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

September 13, 2023

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 5

2 Rooted Graphs 9
   2.1 Basic Definitions and Properties 9
      2.1.1 Edges 9
      2.1.2 Rooted graphs 9
      2.1.3 Vertices 9
      2.1.4 Basic properties of rooted graphs 10
      2.1.5 Out-going edges 10
   2.2 Consistent Edge Sequences, Sub-paths and Paths 11
      2.2.1 Consistency of a sequence of edges 11
      2.2.2 Sub-paths and paths 11
   2.3 Adding Edges 14
   2.4 Trees 15

3 Arithmetic Expressions 15
   3.1 Variables and their domain 15
   3.2 Program and symbolic states 16
   3.3 The $aexp$ type-synonym 16
   3.4 Variables of an arithmetic expression 17
   3.5 Fresh variables 18

4 Boolean Expressions 18
   4.1 Basic definitions 19
      4.1.1 The $bexp$ type-synonym 19
      4.1.2 Satisfiability of an expression 19
      4.1.3 Entailment 19
      4.1.4 Conjunction 19
   4.2 Properties about the variables of an expression 20
      4.2.1 Variables of a conjunction 20
      4.2.2 Variables of an equality 21

5 Labels 22

6 Stores 23
   6.1 Basic definitions 23
      6.1.1 The $store$ type-synonym 23
      6.1.2 Symbolic variables of a store 23
      6.1.3 Fresh symbolic variables 24
## 10 Extending rooted graphs with edges

10.1 Definition and Basic properties .......................... 68
10.2 Extending trees ........................................... 69
10.3 Properties of sub-paths in an extension ................... 72

## 11 Extending subsomption relations

11.1 Definition .................................................. 74
11.2 Properties of extensions ................................... 75
11.3 Properties of sub-paths in an extension ................... 77

## 12 Red-Black Graphs

12.1 Basic Definitions .......................................... 82
   12.1.1 The type of Red-Black Graphs ....................... 82
   12.1.2 Well-formed and finite red-black graphs .......... 84
12.2 Extensions of Red-Black Graphs .......................... 84
   12.2.1 Extension by symbolic execution ................... 85
   12.2.2 Extension by marking ................................ 86
   12.2.3 Extension by subsumption ............................ 87
   12.2.4 Extension by abstraction ............................. 87
   12.2.5 Extension by strengthening ......................... 88
12.3 Building Red-Black Graphs using Extensions ............... 88
12.4 Properties of Red-Black-Graphs .......................... 89
   12.4.1 Invariants of the Red-Black Graphs .............. 89
   12.4.2 Simplification lemmas for sub-paths of the red part. 98
12.5 Relation between red-vertices ............................ 98
12.6 Properties about marking .................................. 110
12.7 Fringe of a red-black graph .............................. 115
   12.7.1 Definition ........................................... 116
   12.7.2 Fringe of an empty red-part ........................ 116
   12.7.3 Evolution of the fringe after extension .......... 116
12.8 Red-Black Sub-Paths and Paths .......................... 123
12.9 Preservation of feasible paths ............................ 127

## 13 Conclusion

13.1 Related Works ............................................ 177
13.2 Summary .................................................. 177
13.3 Future Work ............................................... 177
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 losing 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 interfere 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 contain 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.
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 :
\[ \text{xs} \neq [] = (\exists x \text{xs}'. \text{xs} = \text{xs}' @ [x]) \]
**by** (induct xs rule : rev-induct, auto)

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 =
\[\text{src} :: `'v \]
\[\text{tgt} :: `'v \]

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 =
\[\text{root} :: `'v \]
\[\text{edges} :: `'v edge set \]

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 \(('v, 'x) \text{ rgraph-scheme}\), i.e. an object that
has at least all the components of a \('v \text{ rgraph}\). 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 \(('v, 'x) \text{ rgraph-scheme}\) instead
of \('v \text{ rgraph}\) 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 \text{LTS.thy}).

\begin{definition}
  \begin{align*}
    \text{vertices} :: \Rightarrow \ 'v \text{ set} \\
    \text{where} \\
    \text{vertices} g = \{\text{root } g\} \cup \text{src} '\text{edges} g \cup \text{tgt} '\text{edges} g
  \end{align*}
\end{definition}

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 \textit{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.

\begin{abbreviation}
  \begin{align*}
    \text{loop-free} :: \Rightarrow \ 'v \text{ set} \\
    \text{where} \\
    \text{loop-free } g \equiv \forall \ e \in \text{edges} g. \text{src} e \neq \text{tgt} e
  \end{align*}
\end{abbreviation}

\begin{abbreviation}
  \begin{align*}
    \text{wf-rgraph} :: \Rightarrow \ 'v \text{ set} \\
    \text{where} \\
    \text{wf-rgraph } g \equiv \text{root } g \in \text{src} '\text{edges} g = (\text{edges} g \neq \{\})
  \end{align*}
\end{abbreviation}

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.

\begin{abbreviation}
  \begin{align*}
    \text{out-edges} :: \Rightarrow \ 'v \Rightarrow \ 'v \text{ set} \\
    \text{where} \\
    \text{out-edges} g v \equiv \{e \in \text{edges} g. \text{src} e = v\}
  \end{align*}
\end{abbreviation}
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.

```plaintext
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.

```plaintext
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 $g$.

```plaintext
lemma fst-of-sp-is-vert :
assumes subpath g v1 es v2
shows v1 ∈ vertices g
using assms by (simp add : subpath-def)
```
lemma lst-of-sp-is-vert:
  assumes subpath g v1 es v2
  shows \( v2 \in \text{vertices} \ g \)
using assms by (induction es arbitrary: v1, auto simp add: subpath-def vertices-def)

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)
by (auto simp add: vertices-def subpath-def)

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

abbreviation subpath-from ::
  ('v,'x) rgraph-scheme ⇒ 'v ⇒ 'v edge list ⇒ bool
where
  subpath-from g v es ≡ \( \exists v'. \text{subpath} \ g \ v \ es \ v' \)

abbreviation subpaths-from ::
  ('v,'x) rgraph-scheme ⇒ 'v ⇒ 'v edge list set
where
  subpaths-from g v ≡ \{ es. \text{subpath-from} \ g \ v \ es \}

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

abbreviation path ::
  ('v,'x) rgraph-scheme ⇒ 'v edge list ⇒ 'v ⇒ bool
where
  path g es v ≡ \text{subpath} \ g \ (\text{root} \ g) \ es \ v

abbreviation paths ::
  ('a,'b) rgraph-scheme ⇒ 'a edge list set
where
  paths g ≡ \{ es. \( \exists v. \text{path} \ g \ es \ v \} \)

The empty sequence is a path of any rooted graph.

lemma
  [] ∈ paths g
by (auto simp add : subpath-def vertices-def)

Some useful simplification lemmas for subpath.

**lemma** sp-one :
\[ \text{subpath } g \ v_1 \ [e] \ v_2 = (\text{src } e = v_1 \land e \in \text{edges } g \land \text{tgt } e = v_2) \]
by (auto simp add : subpath-def vertices-def)

**lemma** sp-Cons :
\[ \text{subpath } g \ v_1 \ (e \# \text{es}) \ v_2 = (\text{src } e = v_1 \land e \in \text{edges } g \land \text{subpath } g \ (\text{tgt } e) \ \text{es} \ v_2) \]
by (auto simp add : subpath-def vertices-def)

**lemma** sp-append-one :
\[ \text{subpath } g \ v_1 \ (\text{es}@[e]) \ v_2 = (\text{subpath } g \ v_1 \ \text{es} \ (\text{src } e) \land e \in \text{edges } g \land \text{tgt } e = v_2) \]
by (induct es arbitrary : v1, auto simp add : subpath-def vertices-def)

**lemma** sp-append :
\[ \text{subpath } g \ v_1 \ (\text{es}_1@[e] \text{es}_2) \ v_2 = (\exists v. \text{subpath } g \ v_1 \ \text{es}_1 \ v \land \text{subpath } g \ v \ \text{es}_2 \ v_2) \]
by (induct es1 arbitrary : v1)

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
using assms
by (induct es arbitrary : v)

A sub-path leads to a unique vertex.

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'
using assms(2,3)

proof (induction es arbitrary : v1)
case Nil thus ?case by simp
next
case (Cons e' es)

hence e = e' ∨ (e ≠ e' ∧ e ∈ set es) by auto
thus ?case

proof (elim disjE, goal-cases)
case 1 thus ?case
using assms(1) Cons
by (rule-tac ?x=[| in exI) (cases es, auto simp add: sp-Cons)
next
case 2 thus ?case
using assms(1) Cons(1)[of tgt e'] Cons(2)
by (auto simp add : sp-Cons)
qed
qed

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 ⇒ 'v edge ⇒ ('v, 'x) rgraph-scheme
where
add-edge g e ≡ rgraph.edges-update (λ edges. edges ∪ {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 ∉ vertices g
assumes v ≠ tgt e
assumes subpath (add-edge g e) v es (tgt e)
shows e ∈ set es

proof –
  have es ≠ [] using assms(2,3) by (auto simp add : subpath-def)
then obtain e' es' where es = es' @ [e'] by (simp add : neqNilConv2) blast
thus \( ? \)thesis using \texttt{assms(1,3) by (auto simp add : sp-append-one vertices-def image-def)}
qed

2.4 Trees

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

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

The empty graph is thus a tree.

lemma \( \text{empty-graph-is-tree} \):
assumes \( \text{edges} \) \( g = \{ \} \)
shows \( \text{is-tree} \) \( g \)
using \texttt{assms by (auto simp add : is-tree-def subpath-def vertices-def)}
end

theory \( Aexp \)
imports \( \text{Main} \)
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**

\[ 'v \text{ symvar} = 'v \times \text{nat} \]

**Lemma**

\[ \neg \text{finite (UNIV::'v symvar set)} \]

**By (simp add: finite-prod)**

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 \( 'v \text{ symvar} \) 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**

\[ ('v, 'd) \text{ state} = 'v \Rightarrow 'd \]

### 3.3 The \texttt{aexp} type-synonym

Arithmetic (and boolean, see \texttt{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 ⇒ ′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 $\sigma$. 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 ⇒ ′v set
where
vars e = \{v. ∃ σ val. e (σ(v := val)) ≠ e σ\}

lemma vars-example-1 :
  fixes e::(′v,integer) aexp
  assumes e = (λ σ. σ x - σ y)
  assumes x ≠ y
  shows vars e = \{x,y\}
unfolding set-eq-iff
proof (intro allI iffI)
  fix v assume v ∈ vars e
  then obtain σ val
  where e (σ(v := val)) ≠ e σ
  unfolding vars-def by blast
thus $v \in \{x, y\}$
using assms by (case-tac $v = x$, simp, (case-tac $v = y$, simp+))
next
fix $v$ assume $v \in \{x, y\}$

thus $v \in \text{vars } e$
using assms
by (auto simp add: vars-def)
(rule-tac $?x=\lambda v. 0\ in \ exI$, rule-tac $?x=1\ in \ exI$, simp)+
qed

lemma vars-example-2 :
fixes $e::(\tau,v,\text{integer})\ aexp$
assumes $e = (\lambda \sigma. \sigma x - \sigma y)$
assumes $x = y$
shows $\text{vars } e = \{}$
using assms by (auto simp add: vars-def)

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 ⇒ ('v,'d) aexp ⇒ bool
where
fresh $v\ e ≡ v \notin \text{vars } e$

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.

\textbf{type-synonym} ('v,'d) bexp = ('v,'d) state ⇒ bool

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

\textbf{abbreviation} fresh ::
'v ⇒ ('v,'d) bexp ⇒ bool
\textbf{where}
fresh v e ≡ v \notin vars e

4.1.2 Satisfiability of an expression

A boolean expression \( e \) is satisfiable if there exists a state \( \sigma \) such that \( e \sigma \) is true.

\textbf{definition} sat ::
('v,'d) bexp ⇒ bool
\textbf{where}
sat e = (\exists \sigma. e \sigma)

4.1.3 Entailment

A boolean expression \( \varphi \) entails another boolean expression \( \psi \) if all states
making \( \varphi \) true also make \( \psi \) true.

\textbf{definition} entails ::
('v,'d) bexp ⇒ ('v,'d) bexp ⇒ bool (infixl \( \models_B \ 55 \))
\textbf{where}
\( \varphi \models_B \psi \equiv (\forall \sigma. \varphi \sigma \rightarrow \psi \sigma) \)

4.1.4 Conjunction

In the following, path predicates are represented by sets of boolean expres-
sions. We define the conjunction of a set of boolean expressions \( E \) as the
expression that associates \textit{true} to a state \( \sigma \) if, for all elements \( e \) of \( E \), \( e \) associates \textit{true} to \( \sigma \).

\textbf{definition} \textit{conjunct} ::
\[ (',v',d) \text{ bexp set} \Rightarrow (',v',d) \text{ bexp} \]
\textbf{where}
\[ \text{conjunct} \ E \equiv (\lambda \sigma. \forall \ e \in E. \ e \sigma) \]

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.

