :
assumes *se-extends prb re c′ prb′*
assumes *ui-edge* ‘ (*out-edges* (*red prb′*) (*src re*)) ⊂ *out-edges* (*black prb*) (*fst* (*src re*))
assumes *out-edges* (*black prb*) (*fst* (*tgt re*)) = {}
shows *fringe prb′* = *fringe prb*
⟨*proof*⟩

Moreover, if all possible black edges have been executed from the source of the new edge after the extension, then this source is removed from the fringe.

lemma *seE-fringe5* :
assumes *se-extends prb re c′ prb′*
assumes ¬ (*ui-edge* ‘ (*out-edges* (*red prb′*) (*src re*)) ⊂ *out-edges* (*black prb*) (*fst* (*src re*)))
assumes *out-edges* (*black prb*) (*fst* (*tgt re*)) = {}
shows *fringe prb′* = *fringe prb* − {*src re*}
⟨*proof*⟩

Adding a subsumption to the subsumption relation removes the first member of the subsumption from the fringe.

lemma *subsumE-fringe* :
assumes *subsum-extends prb sub prb′*

shows *fringe prb'* = *fringe prb* − {*subsumee sub*}
⟨*proof*⟩

12.8 Red-Black Sub-Paths and Paths

The set of red-black subpaths starting in red location *rv* is the union of :

- the set of black sub-paths that have a red counterpart starting at *rv* and leading to a non-marked red location,
- the set of black sub-paths that have a prefix represented in the red part starting at *rv* and leading to an element of the fringe. Moreover, the remainings of these black sub-paths must have no non-empty counterpart in the red part. Otherwise, the set of red-black paths would simply be the set of paths of the black part.

definition *RedBlack-subpaths-from* ::
('*vert*, '*var*, '*d*, '*x*) *pre-RedBlack-scheme* ⇒ ('*vert* × *nat*) ⇒ '*vert edge list set*
where
RedBlack-subpaths-from prb rv ≡
ui-es ‘ {*res*. ∃ *rv'*. *subpath* (*red prb*) *rv res rv'* (*subs prb*) ∧ ¬ *marked prb rv'*}
∪ {*ui-es res1* @ *bes2*
| *res1 bes2*. ∃ *rv1*. *rv1* ∈ *fringe prb*
∧ *subpath* (*red prb*) *rv res1 rv1* (*subs prb*)
∧ ¬ (∃ *res21 bes22*. *bes2* = *ui-es res21* @ *bes22*
∧ *res21* ≠ []
∧ *subpath-from* (*red prb*) *rv1 res21* (*subs prb*))
∧ *Graph.subpath-from* (*black prb*) (*fst rv1*) *bes2*}

Red-black paths are red-black subpaths starting at the root of the red part.

abbreviation *RedBlack-paths* ::
('*vert*, '*var*, '*d*, '*x*) *pre-RedBlack-scheme* ⇒ '*vert edge list set*
where
RedBlack-paths prb ≡ *RedBlack-subpaths-from prb* (*root* (*red prb*))

When the red part is empty, the set of red-black subpaths starting at the red root is the set of black paths.

lemma (**in** *finite-RedBlack*) *base-RedBlack-paths* :
assumes *fst* (*root* (*red prb*)) = *init* (*black prb*)
assumes *edges* (*red prb*) = {}
assumes *subs prb* = {}
assumes *confs prb* (*root* (*red prb*)) = *init-conf prb*
assumes *marked prb* = (λ *rv*. *False*)

assumes *strengthenings prb* = (λ *rv*. (λ σ. *True*))

shows *RedBlack-paths prb* = *Graph.paths* (*black prb*)

⟨*proof*⟩

Red-black sub-paths and paths are sub-paths and paths of the black part.

lemma *RedBlack-subpaths-are-black-subpaths* :
assumes *RedBlack prb*
shows *RedBlack-subpaths-from prb rv* ⊆ *Graph.subpaths-from* (*black prb*) (*fst rv*)
⟨*proof*⟩

lemma *RedBlack-paths-are-black-paths* :
assumes *RedBlack prb*
shows *RedBlack-paths prb* ⊆ *Graph.paths* (*black prb*)
⟨*proof*⟩

12.9 Preservation of feasible paths

The following theorem states that we do not loose feasible paths using our five operators, and moreover, configurations *c* at the end of feasible red paths in some graph *prb* will have corresponding feasible red paths in successors that lead to configurations that subsume *c*. As a corollary, our calculus is correct wrt. to execution.

theorem (**in** *finite-RedBlack*) *feasible-subpaths-preserved* :
assumes *RedBlack prb*
assumes *rv* ∈ *red-vertices prb*
shows *feasible-subpaths-from* (*black prb*) (*confs prb rv*) (*fst rv*)
⊆ *RedBlack-subpaths-from prb rv*
⟨*proof*⟩