\textbf{lemma} \textit{vars-of-conj} :
\[ \text{vars} (\lambda \sigma. \ e1 \sigma \land e2 \sigma) \subseteq \text{vars} \ e1 \cup \text{vars} \ e2 \]
\textbf{(is \ vars \ ?e \subseteq \ vars \ e1 \cup \ vars \ e2)}
\textbf{unfolding} \textit{subset-iff}
\textbf{proof} (\textit{intro allI impI})
\textbf{fix} \( v \) \textbf{assume} \( v \in \text{vars} \ ?e \)
then obtain $\sigma \, \text{val}$
where $?e (\sigma (v := \text{val})) \neq ?e \, \sigma$
unfolding $\text{vars-def}$ by blast

hence $e1 (\sigma (v := \text{val})) \neq e1 \, \sigma \lor e2 (\sigma (v := \text{val})) \neq e2 \, \sigma$
by auto

thus $v \in \text{vars } e1 \cup \text{vars } e2$ unfolding $\text{vars-def}$ by blast
qed

lemma finite-conj :
assumes finite $E$
assumes $\forall \, e \in E. \text{finite } (\text{vars } e)$
shows $\text{finite } (\text{vars } (\text{conjunct } E))$
using assms
proof (induct rule : finite-induct, goal-cases)
case 1 thus $\text{?case}$ by (simp add : $\text{vars-def conjunct-def}$)
next
case $\langle 2 \, e \, E \rangle$

thus $\text{?case}$
using $\text{vars-of-conj[of } e \text{ conjunct } E]$ 
by (rule-tac rev-$\text{finite-subset}$, auto simp add : $\text{conjunct-def}$)
qed

4.2.2 Variables of an equality

We proceed analogously for the equality of two arithmetic expressions.

lemma vars-of-eq-a :
shows $\text{vars } (\lambda \, \sigma. \, e1 \, \sigma \, = \, e2 \, \sigma) \subseteq \text{Aexp.vars } e1 \cup \text{Aexp.vars } e2$
(is $\text{vars } ?e \subseteq \text{Aexp.vars } e1 \cup \text{Aexp.vars } e2$)
unfolding $\text{subset-iff}$
proof (intro allI impI)

fix $v$ assume $v \in \text{vars } ?e$

then obtain $\sigma \, \text{val}$ where $?e (\sigma (v := \text{val})) \neq ?e \, \sigma$
unfolding $\text{vars-def}$ by blast

hence $e1 (\sigma (v := \text{val})) \neq e1 \, \sigma \lor e2 (\sigma (v := \text{val})) \neq e2 \, \sigma$
by auto

thus $v \in \text{Aexp.vars } e1 \cup \text{Aexp.vars } e2$
unfolding \( Aexp.vars-def \) by blast
qed

lemma finite-vars-of-a-eq :
assumes finite \( (Aexp.vars e1) \)
assumes finite \( (Aexp.vars e2) \)
shows finite \( (vars (\lambda \sigma. e1 \sigma = e2 \sigma)) \)
using assms vars-of-eq-a[of e1 e2] by (rule-tac rev-finite-subset, auto)

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 \texttt{jump}, \texttt{break}, etc) are labelled by a \texttt{Skip}.

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

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

definition finite-label ::
(\( ('v,'d) label \)) \Rightarrow \textit{bool}
where
finite-label l ≡ case l of
| Assume e ⇒ finite (Bexp.vars e)
| Assign e ⇒ finite (Aexp.vars e)
| - ⇒ True

abbreviation finite-labels ::
(\( ('v,'d) label list \)) \Rightarrow \textit{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 ⇒ 'a store ⇒ 'a symvar
where
symvar v s ≡ (v, s v)

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

lemma
inj (λ v. symvar v s)
by (auto simp add : inj-on-def symvar-def)

The sets of symbolic variables of a store is the image set of the function symvar.
definition symvars ::
    'a store ⇒ 'a symvar set
where
symvars s = (λ 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 ⇒ 'v store ⇒ bool
where
fresh-symvar sv s = (sv ∉ 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 ⇒ ('v symvar, 'd) state ⇒ 'v store ⇒ bool
where
consistent σ σ_{sym} s ≡ (∀ v. σ_{sym} (symvar v s) = σ v)

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

lemma
  ∃ σ σ_{sym}. consistent σ σ_{sym} s
by (auto simp add : consistent-def)

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 σ σ_{sym} s = (∀ sv ∈ symvars s. σ_{sym} sv = σ (fst sv))
by (auto simp add : consistent-def symvars-def symvar-def)

lemma consistent-eq2 :
consistent σ σ_{sym} store = (σ = (λ v. σ_{sym} (symvar v store)))
by (auto simp add : consistent-def)
lemma consistentI1 : 
consistent σ (λ sv. σ (fst sv)) store
using consistent-eq1 by fast

lemma consistentI2 : 
consistent (λ v. σ_sym (symvar v store)) σ_sym store
using consistent-eq2 by fast

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 ⇒ ′v store ⇒ (′v symvar,′d) aexp
where
adapt-aexp e s = (λ σ_sym. e (λ v. σ_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
using assms by (simp add : consistent-eq2 adapt-aexp-def)

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 $v \in \text{Aexp.vars } e$
  shows $\text{symvar } v \in \text{Aexp vars } (\text{adapt-aexp } e \ s) \ (\text{is } \ ?sv \in \text{Aexp vars } \ ?e')$
proof
  obtain $\sigma \ \text{val where } e \ (\sigma (v := \text{val})) \neq e \ \sigma$
  using assms unfolding Aexp.vars-def by blast
moreover
have $(\lambda v. ((\lambda sv. \sigma (fst sv))(?sv := \text{val}) \ (\text{symvar } \text{va } s)) = (\sigma (v := \text{val}))$
  by (auto simp add: symvar-def)
ultimately
show $\ ?thesis$
unfolding Aexp.vars-def mem-Collect-eq
using consistentI1[of $\sigma \ s$]
consistentI2[of $(\lambda sv. \sigma (fst sv))(?sv := \text{val}) \ s$]
by (rule-tac $?x=\lambda sv. \sigma (fst sv)$ in exI, rule-tac $?x=\text{val}$ in exI)
(simp add: adapt-aexp-is-subst)
qed

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:
  $\text{Aexp.vars } (\text{adapt-aexp } e \ s) \subseteq \text{symvars } s \ (\text{is } \text{Aexp vars } \ ?e' \subseteq \text{symvars } s)$
unfolding subset-iff
proof (intro allI impI)
fix sv

assume $sv \in \text{Aexp vars } \ ?e'$
then obtain $\sigma_{sym} \ \text{val}$
where $?e' \ (\sigma_{sym} (sv := \text{val})) \neq ?e' \ \sigma_{sym}$
by (simp add: Aexp.vars-def, blast)

hence $(\lambda x. (\sigma_{sym} (sv := \text{val})) \ (\text{symvar } x \ s)) \neq (\lambda x. \sigma_{sym} (\text{symvar } x \ s))$
proof (intro notI)
assume $(\lambda x. (\sigma_{sym} (sv := \text{val})) \ (\text{symvar } x \ s)) = (\lambda x. \sigma_{sym} (\text{symvar } x \ s))$

hence $?e' (\sigma_{sym} (sv := \text{val})) = ?e' \ \sigma_{sym}$
by (simp add: adapt-aexp-def)
thus $False$
using $\langle ?e' (\sigma_{sym} (sv := val)) \neq ?e' \sigma_{sym}\rangle$
by (elim notE)
qed

then obtain $v$
where $(\sigma_{sym} (sv := val)) (\text{symvar } v s) \neq \sigma_{sym} (\text{symvar } v s)$
by blast

hence $sv = \text{symvar } v s$ by (case-tac $sv = \text{symvar } v s$, simp-all)

thus $sv \in \text{symvars } s$ by (simp add : symvars-def)
qed

lemma $\text{symvar-var-imp-var}$:
assumes $sv \in Aexp.\text{vars} (\text{adapt-aexp } e s)$ (is $sv \in Aexp.\text{vars } ?e'$)
sows $\text{fst } sv \in Aexp.\text{vars } e$
proof −
obtain $v$ where $sv = (v, sv)$
using assms(1) symvars-of-adapt-aexp
unfolding symvars-def symvar-def
by blast

obtain $\sigma_{sym} \text{val}$ where $?e' (\sigma_{sym} (sv := val)) \neq ?e' \sigma_{sym}$
using assms unfolding Aexp.vars-def by blast

moreover
have $(\lambda v. (\sigma_{sym} (sv := val)) (\text{symvar } v s)) = (\lambda v. \sigma_{sym} (\text{symvar } v s)) (v := val)$
using $(sv = (v, sv))$ by (auto simp add : symvar-def)

ultimately
show $\text{thesis}$
using $(sv = (v, sv))$

consistentI2[of $\sigma_{sym} s$]
consistentI2[of $\sigma_{sym} (sv := val) s$]
unfolding Aexp.vars-def
by (simp add : adapt-aexp-is-subst) blast
qed

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:
\[ Aexp.vars (\text{adapt-aexp } e \ s) = (\lambda \ v. \ \text{symvar } v \ s) \ ' Aexp.vars e \]
unfolding set-eq-iff image-def mem-Collect-eq Bex-def

proof (intro allI iffI, goal-cases)
  case (1 sv)

  moreover
  have sv = symvar (fst sv) s
  using 1 symvars-of-adapt-aexp
  by (force simp add: symvar-def symvars-def)

  ultimately
  show ?case using symvar-var-imp-var by blast

next
  case (2 sv) thus ?case using var-imp-symvar-var by fast

qed

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))
unfolding adapt-aexp-vars using assms by auto

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 ::
\[ ('n,'d) bexp \Rightarrow 'v store \Rightarrow ('v symvar,'d) bexp \]
where
\[ adapt-bexp e s = (\lambda \ \sigma. \ e (\lambda x. \ \sigma (\text{symvar } x s))) \]

lemma adapt-bexp-is-subst:
assumes consistent s \sigma s
shows (adapt-bexp e s) \sigma_{sym} = e \sigma
using assms by (simp add: consistent-eq2 adapt-bexp-def)

lemma var-imp-symvar-var2:
assumes \( v \in Bexp.vars \ e \)
shows \( \text{symvar } v \ s \in Bexp.vars (\text{adapt-bexp } e \ s) \) (is \( ?sv \in Bexp.vars \ ?v' \))

proof
  obtain \( \sigma \ \text{val} \) where \( A : e (\sigma (v := \text{val})) \neq e \ \sigma \)
using assms unfolding Bexp.vars-def by blast

moreover
have \((\lambda va. ((\lambda sv. \sigma (fst sv))(?sv := val)) (symvar va s)) = (\sigma(v := val))\)
by (auto simp add : symvar-def)

ultimately
show \(?thesis
unfolding Bexp.vars-def mem-Collect-eq
using consistentI1[of \(\sigma\) s]
consistentI2[of (\(\lambda sv. \sigma (fst sv))(?sv:= val)) s]
by (rule-tac \(?x=\lambda sv. \sigma (fst sv)\) in exI, rule-tac \(?x=val\) in exI)
(simp add : adapt-bexp-is-subst)
qed

lemma symvars-of-adapt-bexp :
\(Bexp.v\)ars (adapt-bexp e s) \(\subseteq\) symvars s \((is\ Bexp.v\)ars \(?e' \subseteq ?SV)\)
proof
fix \(sv\)
assume \(sv \in Bexp.v\)ars \(?e'\)
then obtain \(\sigma_{sym}\) val
where \(?e' (\sigma_{sym} (sv := val)) \neq ?e' \sigma_{sym}\)
by (simp add : Bexp.v\)ars-def, blast)

hence \((\lambda x. (\sigma_{sym} (sv := val)) (symvar x s)) \neq (\lambda x. \sigma_{sym} (symvar x s))\)
by (auto simp add : adapt-bexp-def)

hence \(\exists v. (\sigma_{sym} (sv := val)) (symvar v s) \neq \sigma_{sym} (symvar v s)\) by force
then obtain \(v\)
where \((\sigma_{sym} (sv := val)) (symvar v s) \neq \sigma_{sym} (symvar v s)\)
by blast

hence \(sv = symvar v s\) by (case-tac \(sv = symvar v s, simp-all)\)

thus \(sv \in\) symvars s by (simp add : symvars-def)
qed

lemma symvar-var-imp-var2 :
assumes \(sv \in Bexp.v\)ars (adapt-bexp e s) \((is\ sv \in Bexp.v\)ars \(?e'\))
shows \(fst sv \in Bexp.v\)ars e
proof –
obtain $v$ where $sv = (v, s v)$
using assms symvars-of-adapt-bexp
unfolding symvars-def symvar-def
by blast

obtain $\sigma_{sym}$ $val$ where $?e' (\sigma_{sym} (sv := val)) \neq ?e' \sigma_{sym}$
using assms unfolding vars-def by blast

moreover
have $(\lambda v. (\sigma_{sym} (sv := val)) (\text{symvar } v s)) = (\lambda v. \sigma_{sym} (\text{symvar } v s)) (v := val)$
using $(sv = (v, s v))$ by (auto simp add : symvar-def)

ultimately
show $?thesis$
using $(sv = (v, s v))$
  consistentI2[of $\sigma_{sym}$ s]
  unfolding consistentI2[of $\sigma_{sym}$ (sv := val) s]
unfolding vars-def by (simp add : adapt-bexp-is-subst) blast
qed

lemma adapt-bexp-vars :
  $\text{Bexp.vars (adapt-bexp e s)} = (\lambda v. \text{symvar } v s) \ ' \text{Bexp.vars } e$
(is $\text{Bexp.vars } ?e' = ?R$)
unfolding set-eq-iff image-def mem-Collect-eq Bex-def
proof (intro allI iffI, goal-cases)
case (1 sv)
hence $\text{fst } sv \in \text{vars } e$ by (rule symvar-var-imp-var2)

moreover
have $sv = \text{symvar (fst } sv) s$
using 1 symvars-of-adapt-bexp
by (force simp add: symvar-def symvars-def)

ultimately
show $?case$ by blast
next
case (2 sv)
then obtain $v$ where $v \in \text{vars } e$ $sv = \text{symvar } v s$ by blast
thus $?case$ using var-imp-symvar-var2 by simp
qed
lemma finite-vars-imp-finite-adapt-b:
  assumes finite (Bexp.vars e)
  shows finite (Bexp.vars (adapt-bexp e s))
unfolding adapt-bexp-vars using assms by auto

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 =
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 ::

\[
\text{\('v \text{ symvar, 'd) bexp set\)}
\]

**where**

\[
symvars c = \text{Store.symvars (store c)} \cup \text{Bexp.vars (conjunct (pred c))}
\]

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 ::

\[
\text{\('v \text{ symvar \Rightarrow ('v, 'd) conf \Rightarrow bool}\)}
\]

**where**

\[
\text{fresh-symvar sv c = (sv \notin symvars c)}
\]

7.1.4 Satisfiability

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

**abbreviation** sat ::

\[
\text{\('v, 'd) conf \Rightarrow bool}\)
\]

**where**

\[
sat c \equiv \text{Bexp.sat (conjunct (pred c))}
\]

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 ::

\[
\text{\('v, 'd) conf \Rightarrow ('v, 'd) state set}\)
\]

**where**

\[
\text{states c} = \{\sigma. \exists \sigma_{sym}. \text{consistent } \sigma \sigma_{sym} \text{ (store c) \land conjunct (pred c) } \sigma_{sym}\}
\]

A configuration is satisfiable if and only if its set of states is not empty.
lemma sat-eq :
  sat c = (states c ≠ {})
using consistentI2 by (simp add : sat-def states-def) fast

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 ⇒ (‘v,’d) conf ⇒ bool (infixl ⊑ 55)
where
  $c_2 ⊑ c_1$ ≡ (states $c_2$ ⊆ states $c_1$)

The subsumption relation is reflexive and transitive.

lemma subsums-refl :
  $c ⊑ c$
by (simp only : subsums-def)

lemma subsums-trans :
  $c_1 ⊑ c_2$ ⇒ $c_2 ⊑ c_3$ ⇒ $c_1 ⊑ c_3$
unfolding subsums-def by simp

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 $c_1 ⊑ c_2$
  assumes $c_2 ⊑ c_1$
  shows states $c_1$ = states $c_2$
using assms by (simp add : subsums-def)

A satisfiable configuration can only be subsumed by satisfiable configurations.

lemma sat-sub-by-sat :
  assumes sat $c_2$
  and $c_2 ⊑ c_1$
  shows sat $c_1$
using assms sat-eq[of $c_1$] sat-eq[of $c_2$]
by (simp add : subsums-def) fast

On the other hand, an unsatisfiable configuration can only subsume unsatisfiable configurations.
lemma unsat-subs-unsat :
  assumes ¬ sat c1
  assumes c2 ⊑ c1
  shows ¬ sat c2
using assms sat-eq[of c1] sat-eq[of c2]
by (simp add : subsums-def)

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 ⇒ (′v,d) bexp
where
  sem c = (λ σ. σ ∈ states c)

theorem
  c2 ⊑ c1 ⇐⇒ sem c2 |=B sem c1
unfolding subsums-def sem-def subset-iff entails-def by (rule refl)

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 ⇒ (′v,d) conf ⇒ bool
where
  abstract c c_a ≡ c ⊑ 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 ⇒ ('v,'d) bexp ⇒ bool (infixl |=_c 55)

where
    c |=_c φ ≡ sem c |=_B φ

lemma
    sem c |=_B e ←→ (∀ σ ∈ states c. e σ)
by (auto simp add : states-def sem-def entails-def)

theory SymExec
imports Conf Labels
begin

7.2 Symbolic Execution

We model symbolic execution by an inductive predicate \textit{se} which takes two configurations \(c_1\) and \(c_2\) and a label \(l\) and evaluates to \textit{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 \textit{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 \textit{se} (and \textit{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 \textit{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 \textit{se} and \textit{se-star}

Symbolic execution of \textit{Skip} does not change the configuration from which it is performed.

When the label is of the form \textit{Assume} \(e\), the adaptation of \(e\) to the store is added to the \textit{pred} component.

In the case of an assignment, the \textit{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 \textit{pred} component.

35
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.

\[
\text{inductive } se ::
\]
\[
(\langle v, d \rangle \text{ conf } \Rightarrow (\langle v, d \rangle \text{ label } \Rightarrow (\langle v, d \rangle \text{ conf } \Rightarrow \text{ bool }))
\]
where
\[
se c \text{ Skip } c
\]
\[
\mid se c (\text{ Assume } e) \ (\{ \text{ store } = \text{ store } c, \text{ pred } = \text{ pred } c \cup \{ \text{ adapt-bexp } e (\text{ store } c) \} \} \ |
\]
\[
\mid \text{ fst sv } = v \quad \implies \\
\quad \text{ fresh-symvar } sv c \implies \\
\quad se c (\text{ Assign } v e) \ (\{ \text{ store } = (\text{ store } c)(v := \text{ snd sv}), \text{ pred } = \text{ pred } c \cup \{ (\lambda \sigma. \sigma \text{ sv } = (\text{ adapt-aexp } e (\text{ store } c)) \sigma) \} \} \ |
\]

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

\[
\text{inductive } se-star :: (\langle v, d \rangle \text{ conf } \Rightarrow (\langle v, d \rangle \text{ label list } \Rightarrow (\langle v, d \rangle \text{ conf } \Rightarrow \text{ bool }))
\]
where
\[
\text{ se-star } c \ [] c
\]
\[
\mid se c1 l c2 \implies se-star c2 ls c3 \implies se-star c1 (l \# ls) c3
\]

7.2.2 Basic properties of \( se \)

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

\[
\text{lemma } se-sat-imp-sat :
\]
\[
\text{ assumes } se c l c'
\]
\[
\text{ assumes } sat c'
\]
\[
\text{ shows } sat c
\]
\[
\text{ using } assms \text{ by } \text{ cases (auto simp add : sat-def conjunct-def)}
\]

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

\[
\text{lemma } unsat-imp-se-unsat :
\]
\[
\text{ assumes } se c l c'
\]
\[
\text{ assumes } \neg \text{ sat c}
\]
\[
\text{ shows } \neg \text{ sat c'}
\]
\[
\text{ using } assms \text{ by } \text{ cases (simp add : sat-def conjunct-def)}+
\]

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 \( c \).

\[
\text{lemma } [\text{ simp}]
\]

36
se c Skip c' = (c' = c)
by (simp add : se.simps)

lemma se-Assume-eq :
se c (Assume e) c' = (c' = ⟨ store = store c, pred = pred c ∪ {adapt-bexp e (store c)} ⟩)
by (simp add : se.simps)

lemma se-Assign-eq :
se c (Assign v e) c' =
(∃ sv. fresh-symvar sv c
∧ fst sv = v
∧ c' = ⟨ store = (store c)(v := snd sv),
pred = insert (λσ. σ sv = adapt-aexp e (store c) σ) (pred c) ⟩)
by (simp only : se.simps, blast)

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') =
(λσ. conjunct (pred c) σ ∧ adapt-bexp e (store c) σ)
using assms se-Assume-eq[of c e c']
by (auto simp add : conjunct-def)

lemma path-pred-of-se-Assign :
assumes se c (Assign v e) c'
shows ∃ sv. conjunct (pred c') =
(λσ. conjunct (pred c) σ ∧ σ sv = adapt-aexp e (store c) σ)
using assms se-Assign-eq[of c v e c']
by (fastforce simp add : conjunct-def)

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' = {σ ∈ states c. e σ}
using assms se-Assume-eq[of c e c']
by (auto simp add : adapt-bexp-is-subst states-def conjunct-def)

Let $c$ and $c'$ be two configurations such that $c'$ is obtained from $c$ by symbolic execution of a label of the form $\text{Assign } v e$. 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 $\sigma'$ is a state of $c'$, then it has been obtained from an adequate update of a state $\sigma$ of $c$.

**lemma** states-of-se-assign1 :

**assumes** se c (Assign v e) c'

**assumes** $\sigma' \in \text{states } c$

**shows** $\exists \sigma \in \text{states } c. \sigma' = (\sigma (v := e \sigma))$

**proof**

- obtain $\sigma_{\text{sym}}$
  - where 1 : consistent $\sigma' \sigma_{\text{sym}} (\text{store } c')$
  - and 2 : conjunct (pred c') $\sigma_{\text{sym}}$
  - using assms(2) unfolding states-def by blast

then obtain $\sigma$

where 3 : consistent $\sigma \sigma_{\text{sym}} (\text{store } c)$

using consistentI2 by blast

moreover

have conjunct (pred c) $\sigma_{\text{sym}}$

using assms(1) 2 by (auto simp add : se-Assign-eq conjunct-def)

ultimately

have $\sigma \in \text{states } c$ by (simp add : states-def) blast

moreover

have $\sigma' = \sigma (v := e \sigma)$

**proof**

- have $\sigma' v = e \sigma$
  - **proof**
    - have $\sigma' v = \sigma_{\text{sym}} (\text{symvar } v (\text{store } c'))$
      - using 1 by (simp add : consistent-def)

moreover

have $\sigma_{\text{sym}} (\text{symvar } v (\text{store } c')) = (\text{adapt-aexp } e (\text{store } c)) \sigma_{\text{sym}}$

using assms(1) 2 se-Assign-eq[of c v e c']

by (force simp add : symvar-def conjunct-def)

moreover

have $(\text{adapt-aexp } e (\text{store } c)) \sigma_{\text{sym}} = e \sigma$

38
using 3 by (rule adapt-acexp-is-subst)

ultimately
show ?thesis by simp
qed

moreover
have ∀ x. x ≠ v → σ' x = σ x
proof (intro allI impI)
fix x

assume x ≠ v

moreover
hence σ' x = σ_sym (symvar x (store c))
using assms(1)
proof (unfolding consistent-def symvar-def)
by (drule-tac ?x=x in spec)
(auto simp add : se-Assign-eq)

moreover
have σ_sym (symvar x (store c)) = σ x
using 3 by (auto simp add : consistent-def)

ultimately
show σ' x = σ x by simp
qed

ultimately
show ?thesis by auto
qed

ultimately
show ?thesis by (simp add : states-def) blast
qed

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 ∃ σ ∈ states c. σ' = σ (v := e σ)
shows σ' ∈ states c'
proof –
obtain σ
where σ ∈ states c
and σ' = σ (v := e σ)
using assms(2) by blast
then obtain $\sigma_{\text{sym}}$
where 1: consistent $\sigma \sigma_{\text{sym}} (\text{store } c)$
and 2: conjunct $\text{pred } c) \sigma_{\text{sym}}$
unfolding states-def by blast

obtain sv
where 3: fresh-symvar sv c
and 4: $\text{fst } sv = v$
and 5: $c' = (\text{store } (c)(v := \text{snd } sv),
\text{pred } = \text{insert } (\lambda \sigma. \sigma sv = \text{adapt-aexp } e (\text{store } c) \sigma) (\text{pred } c) \Sigma$
using assms(1) se-Assign-eq[of $c \text{ v } c'\] by blast

define $\sigma_{\text{sym}}'$ where $\sigma_{\text{sym}}' = \sigma_{\text{sym}} (sv := e \sigma)$

have consistent $\sigma' \sigma_{\text{sym}}' (\text{store } c')$
using $\sigma' = \sigma (v := e \sigma), 1 4 5$
by (auto simp add : symvar-def consistent-def $\sigma_{\text{sym}}'-\text{def}$)

moreover
have conjunct $\text{pred } c' \sigma_{\text{sym}}'$
proof –
have conjunct $\text{pred } c \sigma_{\text{sym}}'$
using 2 3 by (simp add : fresh-symvar-def symvars-def Bexp.vars-def $\sigma_{\text{sym}}'-\text{def}$)

moreover
have $\sigma_{\text{sym}}' \text{ sv = (adapt-aexp } e (\text{store } c)) \sigma_{\text{sym}}'$
proof –
have Aexp.fresh sv (adapt-aexp e (store c))
using 3 symvars-of-adapt-aexp[of e store c]
by (auto simp add : fresh-symvar-def symvars-def)

thus $\text{thesis}$
using adapt-aexp-is-subst[of 1, of e]
by (simp add : Aexp.vars-def $\sigma_{\text{sym}}'-\text{def}$)
qed

ultimately
show $\text{thesis}$ using 5 by (simp add : conjunct-def)
qed

ultimately
show $\text{thesis}$ unfolding states-def by blast
qed

40
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** 
\( \text{states-of-se-assign} \) :
- **Assumes** \( \text{se } c (\text{Assign } v e) \ c' \)
- **Shows** \( \text{states } c' = \{ \sigma (v := e \sigma) \mid \sigma. \sigma \in \text{states } c \} \)
- **Using** \( \text{assms } \text{states-of-se-assign1 } \text{states-of-se-assign2} \) **by** **fast**