Red-black paths being red-black sub-path starting from the red root, and feasible paths being feasible sub-paths starting at the black initial location, it follows from the previous theorem that the set of feasible paths when considering the configuration of the root is a subset of the set of red-black paths.

theorem (**in** *finite-RedBlack*) *feasible-path-inclusion* :
assumes *RedBlack prb*
shows *feasible-paths* (*black prb*) (*confs prb* (*root* (*red prb*))) ⊆ *RedBlack-paths prb*
⟨*proof*⟩

The configuration at the red root might have been abstracted. In this case, the initial configuration is subsumed by the current configuration at the root. Thus the set of feasible paths when considering the initial configuration is also a subset of the set of red-black paths.

lemma *init-subsumed* :
assumes *RedBlack prb*
shows *init-conf prb* $\sqsubseteq$ *confs prb* (*root* (*red prb*))
⟨*proof*⟩

theorem (**in** *finite-RedBlack*) *feasible-path-inclusion-from-init* :
assumes *RedBlack prb*
shows *feasible-paths* (*black prb*) (*init-conf prb*) $\subseteq$ *RedBlack-paths prb*
⟨*proof*⟩

end

13 Conclusion

13.1 Related Works

Our work is inspired by Tracer [1] and the more wider class of CEGAR-like systems [2, 3, 4, 5, 6] 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 [10] in the Isabelle AFP. However, we do not use any deep graph-theory in our work; graphs are 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 NanoJava[11], Ninja[12], IMP[13], IMP^{++}[14] etc. However, these works represent the underlying abstract syntax by a free data-type 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 CFGs in HOL.

13.2 Summary

We formally proved the correctness of a set of graph transformations used by systems that compute approximations of sets of (feasible) paths by building symbolic evaluation graphs with unbounded loops. 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 abstractions or for computing interpolants or invariants can be added to the current framework, reusing the existing machinery for graphs, paths, configurations, etc.

13.3 Future Work

Currently, we are implementing in OCAML a prototype 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 mapped to implementations written by hand. This represents a substantial albeit rewarding effort that has not yet been undertaken.

References

[1] J. Jaffar, V. Murali, J. A. Navas, and A. E. Santosa, "TRACER: A Symbolic Execution Tool for Verification," in *Proceedings of CAV '12*, 2012, pp. 758–766.

[2] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The Software Model Checker Blast," *STTT*, vol. 9, no. 5-6, pp. 505–525, 2007.

[3] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "SATABS: SAT-Based Predicate Abstraction for ANSI-C," in *Proceedings of TACAS '05*, 2005, pp. 570–574.

[4] F. Ivancic, Z. Yang, M. K. Ganai, A. Gupta, I. Shlyakhter, and P. Ashar, "F-Soft: Software Verification Platform," in *Proceedings of CAV '05*, 2005, pp. 301–306.

[5] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko, "Synthesizing Software Verifiers from Proof Rules," in *Proceedings of PLDI '12*, 2012, pp. 405–416.

[6] K. L. McMillan, *Proceedings of CAV '06*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, ch. Lazy Abstraction with Interpolants, pp. 123–136.

[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] 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

[9] 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," *International Journal on Software Tools for Technology Transfer*, vol. 14, no. 1, pp. 73–93, 2011.

[10] L. Noschinski, "A Graph Library for Isabelle," *Mathematics in Computer Science*, vol. 9, no. 1, pp. 23–39, 2015. [Online]. Available: https://doi.org/10.1007/s11786-014-0183-z

[11] D. v. Oheimb and T. Nipkow, "Hoare logic for nanojava: Auxiliary variables, side effects, and virtual methods revisited," ser. LNCS. Springer-Verlag, 2002, pp. 89–105. [Online]. Available: http://dl.acm.org/citation.cfm?id=647541.730154

[12] A. Lochbihler, "Java and the java memory model - A unified, machine-checked formalisation," in *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, ser. LNCS. Springer-Verlag, 2012, pp. 497–517. [Online]. Available: https://doi.org/10.1007/978-3-642-28869-2_25

[13] T. Nipkow, "Winskel is (almost) right: Towards a mechanized semantics," *Formal Asp. Comput.*, vol. 10, no. 2, pp. 171–186, 1998. [Online]. Available: https://doi.org/10.1007/s001650050009

[14] A. D. Brucker and B. Wolff, "An Extensible Encoding of Object-oriented Data Models in HOL with an Application to IMP++," *Journal of Automated Reasoning (JAR)*, vol. Selected Papers of the AVOCS-VERIFY Workshop 2006, no. 3–4, pp. 219–249, 2008, serge Autexier, Heiko Mantel, Stephan Merz, and Tobias Nipkow (eds).