### 7.2.3 Monotonicity of \( \text{se} \)

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

**Theorem** \( \text{se-mono-for-sub} \) :
- **Assumes** \( \text{se } c l c1' \)
- **Assumes** \( \text{se } c2 l c2' \)
- **Assumes** \( c2 \sqsubseteq c1 \)
- **Shows** \( c2' \sqsubseteq c1' \)
- **Using** \( \text{assms} \)
- **By** ((cases \( l \)),
  - (simp add : ),
  - (simp add : states-of-se-assume subsums-def, blast),
  - (simp add : states-of-se-assign subsums-def, blast))

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

**Theorem** \( \text{se-mono-for-states-eq} \) :
- **Assumes** \( \text{states } c1 = \text{states } c2 \)
- **Assumes** \( \text{se } c l c1' \)
- **Assumes** \( \text{se } c2 l c2' \)
- **Shows** \( \text{states } c2' = \text{states } c1' \)
- **Using** \( \text{assms}(1) \)
  - \( \text{se-mono-for-sub}[\text{OF } \text{assms}(2,3)] \)
  - \( \text{se-mono-for-sub}[\text{OF } \text{assms}(3,2)] \)
- **By** \( \text{simp add : subsums-def} \)

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** \( \text{se-succs-states} \) :
- **Assumes** \( \text{se } c l c1 \)
- **Assumes** \( \text{se } c l c2 \)
- **Shows** \( \text{states } c1 = \text{states } c2 \)
- **Using** \( \text{assms } \text{se-mono-for-states-eq} \) **by** **fast**
7.2.4 Basic properties of \textit{se\_star}

Some simplification lemmas for \textit{se\_star}.

\textbf{lemma} \texttt{[simp]} :
\[
\text{se\_star } c \llbracket \; c' = (c' = c) \\
\text{by (subst se\_star.simps) auto}
\]

\textbf{lemma} \textit{se\_star-Cons} :
\[
\text{se\_star } c1 \ (l \# ls ) c2 = (\exists \ c. \ se \ c1 \ l \ c \land \ se\_star \ c \ ls \ c2) \\
\text{by (subst (1) se\_star.simps) blast}
\]

\textbf{lemma} \textit{se\_star-one} :
\[
\text{se\_star } c1 \ [l] \ c2 = se \ c1 \ l \ c2 \\
\text{using se\_star-Cons by force}
\]

\textbf{lemma} \textit{se\_star-append} :
\[
\text{se\_star } c1 \ (ls1 \@ ls2) \ c2 = (\exists \ c. \ se\_star \ c1 \ ls1 \ c \land \ se\_star \ c \ ls2 \ c2) \\
\text{by (induct ls1 arbitrary : c1, simp-all add : se\_star-Cons) blast}
\]

\textbf{lemma} \textit{se\_star-append-one} :
\[
\text{se\_star } c1 \ (ls \@ [l]) \ c2 = (\exists \ c. \ se\_star \ c1 \ ls \ c \land \ se \ c \ l \ c2) \\
\text{unfolding se\_star-append se\_star-one by (rule refl)}
\]

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

\textbf{lemma} \textit{unsat-imp-se\_star-unsat} :
\[
\text{assumes se\_star } c \ ls \ c' \\
\text{assumes } \neg \ sat \ c \\
\text{shows } \neg \ sat \ c' \\
\text{using assms} \\
\text{by (induct ls arbitrary : c)} \\
\quad (simp, force simp add : se\_star-Cons unsat-imp-se-unsat)
\]

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

\textbf{lemma} \textit{se\_star-sat-imp-sat} :
\[
\text{assumes se\_star } c \ ls \ c' \\
\text{assumes sat } c' \\
\text{shows sat } c \\
\text{using assms}
\]
by (induct ls arbitrary : c)
  (simp, force simp add : se-Star-Cons se-sat-imp-sat)

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 ⊑ c1
shows c2′ ⊑ c1′

using assms
by (induct ls arbitrary : c1 c2)
  (auto simp add : se-star-Cons se-mono-for-sub)

**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′

using assms(1)
  se-star-mono-for-sub[OF assms(2,3)]
  se-star-mono-for-sub[OF assms(3,2)]
by (simp add : subsums-def)

**Lemma se\_star-succs-states**:

assumes se\_star c ls c1
assumes se\_star c ls c2
shows states c1 = states c2

using assms se-star-mono-for-states-eq by fast

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** ::
\((\forall v \cdot d)\) \text{conf} \Rightarrow \text{bool}

\text{where}
\begin{align*}
\text{updatable} \; c & \equiv \forall \; v \cdot \exists \; sv. \; \text{fst} \; sv = v \land \text{fresh-symvar} \; sv \; c
\end{align*}

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

\textbf{lemma} \text{updatable-imp-ex-se-suc}:
\begin{align*}
\text{assumes} & \, \text{updatable} \; c \\
\text{shows} & \, \exists \; c', \; \text{se} \; c \; l \; c'
\end{align*}
\textbf{using} \text{assms} \\
\textbf{by} \text{(cases l, simp-all add : se-Assert-eq se-Assign-eq updatable-def)}

A sufficient condition for a configuration to be updatable is that its path predicate has a finite number of variables. The \text{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 \(SV\).

\textbf{lemma} \text{finite-symvars-imp-ex-greatest-symvar}:
\begin{align*}
\text{fixes} & \; SV \; :: \; 'a \; \text{symvar set} \\
\text{assumes} & \, \text{finite} \; SV \\
\text{assumes} & \, \exists \; sv \in \; SV. \; \text{fst} \; sv = v \\
\text{shows} & \, \exists \; sv \in \{sv \in SV. \; \text{fst} \; sv = v\}. \\
& \quad \forall \; sv' \in \{sv \in SV. \; \text{fst} \; sv = v\}. \; \text{snd} \; sv' \leq \text{snd} \; sv
\end{align*}
\textbf{proof} –
\begin{align*}
\text{have} & \, \text{finite} \; (\text{snd} \; \{sv \in SV. \; \text{fst} \; sv = v\}) \\
\text{and} & \; \{sv \in SV. \; \text{fst} \; sv = v\} \neq \{\}
\end{align*}
\textbf{using} \text{assms} \text{by} \text{auto}

\textbf{moreover}
\textbf{have} \; \forall \; (E::nat \; \text{set}), \; \text{finite} \; E \land E \neq \{\} \; \rightarrow \; (\exists \; n \in \; E. \; \forall \; m \in \; E. \; m \leq n)
\textbf{by} \text{(intro allI impI, induct-tac rule : finite-ne-induct)}
\textbf{(simp+, force)}

\textbf{ultimately}
obtain $n$
where $n \in \operatorname{snd}' \{ sv \in SV. \text{fst}~sv = v \}$
and $\forall~m \in \operatorname{snd}' \{ sv \in SV. \text{fst}~sv = v \}. m \leq n$
by blast

moreover
then obtain $sv$
where $sv \in \{ sv \in SV. \text{fst}~sv = v \}$ and $\text{snd}~sv = n$
by blast

ultimately
show $\text{thesis}$ by blast
qed

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 ($\text{Bexp} . \text{vars} (\text{conjunct} (\text{pred} c))$) (is finite ?V)
shows updatable $c$
unfolding updatable-def
proof (intro allI)
fix $v$

show $\exists sv. \text{fst}~sv = v \land \text{fresh-symvar}~sv~c$
proof (case-tac $\exists sv \in ?V. \text{fst}~sv = v$, goal-cases)
case 1

then obtain max-sv
where $\text{max-sv} \in ?V$
and $\text{fst}~\text{max-sv} = v$
and $\text{max} : \forall sv' \in \{ sv \in ?V. \text{fst}~sv = v \}. \text{snd}~sv' \leq \text{snd}~\text{max-sv}$
by blast

show $\text{thesis}$
using max
unfolding fresh-symvar-def symvars-def Store.symvars-def symvar-def
proof (case-tac snd max-sv $\leq$ store $c$ $v$, goal-cases)
case 1 thus $?\text{case}$ by (rule-tac $?x = (v, \text{Suc} (\text{store} c v))$ in $\text{exI}$) auto
next
case 2 thus $?\text{case}$ by (rule-tac $?x = (v, \text{Suc} (\text{snd} max-sv))$ in $\text{exI}$) auto
The path predicate of a configuration whose \( \text{pred} \) 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 \texttt{finite-imp-ex-se-succ} :
\begin{align*}
\text{assumes} & \quad \text{finite (\text{pred} c)} \\
\text{assumes} & \quad \forall \, e \in \text{pred} \, c. \, \text{finite (Bexp.vars e)} \\
\text{shows} & \quad \exists \, c'. \, \text{se} \, c \, l \, c'
\end{align*}
using \texttt{finite-pred-imp-se-updatable[OF finite-conj[OF assms(1,2)]]}
by (rule updatable-imp-ex-se-suc)

For symbolic execution not to block \textit{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 \( \text{pred} \) component and of the sets of variables of its elements. Thus, successors \( \text{se} \) 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 \( \text{pred} \) 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 \( \text{pred} \) component.

lemma \texttt{se-preserves-finiteness1} :
\begin{align*}
\text{assumes} & \quad \text{finite-label l} \\
\text{assumes} & \quad \text{se} \, c \, l \, c' \\
\text{assumes} & \quad \forall \, e \in \text{pred} \, c. \, \text{finite (Bexp.vars e)}
\end{align*}
shows \( \forall e \in \text{pred} \ c', \text{finite} (B\text{exp}.\text{vars} \ e) \)
proof (cases \( l \))
case Skip then \( \text{thesis} \) using \( \text{assms} \) by (simp add : )
next
case (Assume \( e \)) then \( \text{thesis} \) using \( \text{assms} \) finite-vars-imp-finite-adapt-b by (auto simp add : se-Assume-eq finite-label-def)
next
case (Assign \( v \ e \))
then obtain \( sv \) where fresh-synvar \( sv \ c \) and \( \text{fst} \ sv = v \) and \( c' = (\LoadIdentity \ \text{store} = (\text{store} \ c)(v := \text{snd} \ sv), \text{pred} = \text{insert} (\lambda \sigma. \sigma \ sv = \text{adapt-aexp} \ e (\text{store} \ c) \sigma) (\text{pred} c) ) \)
using \( \text{assms}(2) \) se-Assign-eq[of \( c \ v \ e \ c' \)] by blast
moreover
have finite \((B\text{exp}.\text{vars} (\lambda \sigma. \sigma \ sv = \text{adapt-aexp} \ e (\text{store} \ c) \sigma))\)
proof –
have finite \((A\text{exp}.\text{vars} (\lambda \sigma. \sigma \ sv))\)
by (auto simp add : Aexp.vars-def)
moreover
have finite \((A\text{exp}.\text{vars} (\text{adapt-aexp} \ e (\text{store} \ c)))\)
using \( \text{assms}(1) \) Assign finite-vars-imp-finite-adapt-a
by (auto simp add : finite-label-def)
ultimately
show \( \text{thesis} \) using finite-vars-of-a-eq by auto
qed
ultimately
show \( \text{thesis} \) using \( \text{assms} \) by auto
qed

lemma \( \text{se-preserves-finiteness2} \) :
assumes \( \text{se} \ c \ l \ c' \)
assumes finite \((\text{pred} \ c)\)
shows finite \((\text{pred} \ c')\)
using \( \text{assms} \)
by (cases \( l \) )
(auto simp add : se-Assume-eq se-Assign-eq)
We are now ready to prove that a sufficient condition for symbolic execution not to block along a sequence of labels is that the \textit{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.

\textbf{lemma} finite-imp-ex-se-succ :
\begin{itemize}
  \item \textbf{assumes} finite (pred \(c\))
  \item \textbf{assumes} \(\forall e \in \text{pred } c. \text{finite (Bexp.vars } e)\)
  \item \textbf{assumes} finite-labels \(ls\)
  \item \textbf{shows} \(\exists c'. \text{se-star } c \text{ ls } c'\)
\end{itemize}
\textbf{using} \textit{assms}
\textbf{proof} (\textit{induct ls arbitrary : c, goal-cases})
\begin{itemize}
  \item \textbf{case} 1 \textbf{show} \(\text{?case using } \text{se-star.simps by blast}\)
  \item \textbf{next}
    \begin{itemize}
      \item \textbf{case} (2 \(l \text{ ls } c\))
        \begin{itemize}
          \item \textbf{then obtain } \(c1 \text{ where } \text{se } c \text{ l } c1 \text{ using } \text{finite-imp-ex-se-succ by blast}\)
        \end{itemize}
    \end{itemize}
  \item \textbf{hence} finite (pred \(c1\))
  \item \textbf{and} \(\forall e \in \text{pred } c1. \text{finite (Bexp.vars } e)\)
  \item \textbf{using} 2 se-preserves-finiteness1 se-preserves-finiteness2 by \textit{fastforce+}
  \item \textbf{moreover}
    \textbf{have} finite-labels \(ls\) \textbf{using} 2 by \textit{simp}
  \item \textbf{ultimately}
    \textbf{obtain} \(c2 \text{ where } \text{se-star } c1 \text{ ls } c2 \text{ using } 2 \text{ by blast}\)
  \item \textbf{thus} \textbf{?case using } (se \(c \text{ l } c1\)) \textbf{using } se-star-Cons \textbf{by blast}\)
\end{itemize}
\textbf{qed}

\textbf{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 \(c\).

\textbf{definition} feasible :: \(('v,'d) \text{ conf } \Rightarrow ('v,'d) \text{ label list } \Rightarrow \text{bool where}
\begin{align*}
\text{feasible } c \text{ ls } &\equiv \exists c'. \text{se-star } c \text{ ls } c' \land \text{sat } c'
\end{align*}

A simplification lemma for the case where \(ls\) is not empty.

\textbf{lemma} feasible-Cons :
\begin{align*}
\text{feasible } c \text{ (l#ls) } &\equiv \exists c'. \text{se } c \text{ l } c' \land \text{sat } c' \land \text{feasible } c' \text{ ls}
\end{align*}
\textbf{proof} (\textit{intro iffI, goal-cases})
The following theorem is very important for the rest of this formalization. It states that, given two configurations $c_1$ and $c_2$ such that $c_1$ subsums $c_2$, then any feasible sequence of labels from $c_2$ is also feasible from $c_1$. 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 $c_1$)
- assumes finite (pred $c_2$)
- assumes $\forall e \in$ pred $c_1$. finite (Bexp.vars $e$)
- assumes $\forall e \in$ pred $c_2$. finite (Bexp.vars $e$)
- assumes $c_2 \subseteq c_1$
- assumes feasible $c_2$ ls
- shows feasible $c_1$ ls

**proof** (induct ls arbitrary : $c_1$ $c_2$)

- case Nil thus $\_case$ by (simp add : feasible-def sat-sub-by-sat)

**next**

- case (Cons $l$ ls $c_1$ $c_2$)

  then obtain $c_2'$ where $se$ $c_2$ $l$ $c_2'$
  and $sat$ $c_2'$
  and feasible $c_2'$ ls
  using feasible-Cons by blast

  obtain $c_1'$ where $se$ $c_1$ $l$ $c_1'$
  using finite-conj[OF Cons(3,5)]
  finite-pred-imp-se-updatable
  updatable-imp-ex-se-suc
  by blast

  moreover
  hence $sat$ $c_1'$
using se-mono-for-sub[\texttt{OF - se c2 l c2'}} Cons(7)]

\texttt{sat-sub-by-sat}[\texttt{OF} \langle \texttt{sat c2'}\rangle]

by \texttt{fast}

moreover

have \texttt{feasible c1'} \texttt{ls}

proof —

have \texttt{finite-label l}

and \texttt{finite-labels ls using Cons(2) by simp-all}

have \texttt{finite (pred c1')}

by (rule se-preserves-finiteness2[\texttt{OF} \langle \texttt{se c1 l c1'}} Cons(3)])

moreover

have \texttt{finite (pred c2')}

by (rule se-preserves-finiteness2[\texttt{OF} \langle \texttt{se c2 l c2'}} Cons(4)])

moreover

have \(\forall e \in \text{pred c1'}. \text{finite (Bexp.vars e)\) by (rule se-preserves-finiteness1[\texttt{OF} \langle \texttt{finite-label l} \langle \texttt{se c1 l c1'}} Cons(5)])

moreover

have \(\forall e \in \text{pred c2'}. \text{finite (Bexp.vars e)\) by (rule se-preserves-finiteness1[\texttt{OF} \langle \texttt{finite-label l} \langle \texttt{se c2 l c2'}} Cons(6)])

moreover

have \texttt{c2' \subseteq c1'}

by (rule se-mono-for-sub[\texttt{OF} \langle \texttt{se c1 l c1'}} \langle \texttt{se c2 l c2'}} Cons(7)])

ultimately

show \texttt{thesis using Cons(1) \langle \texttt{feasible c2'} \texttt{ls} \langle \texttt{finite-labels ls} \rangle by fast}

qed

ultimately

show \texttt{case} by (\texttt{auto simp add : feasible-Cons})

qed

7.4 Concrete execution

We illustrate our notion of symbolic execution by relating it with \textit{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.

\textbf{inductive } ce ::
\((‘v,’d)\) state \(\Rightarrow\) \((‘v,’d)\) label \(\Rightarrow\) \((‘v,’d)\) state \(\Rightarrow\) bool
\textbf{where}
\(ce \sigma \text{ Skip} \sigma\)
| \(e \sigma \Rightarrow ce \sigma (\text{Assume} e) \sigma\)
| \(ce \sigma (\text{Assign} v e) (\sigma(v := e \sigma))\)

\textbf{inductive } ce-star :: \((‘v,’d)\) state \(\Rightarrow\) \((‘v,’d)\) label list \(\Rightarrow\) \((‘v,’d)\) state \(\Rightarrow\) bool
\textbf{where}
\(ce-star c [] c\)
| \(ce c1 l c2 \Rightarrow ce-star c2 ls c3 \Rightarrow ce-star c1 (l \# ls) c3\)

\textbf{lemma } [simp]:
\(ce \sigma \text{ Skip} \sigma’ = (\sigma’ = \sigma)\)
\textbf{by (auto simp add : ce.simps)}

\textbf{lemma } [simp]:
\(ce \sigma (\text{Assume} e) \sigma’ = (\sigma’ = \sigma \land e \sigma)\)
\textbf{by (auto simp add : ce.simps)}

\textbf{lemma } [simp]:
\(ce \sigma (\text{Assign} v e) \sigma’ = (\sigma’ = \sigma(v := e \sigma))\)
\textbf{by (auto simp add : ce.simps)}

\textbf{lemma } se-as-ce :
\textbf{assumes} se c l c’
\textbf{shows} states c’ = \{\sigma’: \exists \sigma \in states c. ce \sigma l \sigma’\}
\textbf{using assms}
\textbf{by (cases l)}
\textbf{(auto simp add: states-of-se-assume states-of-se-assign)}

\textbf{lemma } [simp]:
\(ce-star \sigma [] \sigma’ = (\sigma’ = \sigma)\)
\textbf{by (subst ce-star.simps) simp}

\textbf{lemma } ce-star-Cons :
\(ce-star \sigma 1 (l \# ls) \sigma 2 = (\exists \sigma. ce \sigma 1 l \sigma \land ce-star \sigma ls \sigma 2)\)
\textbf{by (subst (1) ce-star.simps) blast}
lemma se-star-as-ce-star :
  assumes se-star c ls c'
  shows \( \text{states } c' = \{ \sigma' . \exists \sigma \in \text{states } c . \text{ce-star } \sigma \text{ ls } \sigma' \} \)
  using assms
proof (induct ls arbitrary : c)
  case Nil thus \(?\)case by simp
next
  case (Cons l ls c)

  then obtain c'' where se c l c''
  \(\text{and } \text{se-star } c'' \text{ ls } c'\)
  using se-star-Cons by blast

  show \(?\)case
  unfolding set-eq_iff Bex_def mem_Collect_eq
  proof (intro allI iffI, goal-cases)
    case (1 \(\sigma'\))

    then obtain \(\sigma''\) where \(\sigma'' \in \text{states } c''\)
    \(\text{and } \text{ce-star } \sigma'' \text{ ls } \sigma'\)
    using Cons(1) \(\text{se-star } c'' \text{ ls } c'\) by blast

    moreover
    then obtain \(\sigma\) where \(\sigma \in \text{states } c\)
    \(\text{and } \text{ce } \sigma \text{ l } \sigma''\)
    using \(\text{se } c \text{ l } c''\), se-as-ce by blast

    ultimately
    show \(?\)case by (simp add: ce-star-Cons) blast
  next
  case (2 \(\sigma'\))

  then obtain \(\sigma\) where \(\sigma \in \text{states } c\)
  \(\text{and } \text{ce-star } \sigma \text{ (l#ls) } \sigma'\)
  by blast

  moreover
  then obtain \(\sigma''\) where \(\text{ce } \sigma \text{ l } \sigma''\)
  \(\text{and } \text{ce-star } \sigma'' \text{ ls } \sigma'\)
  using ce-star-Cons by blast

  ultimately
  show \(?\)case
  using Cons(1) \(\text{se-star } c'' \text{ ls } c'\), \(\text{se } c \text{ l } c''\) by (auto simp add: se-as-ce)
qed
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 ⇒ ('var,'d) label

We call initial location the root of the underlying graph.

abbreviation init ::
   ('vert,'var,'d,'x) lts-scheme ⇒ '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

\( trace as L \equiv map L as \)

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

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

\textbf{abbreviation graph ::}

\( ('\text{vert},'\text{var},'d,'x) \text{lts-scheme} \Rightarrow '\text{vert} \text{rgraph} \)

where

\( \text{graph} \text{lts} \equiv \text{rgraph.truncate lts} \)

An LTS is well-formed if its underlying \textit{rgraph} is well-formed.

\textbf{abbreviation wf-lts ::}

\( ('\text{vert},'\text{var},'d,'x) \text{lts-scheme} \Rightarrow bool \)

where

\( \text{wf-lts} \text{lts} \equiv \text{wf-rgraph (graph lts)} \)

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

\textbf{abbreviation finite-lts ::}

\( ('\text{vert},'\text{var},'d,'x) \text{lts-scheme} \Rightarrow bool \)

where

\( \text{finite-lts} \text{lts} \equiv \forall \ l \in \text{range (labelling lts)}. \text{finite-label} \ l \)

8.2 Feasible sub-paths and paths

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

\textbf{abbreviation feasible-subpath ::}

\( ('\text{vert},'\text{var},'d,'x) \text{lts-scheme} \Rightarrow ('\text{var},'d) \text{conf} \Rightarrow '\text{vert} \Rightarrow '\text{vert} \text{edge list} \Rightarrow '\text{vert} \Rightarrow bool \)

where

\( \text{feasible-subpath} \text{lts pc l1 as l2} \equiv \text{Graph.subpath lts l1 as l2} \)

\&\text{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.

\textbf{abbreviation feasible-subpath-from ::}
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.

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

A subsumption relation is a set of subsumptions.

\textbf{type-synonym} \ 'v sub-rel-t = \,'v sub-t set

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

\textbf{abbreviation} \ subsumee :: \\
\textit{\,'v sub-t} \Rightarrow \textit{(\,'v \times \text{nat})}

\textbf{where} \ \\
\textit{subsumee sub} \equiv \text{fst sub}

\textbf{abbreviation} \ subsumer :: \\
\textit{\,'v sub-t} \Rightarrow \textit{(\,'v \times \text{nat})}

\textbf{where} \ \\
\textit{subsumer sub} \equiv \text{snd sub}

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

\textbf{abbreviation} \ subsumees :: \\
\textit{\,'v sub-rel-t} \Rightarrow \textit{(\,'v \times \text{nat}) set}

\textbf{where} \ \\
\textit{subsumees subs} \equiv \text{subsumee ' subs}

\textbf{abbreviation} \ subsumers :: \\
\textit{\,'v sub-rel-t} \Rightarrow \textit{(\,'v \times \text{nat}) set}

\textbf{where} \ \\
\textit{subsumers subs} \equiv \text{subsumer ' subs}

The two following lemmas will prove useful in the following.

\textbf{lemma} \ subsumees-conv : \\
\textit{subsumees subs} = \{ \textit{v}. \ \exists \textit{v'}. (\textit{v,v'}) \in \text{subs} \}

\textbf{by force}
lemma subsumers-conv :
subsumers subs = {v’ . ∃ v. (v,v’) ∈ subs}
by force

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

abbreviation vertices ::
’v sub-rel-t ⇒ (’v × nat) set
where
vertices subs ≡ subsumers subs ∪ 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 ∈ subs ⇒ fst (subsumee sub) = fst (subsumer sub)

assumes subsumed-by-one :
∀ v ∈ subsumees subs. ∃! v’. (v,v’) ∈ subs

assumes inter-empty :
subsumers subs ∩ subsumees subs = {}
lemmaswf-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) ∈ subs
   assumes (v,v2) ∈ subs
   shows v1 = v2
   using assms wf-sub-rel unfolding subsumees-conv by blast

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') ∈ subs+
   shows (v,v') ∈ subs
   by force

lemma trancl-eq :
   subs+ = subs
   using in-trancl-imp r-into-trancl[of - subs] by fast
end

The empty subsumption relation is well-formed.

lemma
   wf-sub-rel {}
   by (auto simp add : wf-sub-rel-def)

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 :
  \( \text{sub-rel-of } g \ (\text{subs}^+) \)
using tranclD[of - - subs] tranclD2[of - - subs] sub-rel-of
unfolding sub-rel-of-def subsumers-conv subsumees-conv by blast
end

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

lemma
  \( \text{sub-rel-of } g \ \{\} \)
by (auto simp add : sub-rel-of-def)

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
  \( \text{wf-sub-rel-of } g \ \{\} \)
by (auto simp add : sub-rel-of-def wf-sub-rel-def wf-sub-rel-of-def)

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 \( v_1 \) to \( v_2 \) 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.

\[
\text{fun } \text{ces} :: (v 	imes \text{nat}) \Rightarrow (v 	imes \text{nat}) \Rightarrow (v 	imes \text{sub-rel-t}) \Rightarrow \text{bool}
\]

where

\[
\begin{align*}
\text{ces } v_1 \ & [] \ v_2 \ 
\text{subs} = (v_1 = v_2 \ \vee \ (v_1,v_2) \in \text{subs}^+) \\
| \ 
\text{ces } v_1 \ (e\#es) \ v_2 \ 
\text{subs} = ((v_1 = \text{src } e \ \vee \ (v_1,\text{src } e) \in \text{subs}^+) \ \wedge \ \text{ces } (\text{tgt } e) \ es \ v_2 \ 
\text{subs})
\end{align*}
\]

A consistent sequence from \( v_1 \) to \( v_2 \) without a subsumption relation is consistent between these two vertices in presence of any relation.

lemma

\begin{align*}
\text{assumes } & \text{Graph}.\text{ces } v_1 \ es \ v_2 \\
\text{shows } & \text{ces } v_1 \ es \ v_2 \ \text{subs} \\
\text{using } & \text{assms by } (\text{induct } es \ \text{arbitrary} : v_1 , \text{auto})
\end{align*}

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

lemma

\begin{align*}
\text{assumes } & \text{ces } v_1 \ es \ v_2 \ \{} \\
\text{shows } & \text{ces } v_1 \ es \ v_2 \\
\text{using } & \text{assms by } (\text{induct } es \ \text{arbitrary} : v_1 , \text{auto})
\end{align*}

Let \((v_1, v_2)\) be an element of a subsumption relation, and \( es \) a sequence of edges consistent wrt. this relation from vertex \( v_2 \). Then \( es \) is also consistent from \( v_1 \). Even if this lemma will not be used much in the following, this is the base fact for saying that paths feasible from a subsume are also feasible from its subsumer.

lemma acas-imp-dcas :

\begin{align*}
\text{assumes } & (v_1,v_2) \in \text{subs} \\
\text{assumes } & \text{ces } v_2 \ es \ v \ \text{subs} \\
\text{shows } & \text{ces } v_1 \ es \ v \ \text{subs} \\
\text{using } & \text{assms by } (\text{cases es, simp-all} \ (\text{intro disjI2, force})+)
\end{align*}

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

lemma ces-Un :

\begin{align*}
\text{assumes } & \text{ces } v_1 \ es \ v_2 \ \text{subs}
\end{align*}
shows \( ces \, v1 \, es \, v2 \, (subs1 \cup subs2) \)
using \(\text{assms}\) by \((\text{induct es arbitrary : v1, auto simp add : trancl-mono})\)

A rephrasing of the previous lemma.

**lemma cas-subset :**

assumes \( ces \, v1 \, es \, v2 \, subs1 \)
assumes \( subs1 \subseteq subs2 \)
shows \( ces \, v1 \, es \, v2 \, subs2 \)
using \(\text{assms}\) by \((\text{induct es arbitrary : v1, auto simp add : trancl-mono})\)

Simplification lemmas for \(\text{SubRel.ces}\).

**lemma ces-append-one :**

\( ces \, v1 \, (es \# e) \, v2 \, subs = (ces \, v1 \, es \, (src \, e) \, subs \wedge ces \, (src \, e) \, [e] \, v2 \, subs) \)
by \((\text{induct es arbitrary : v1, auto})\)

**lemma ces-append :**

\( ces \, v1 \, (es1 \# es2) \, v2 \, subs = (\exists \, v. \, ces \, v1 \, es1 \, v \, subs \wedge ces \, v \, es2 \, v2 \, subs) \)
proof \((\text{intro iffI, goal-cases})\)

case 1 thus \(?case\)
by \((\text{induct es1 arbitrary : v1})\)
\((\text{simp-all del : split-paired-Ex, blast})\)
next

case 2 thus \(?case\)
proof \((\text{induct es1 arbitrary : v1})\)

case \((\text{Nil v1})\)

then obtain \( v \) where \( ces \, v1 \, [] \, v \, subs \)

and \( ces \, v \, es2 \, v2 \, subs \)

by \(\text{blast}\)

thus \(?case\)
unfolding \(\text{ces.simps}\)
proof \((\text{elim disjE, goal-cases})\)

case 1 thus \(?case\) by \(\text{simp}\)
next

case 2 thus \(?case\) by \((\text{cases es2})\) \((\text{simp, intro disjI2, fastforce})\)+

next

case \(\text{Cons}\) thus \(?case\) by \(\text{auto}\)

qed

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 $v_2$ is also a vertex of $g$.

**Lemma (in sub-rel-of) ces-imp-ends-vertices:**

assumes $ces\ v_1\ es\ v_2\ subs$
assumes $set\ es\ \subseteq\ edges\ g$
assumes $v_1\ \in\ Graph\ vertices\ g$
shows $v_2\ \in\ Graph\ vertices\ g$
using $assms\ trancl-sub-rel-of$
unfolding $sub-rel-of-def\ subsumers-conv\ vertices-def$
by $(induct\ es\ arbitrary:\ v_1)\ (force,\ (simp\ del:split-paired-Ex,\ fast))$

### 9.3.2 Sub-paths

A sub-path leading from $v_1$ to $v_2$, two vertices of a graph $g$ equipped with a subsumption relation $subs$, is a sequence of edges consistent wrt. $subs$ from $v_1$ to $v_2$ 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 ::**

$((\forall v\ \in\ nat, x)\ rgraph-scheme\ \Rightarrow\ ((\forall v\ \in\ nat)\ \Rightarrow\ (\forall v\ \in\ nat)\ \Rightarrow\ (\forall v\ \in\ nat))\ \Rightarrow\ (\forall v\ \in\ nat)\ \Rightarrow\ bool$

where

$\text{subpath}(g, v_1, es, v_2, subs) \equiv sub\ rel\ of\ g\ subs$
$\wedge\ v_1\ \in\ Graph\ vertices\ g$
$\wedge\ ces\ v_1\ es\ v_2\ subs$
$\wedge\ set\ es\ \subseteq\ edges\ g$

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

**Abbreviation subpath-from ::**

$((\forall v\ \in\ nat, x)\ rgraph-scheme\ \Rightarrow\ ((\forall v\ \in\ nat)\ \Rightarrow\ (\forall v\ \in\ nat))\ \Rightarrow\ bool$

where

$\text{subpath-from}(g, v, es, subs) \equiv \exists v'.\ \text{subpath}(g, v, es, v')\ subs$

Simplification lemmas for $SubRel\ subpath$.

**Lemma Nil-sp :**

$\text{subpath}(g, v_1\ \|\ v_2, subs) \leftrightarrow sub\ rel\ of\ g\ subs$
$\wedge\ v_1\ \in\ Graph\ vertices\ g$
$\wedge\ (v_1 = v_2 \vee (v_1, v_2) \in subs^+)$

by $(auto\ simp\ add: subpath-def)$

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:
\[ \text{subpath } g \text{ } v1 \text{ } \{\} \text{ } v2 \text{ } \text{sub} \leftrightarrow \text{sub-rel-of } g \text{ } \text{sub} \]
\[ \land \text{ } v1 \in \text{Graph.} \text{vertices } g \]
\[ \land \text{ } (v1 = v2 \lor (v1, v2) \in \text{sub}) \]
using trancl-eq by (simp add : Nil-sp)

Simplification lemma for the one-element sequence.

lemma sp-one:
- shows \[ \text{subpath } g \text{ } v1 \text{ } [e] \text{ } v2 \text{ } \text{sub} \leftrightarrow \text{sub-rel-of } g \text{ } \text{sub} \]
  \[ \land \text{ } (v1 = \text{src } e \lor (v1, \text{src } e) \in \text{sub}^+) \]
  \[ \land \text{ } e \in \text{edges } g \]
  \[ \land \text{ } (\text{tgt } e = v2 \lor (\text{tgt } e, v2) \in \text{sub}^+) \]
using sub-rel-of.trancl-sub-rel-of[of g sub]
by (intro iffI, auto simp add : vertices-def sub-rel-of-def subpath-def)

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 \[ \text{subpath } g \text{ } v1 \text{ } [e] \text{ } v2 \text{ } \text{sub} \leftrightarrow \text{sub-rel-of } g \text{ } \text{sub} \]
  \[ \land \text{ } (v1 = \text{src } e \lor (v1, \text{src } e) \in \text{sub}) \]
  \[ \land \text{ } e \in \text{edges } g \]
  \[ \land \text{ } (\text{tgt } e = v2 \lor (\text{tgt } e, v2) \in \text{sub}^+) \]
using sp-one wf-sub-rel.trancl-eq[OF wf-sub-rel] by fast

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

lemma sp-Cons:
- shows \[ \text{subpath } g \text{ } v1 \text{ } (e \# \text{es}) \text{ } v2 \text{ } \text{sub} \leftrightarrow \text{sub-rel-of } g \text{ } \text{sub} \]
  \[ \land \text{ } (v1 = \text{src } e \lor (v1, \text{src } e) \in \text{sub}^+) \]
  \[ \land \text{ } e \in \text{edges } g \]
  \[ \land \text{ } \text{subpath } g \text{ } (\text{tgt } e) \text{ } \text{es} \text{ } v2 \text{ } \text{sub} \]
using sub-rel-of.trancl-sub-rel-of[of g sub]
by (intro iffI, auto simp add : subpath-def vertices-def sub-rel-of-def)

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

lemma (in wf-sub-rel-of) sp-Cons:
- \[ \text{subpath } g \text{ } v1 \text{ } (e \# \text{es}) \text{ } v2 \text{ } \text{sub} \leftrightarrow \text{sub-rel-of } g \text{ } \text{sub} \]
  \[ \land \text{ } (v1 = \text{src } e \lor (v1, \text{src } e) \in \text{sub}^+) \]
  \[ \land \text{ } e \in \text{edges } g \]
  \[ \land \text{ } \text{subpath } g \text{ } (\text{tgt } e) \text{ } \text{es} \text{ } v2 \text{ } \text{sub} \]
using sp-Cons wf-sub-rel.trancl-eq[OF wf-sub-rel] by fast

Simplification lemma for SubRel.subpath when the sequence is known to end by a given edge.
lemma \textit{sp-append-one}:
\begin{align*}
\text{subpath } g &\ v1 \ (es @ [e]) \ v2 \ \text{subs} \iff \text{subpath } g \ v1 \ es \ (src \ e) \ \text{subs} \\
&\land e \in \text{edges } g \\
&\land (tgt \ e = v2 \lor (tgt \ e, v2) \in \text{subs}^+)
\end{align*}

\textbf{unfolding subpath-def by (auto simp add : ces-append-one)}

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

\textbf{lemma (in \textit{wf-sub-rel}) \textit{sp-append-one}:
\begin{align*}
\text{subpath } g &\ v1 \ (es @ [e]) \ v2 \ \text{subs} \iff \text{subpath } g \ v1 \ es \ (src \ e) \ \text{subs} \\
&\land e \in \text{edges } g \\
&\land (tgt \ e = v2 \lor (tgt \ e, v2) \in \text{subs})
\end{align*}

\textbf{using \textit{sp-append-one in-trancl-imp by fast}}

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

\textbf{lemma \textit{sp-append}:
\begin{align*}
\text{subpath } g &\ v1 \ (es1 @ es2) \ v2 \ \text{subs} \iff \\
&\exists v. \text{subpath } g \ v1 \ es1 \ v \ \text{subs} \land \text{subpath } g \ v \ es2 \ v2 \ \text{subs}
\end{align*}

\textbf{proof (intro iffI, goal-cases)}
\begin{itemize}
    \item \textbf{case 1 thus ?case}
    \begin{itemize}
        \item \textbf{using sub-rel-of.ces-imp-ends-vertices}
        \item \textbf{by (simp add : subpath-def ces-append) blast}
    \end{itemize}
\end{itemize}
next
\begin{itemize}
    \item \textbf{case 2 thus ?case}
    \begin{itemize}
        \item \textbf{unfolding subpath-def}
        \item \textbf{by (simp only : ces-append) fastforce}
    \end{itemize}
\end{itemize}
\textbf{qed}

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

\textbf{lemma \textit{fst-of-sp-is-vert}:
\begin{itemize}
    \item \textbf{assumes subpath } g \ v1 \ es \ v2 \ \text{subs}
    \item \textbf{shows } v1 \in \text{Graph.vertices } g
\end{itemize}

\textbf{using assms by (simp add : subpath-def)}

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

\textbf{lemma \textit{lst-of-sp-is-vert}:
\begin{itemize}
    \item \textbf{assumes subpath } g \ v1 \ es \ v2 \ \text{subs}
    \item \textbf{shows } v2 \in \text{Graph.vertices } g
\end{itemize}

64
using assms sub-rel-of.trancl-sub-rel-of[of g subs]
by (induction es arbitrary : v1)
  (force simp add : subpath-def sub-rel-of-def, (simp add : sp-Cons, fast))

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 (cases es)
  case Nil
  moreover
  hence v1 ∈ Graph.vertices g
  and v1 = v2 ∨ (v1,v2) ∈ subs⁺
  using assms(1) by (simp-all add : Nil-sp)

  ultimately
  show ?thesis
  using assms(1,2)
    Nil-sp[of g v1 v2 subs]
    trancl-into-trancl[of v1 v2 subs v3]
  by (auto simp add : subpath-def)
next
  case Cons

  then obtain es' e where es = es' @ [e] using neq-conv2[of es] by blast

  thus ?thesis using assms trancl-into-trancl by (simp add : sp-append-one) fast qed

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 = [] \lor subpath g v2 es v subs
using assms
  wf-sub-rel.subsumed-by-two-imp[OF wf-sub-rel assms(1)]
by (cases es)
  (fast, (intro disjI2, fastforce simp add : sp-Cons))

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 = []
using assms tranclD by (cases es) (auto simp add : sp-Cons, force)

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 \exists es', es = es' @ [e] \land e /∈ set es'
using assms(3,4)
proof (induction es arbitrary : v1)
  case (Nil v1) thus \?case by simp
next
  case (Cons e' es v1)

  hence subpath g (tgt e') es v2 subs
  and e = e' \lor (e \neq e' \land e \in set es) by (auto simp add : sp-Cons)

  thus \?case
proof (elim disjE, goal-cases)
  case 1 thus \?case
  using sp-from-de-empty[OF assms(1,2)] by fastforce
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 \in \mathit{Graph.\text{vertices}} \( g \) \)
- **Assumes** \( v \neq tgt \in \mathit{Graph.\text{vertices}} \( g \) \)
- **Assumes** `subpath (add-edge g e) v es (tgt e) subs`
- **Shows** \( e \in \text{set} es \)

**Proof** –

1. **Have** \( tgt \notin \mathit{subsumers subs} \) using `assms(1) sub-rel-of` by `auto`
2. **Hence** \( (v, tgt) \notin \text{subs}^+ \) using `tranclD2` by `force`
3. **Hence** \( es \neq [] \) using `assms(2,3)` by `auto simp : Nil-sp`
4. **Then obtain** \( es \leftrightarrow e' \) where \( es' = es' \circ [e'] \) by `simp add : neq-Nil-conv2` `blast`
5. **Moreover**
   - **Hence** \( e' \in \mathit{edges (add-edge g e)} \) using `assms(3)` by `auto simp add : subpath-def`
6. **Moreover**
   - **Have** \( tgt e' = tgt e \)
   - using `tranclD2` `assms(3)` `\langle tgt \notin \mathit{subsumers subs} \rangle` `\langle es = es' \circ [e'] \rangle` `by (force simp add : sp-append-one)`
7. **Ultimately**
   - **Show** `?thesis` using `assms(1)` unfolding `vertices-def image-def` by `force`

**qed**

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 \( \text{extends} :: \)
\( (\ 'v', 'x' ) \ rgraph-scheme \Rightarrow 'v' \ edge \Rightarrow (\ 'v', 'x' ) \ rgraph-scheme \Rightarrow \text{bool} \)

where
\( \text{extends} g e g' \equiv \text{src} e \in \text{Graph.vertices} g \)
\( \land \ tgt e \notin \text{Graph.vertices} g \)
\( \land \ g' = (\text{add-edge} \ g \ e) \)

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

lemma \( \text{extends-tgt-out-edges} : \)
\( \text{assumes} \ \text{extends} g e g' \)
\( \text{shows} \ \text{out-edges} g' (\text{tgt} e) = \{\} \)

using \( \text{assms unfolding vertices-def image-def by force} \)

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

lemma (in \( \text{sub-rel-of} \))
\( \text{assumes} \ \text{extends} g e g' \)
\( \text{shows} \ \text{sub-rel-of} g' \ subs \)

using \( \text{assms sub-rel-of by (auto simp add : sub-rel-of-def vertices-def)} \)

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

lemma \( \text{sp-in-extends} : \)
\( \text{assumes} \ \text{extends} g e g' \)
\( \text{assumes} \ \text{Graph.subpath} g \ v1 \ es \ v2 \)
\( \text{shows} \ \text{Graph.subpath} g' v1 \ es \ v2 \)

using \( \text{assms by (auto simp add : Graph.subpath-def vertices-def)} \)
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** (intro allI impI)

fix v

have root g' = root g using assms(2) by simp

assume v ∈ Graph.vertices g'

hence v ∈ Graph.vertices g ∨ v = tgt e using assms(2) by (auto simp add : vertices-def)

thus ∃! es. path g' es v

**Proof** (elim disjE, goal-cases)

**Case 1**

then obtain es

where Graph.path g es v

and ∀ es'. Graph.path g es' v → es' = es

using assms(1) unfolding Ex1-def is-tree-def by blast

hence Graph.path g' es v

using assms(2) sp-in-extends[OF assms(2)]

by (subst ⟨root g' = root g⟩)

moreover

have ∀ es'. Graph.path g' es' v → es' = es

**Proof** (intro allI impI)

fix es'

assume Graph.path g' es' v

thus es' = es

**Proof** (case-tac e ∈ set es', goal-cases)

**Case 1**
then obtain \( es'' \)
where \( es' = es'' @ [e] \)
and \( e \notin set es'' \)
using \((\Graph\text{path } g' \ es' v)\)
\((\Graph\text{sp-through-de-decomp}(OF \text{extends-tgt-out-edges}(OF \text{assms}(2)))\)
by blast

hence \( v = \text{tgt } e \)
using \((\Graph\text{path } g' \ es' v)\)
by (simp add : \Graph\text{sp-append-one})

thus \(?\text{thesis}\)
using \assms(2)
\((\Graph\text{lst-of-sp-is-vert}(OF \text{assms}(\Graph\text{path } g \ es v))\)
by simp

next

case 2 thus \(?\text{thesis}\)
using \assms
\((\forall es'. \Graph\text{path } g' \ es' v \rightarrow es' = es) \langle\Graph\text{path } g' \ es' v\rangle\)
by (auto simp add : \Graph\text{subpath-def vertices-def})

qed

ultimately
show \(?\text{thesis}\) by auto

next

case 2

then obtain \( es' \)
where \( \Graph\text{path } g \ es (\text{src } e) \)
and \( \forall es'. \Graph\text{path } g \ es' (\text{src } e) \rightarrow es' = es \)
using \assms(1,2) unfolding is-tree-def by blast

hence \( \Graph\text{path } g' \ es (\text{src } e) \)
using \sp-in-extends[OF \assms(2)]
by (subst \root' = \root g)

hence \( \Graph\text{path } g' (es @ [e]) (\text{tgt } e) \)
using \assms(2) by (auto simp add : Graph.sp-append-one)

moreover
have \( \forall es'. \Graph\text{path } g' \ es' (\text{tgt } e) \rightarrow es' = es @ [e] \)
proof (intro \allI \impI)
  fix \( es' \)
assume Graph.\textit{path} \( g' \) \( es' \) (tgt \( e \))

moreover
\textbf{hence} \( e \in \text{set} \ es' \)
using \textit{assms}
\begin{quote}
\emph{sp-ends-in-tgt-imp-mem[of e g root g es']}
\end{quote}
by (auto simp add : \textit{Graph.subpath-def vertices-def})

moreover
\textbf{have} \( \text{out-edges} g' (\text{tgt} e) = \{\} \)
using \textit{assms} by (intro \textit{extends-tgt-out-edges})

ultimately
\textbf{have} \( \exists \ es''. \ es' = \mathit{es''} @ [e] \land e \notin \text{set} \ es'' \)
by (elim \textit{Graph.sp-through-de-decomp})

\textbf{then obtain} \( \mathit{es''} \)
\textbf{where} \( \mathit{es' = \mathit{es''} @ [e]} \)
\textbf{and} \( e \notin \text{set} \ es'' \)
\textbf{by blast}

\textbf{hence} \( \text{Graph.path} g' \ es'' (\text{src} e) \)
using \( \langle \text{Graph.path} g' \ es' (\text{tgt} e) \rangle \)
by (auto simp add : \textit{Graph.sp-append-one})

\textbf{hence} \( \text{Graph.path} g \ es'' (\text{src} e) \)
using \textit{assms(2) \langle e \notin \text{set} \ es'' \rangle}
by (auto simp add : \textit{Graph.subpath-def vertices-def})

\textbf{hence} \( \mathit{es'' = es} \)
using \( \forall \ as'. \ \text{Graph.path} g \ as' (\text{src} e) \rightarrow as' = es \)
by simp

\textbf{thus} \( \mathit{es' = es @ [e]} \) using \( \langle \mathit{es'} = \mathit{es''} @ [e] \rangle \) by simp
\textbf{qed}

ultimately
\textbf{show} \ ?thesis using 2 by auto
\textbf{qed}
\textbf{qed}
10.3 Properties of sub-paths in an extension

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

**Lemma** \texttt{sp-in-extends-w-subs}:

\[ \text{assumes} \quad \text{extends } g \ a \ g' \]
\[ \text{assumes} \quad \text{subpath } g \ v1 \ es \ v2 \ subs \]
\[ \text{shows} \quad \text{subpath } g' \ v1 \ es \ v2 \ subs \]
\[ \text{using} \ \text{assms by (auto simp add : subpath-def sub-rel-of-def vertices-def)} \]

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 \texttt{sub-rel-of}) \texttt{sp-from-old-verts-imp-sp-in-old}:

\[ \text{assumes} \quad \text{extends } g \ e \ g' \]
\[ \text{assumes} \quad v1 \in \text{Graph.vertices } g \]
\[ \text{assumes} \quad v2 \in \text{Graph.vertices } g \]
\[ \text{assumes} \quad \text{subpath } g' \ v1 \ es \ v2 \ subs \]
\[ \text{shows} \quad \text{subpath } g \ v1 \ es \ v2 \ subs \]
\[ \text{proof} \]
\[ \quad \text{have } e \notin \text{set } es \]
\[ \quad \text{proof (intro notI)} \]
\[ \quad \quad \text{assume } e \in \text{set } es \]
\[ \quad \quad \quad \text{have } v2 = \text{tgt } e \]
\[ \quad \quad \quad \text{proof} \]
\[ \quad \quad \quad \quad \text{have } \text{tgt } e \notin \text{subsumees } subs \text{ using sub-rel-of assms(1) by fast} \]
\[ \quad \quad \text{moreover} \]
\[ \quad \quad \quad \text{have } \text{out-edges } g' (\text{tgt } e) = \{\} \text{ using assms(1) by (rule extends-tgt-out-edges)} \]
\[ \quad \quad \text{ultimately} \]
\[ \quad \quad \quad \text{have } \exists \ es'. \ es = es' @ [e] \land e \notin \text{set } es' \]
\[ \quad \quad \quad \text{using assms(4) \& } e \in \text{set } es' \]
\[ \quad \quad \quad \text{by (intro sp-through-de-decomp)} \]
\[ \quad \quad \text{then obtain } es' \text{ where } es = es' @ [e] \land e \notin \text{set } es' \text{ by blast} \]
\[ \quad \quad \text{hence } \text{tgt } e = v2 \lor (\text{tgt } e, v2) \in \text{subs}^+ \]
\[ \quad \quad \text{using assms(4) by (simp add : sp-append-one)} \]
\[ \quad \text{thus } \text{thesis using } (\text{tgt } e \notin \text{subsumees } subs) \text{ tranclD[of tgt } e \ v2 \ subs] \text{ by force qed} \]
\[ \quad \text{thus False using assms(1,3) by simp} \]
thus \(？thesis\)

using \(sub-rel-of\) \(assms\)

unfolding \(subpath-def\) \(sub-rel-of-def\) by auto

qed

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 \(es = []\)

using \(sub-rel-of\) \(assms\)

extends-tgt-out-edges

\(sp-from-de-empty[\text{of} tgt\ e\ subs\ g'\ es\ v]\)

by fast

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 \(\exists es'. es = es' @ [e] \land e \notin set\ es' \land subpath\ g\ v\ es\'\ (src\ e)\ subs\)

proof –

obtain \(es'\) where \(es = es' @ [e]\) and \(e \notin set\ es'\)

using \(sub-rel-of\) \(assms(1,2,3)\)

extends-tgt-out-edges[\(OF\ assms(1)\)]

sp-through-de-decomp[\(of\ e\ subs\ g'\ v\ es\ tgt\ e\)]

sp-ends-in-tgt-imp-mem[\(of\ e\ v\ es\)]

by blast

moreover

have \(subpath\ g\ v\ es'\ (src\ e)\ subs\)

proof –

have \(v \in\ Graph.\vertices\ g\)

using \(assms(1,3)\) \(fst-of-sp-is-vert[\(OF\ assms(2)\)]\)

by \(auto\ simp\ add:\ vertices-def\)

moreover

have \(SubRel.\subpath\ g'\ v\ es'\ (src\ e)\ subs\)
using assms(2) 〈es = es’ @ [e]〉 by (simp add : sp-append-one)

ultimately
  show ?thesis
  using assms(1) sub-rel-of 〈e ∉ set es’〉
  unfolding subpath-def by (auto simp add : sub-rel-of-def)
qed

ultimately
  show ?thesis by blast
qed

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 subsomption relation after adding a subsomption to this relation. We are only interested in adding subsomptions 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 subsomption to be added.

11.1 Definition

Extending a subsomption relation subs consists in adding a subsomption sub such that:

- the two vertices involved are distinct,
- they are occurrences of the same vertex,
- they are both vertices of the graph,
- the subsume must not already be a subsumer or a subsumee,
- the subsumer must not be a subsumee (but it can already be a sub-
sumer),
• 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 × nat),'z) rgraph-scheme ⇒ 'v sub-rel-t ⇒ 'v sub-t ⇒ 'v sub-rel-t ⇒ bool
where
extends g subs sub subs' ≡ (subsumee sub ≠ subsumer sub ∧ fst (subsumee sub) = fst (subsumer sub) ∧ subsumee sub ∈ Graph.vertices g ∧ subsumee sub ∉ subsumers subs ∧ subsumee sub ∉ subsumees subs ∧ subsumer sub ∈ Graph.vertices g ∧ subsumer sub ∉ subsumees subs ∧ out-edges g (subsumee sub) = {} ∧ subs' = subs ∪ {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'
using assms sub-rel-of unfolding sub-rel-of-def by force

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'
unfolding wf-sub-rel-def
proof (intro conjI, goal-cases)
case 1 show ?case using wf-sub-rel assms by auto
next
case 2 show ?case
unfolding Ball-def
proof (intro allI impI)
fix $v$

assume $v \in \text{subsumees} \; \text{subs}'$

hence $v = \text{subsume} \; \text{sub} \lor v \in \text{subsumees} \; \text{subs}$ using assms by auto

thus $\exists! \; v'. \; ((v,v') \in \text{subs}')$

proof (elim disjE, goal-cases)

case 1 show ?thesis

unfolding $\text{Ex1-def}$

proof (rule-tac ?x=\text{subsumer sub in ex1, intro conj})

show $(v, \text{subsumer sub}) \in \text{subs}'$ using 1 assms by simp

next

have $v \notin \text{subsumees} \; \text{subs}$ using assms 1 by auto

thus $\forall \; v'. \; (v, v') \in \text{subs}' \longrightarrow v' = \text{subsumer sub}$

using assms by auto force

qed

next

then obtain $v'$ where $(v,v') \in \text{subs}$ by auto

hence $v \neq \text{subsume} \; \text{sub}$

using assms unfolding $\text{subsumees-conv}$ by (force simp del : split-paired-All split-paired-Ex)

show ?thesis

using assms

$(v \neq \text{subsume} \; \text{sub})$

$(v,v') \in \text{subs'} \; \text{subsumed-by-one}$

unfolding $\text{subsumees-conv} \; \text{Ex1-def}$

by (rule-tac ?x=v' in exI)

(auto simp del : split-paired-All split-paired-Ex)

qed

qed

next

case 3 show ?case using $\text{wf-sub-rel}$ assms by auto

qed

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

lemma (in $\text{wf-sub-rel-of}$) $\text{extends-imp-wf-sub-rel-of} :$

assumes $\text{extends} \; g \; \text{subs} \; \text{sub} \; \text{subs}'$

shows $\text{wf-sub-rel-of} \; g \; \text{subs}'$

using $\text{sub-rel-of}$ assms
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'`
- `subpath g v1 es v2 subs`

Shows: `subpath g v1 es v2 subs'`

Using: `assms ces-Un[of v1 es v2 subs {sub}]`

By: `(simp add : subpath-def sub-rel-of-def)`

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 subsumer of the new subsumption, then the sub-paths starting at any other vertex.

First, we are interested in sub-paths starting at the subsumer 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'`
- `subpath g v1 es v2 subs'

Shows: `es = [] \lor subpath g v2 es v subs'`

Using: `assms`  
- `extends-imp-wf-sub-rel-of[of assms(1)]`
- `wf-sub-rel-of.sp-from-subsumee[of g subs' v1 v2 es v]`

By: `simp`

After an extension, sub-paths starting at any other vertex than the new subsumer 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 $\text{sub}$ is consistent wrt. a subsumption relation $\text{subs}$, then $\text{sub}$ must occur in the transitive closure of $\text{subs}$ i.e. the consistency of the sequence directly (and partially) depends on $\text{sub}$. 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 $\text{sub}$ is a member of $\text{subs}$.

$$\text{fun \ uses-sub ::}$$

$$(\text{'v} \times \text{nat}) \Rightarrow (\text{'v} \times \text{nat}) \Rightarrow ((\text{'v} \times \text{nat}) \times (\text{'v} \times \text{nat})) \Rightarrow \text{bool}$$

where

- $\text{uses-sub v1 [] v2 sub} = (v1 \neq v2 \land \text{sub} = (v1,v2))$
- $\text{uses-sub v1 (e#es) v2 sub} = (v1 \neq \text{src e} \land \text{sub} = (v1,\text{src e}) \lor \text{uses-sub (tgt e) es v2 sub})$

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

**lemma**

- **assumes** $\text{uses-sub v1 es v2 sub}$
- **assumes** $\text{ces v1 es v2 subs}$
- **shows** $\text{sub} \in \text{subs}^+$
  using **assms** by (induction $\text{es}$ arbitrary : $v1$) fastforce+

This reduces to the membership of $\text{sub}$ to $\text{subs}$ when the latter is well-formed.

**lemma (in \text{wf-sub-rel})**

- **assumes** $\text{uses-sub v1 es v2 sub}$
- **assumes** $\text{ces v1 es v2 subs}$
- **shows** $\text{sub} \in \text{subs}$
  using **assms trancl-eq** by (induction $\text{es}$ arbitrary : $v1$) fastforce+

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

**lemma extends-and-sp-imp-not-using-sub :**

- **assumes** $\text{extends g subs (v,v')} \text{ subs'}$
- **assumes** $\text{subpath g v1 es v2 subs}$
- **shows** $\neg \text{uses-sub v1 es v2 (v,v')}$
  proof (intro notI)
  assume $\text{uses-sub v1 es v2 (v,v')}$

moreover

- **have** $\text{ces v1 es v2 subs using assms(2) by (simp add : subpath-def)}$

ultimately

- **have** $(v,v') \in \text{subs}^+$ by (induction $\text{es}$ arbitrary : $v1$) fastforce+
thus False
using assms(1) unfolding subsumees_conv
by (elim conjE) (frule tranclD, force)
qed

Suppose that the empty sequence is a sub-path leading from \( v_1 \) to \( v_2 \) after the extension. Then, the empty sequence is a sub-path leading from \( v_1 \) to \( v_2 \) in the graph before the extension if and only if \((v_1, v_2)\) 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 \iff (v1 \neq v \lor v2 \neq v')
proof (intro iffI, goal-cases)
  case 1 thus \?case
  using assms
  extends-and-sp-imp-not-using-sub[OF assms(1), of v1 [] v2]
  by auto
next
  case 2
  have v1 = v2 \lor (v1,v2) \in subs'
  and v1 \in Graph.vertices g
  using assms(2)
  wf-sub-rel.extends-imp-wf-sub-rel[OF wf-sub-rel assms(1)]
  by (simp-all add : wf-sub-rel.Nil-sp)
moreover
  hence v1 = v2 \lor (v1,v2) \in subs
  using assms(1) 2 by auto
moreover
  have v2 \in Graph.vertices g
  using assms(2) by (intro lst-of-sp-is-vert)
ultimately
  show subpath g v1 [] v2 subs
  using sub-rel-of by (auto simp add : subpath-def)
qed

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 :
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`

- **Show** `subpath g v1 es v2 subs`

**Proof** (case-tac uses-sub v1 es v2 (v,v'), goal-cases)

Case 1

Thus `?thesis`

Using `assms(2,3)`

Proof (induction es arbitrary : v1)
case (Nil v1) thus ?case by auto

next
case (Cons edge es v1)

hence v1 = src edge ∨ (v1, src edge) ∈ subs'
and edge ∈ edges g
and subpath g (tgt edge) es v2 subs'
using assms(1) extends-imp-wf-sub-rel-of
by (simp-all add : wf-sub-rel-of.sp-Cons)

hence subpath g v1 [edge] (tgt edge) subs'
using wf-sub-rel-of.sp-one[OF extends-imp-wf-sub-rel-of[OF assms(1)]]
by (simp add : subpath-def) fast

have subpath g v1 [edge] (tgt edge) subs
proof −
have ¬ uses-sub v1 [edge] (tgt edge) (v,v')
using assms(1) Cons(2,4) by auto

thus ?thesis
using assms(1) (subpath g v1 [edge] (tgt edge) subs')
by (elim sp-in-extends-not-using-sub)
qed

thus ?case
proof (case-tac tgt edge = v, goal-cases)
case 1 thus ?thesis
using (subpath g v1 [edge] (tgt edge) subs)
  (subpath g (tgt edge) es v2 subs');
by (intro disjI2, rule-tac ?x=[edge] in exI) auto
next
case 2

moreover
have uses-sub (tgt edge) es v2 (v,v') using Cons(2,4) by simp

ultimately
have ?P es (tgt edge)
using (subpath g (tgt edge) es v2 subs');
by (intro Cons.IH)

thus ?thesis
proof (elim disjE exE conjE, goal-cases)
case 1 thus ?thesis
using (subpath g (tgt edge) es v2 subs');
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**,
• \textit{subs} is the subsumption relation over the vertices of \textit{red},

• \textit{init-conf} is the initial configuration,

• \textit{confs} is a function associating configurations to the vertices of \textit{red},

• \textit{marked} is a function associating truth values to the vertices of \textit{red}. We use it to represent the fact that a particular configuration (associated to a red location) is known to be unsatisfiable,

• \textit{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 \textit{RedBlack}. From now on, we call “red-black graphs” the pre-\textit{RedBlack}’s obtained by \textit{RedBlack} and “pre-red-black graphs” all other ones.

\begin{verbatim}
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
\end{verbatim}

We call \textit{red vertices} the set of vertices of the red graph.

\begin{verbatim}
abbreviation red-vertices ::
  (′vert,′var,′d,′x) pre-RedBlack-scheme ⇒ (′vert × nat) set
where
  red-vertices lts ≡ Graph.vertices (red lts)
\end{verbatim}

\textit{ui-edge} is the operation of “unindexing” the ends of a red edge, thus giving the corresponding black edge.

\begin{verbatim}
abbreviation ui-edge ::
  (′vert × nat) edge ⇒ ′vert edge
where
  ui-edge e ≡ (| src = fst (src e), tgt = fst (tgt e) |)
\end{verbatim}

We extend this idea to sequences of edges.

\begin{verbatim}
abbreviation ui-es ::
  (′vert × nat) edge list ⇒ ′vert edge list
where
  ui-es es ≡ map ui-edge es
\end{verbatim}
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-RedBlacks 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 ::

\[
\begin{align*}
('\text{vert}', '\text{var}', 'd') \text{ pre-RedBlack} & \Rightarrow ('\text{vert} \times \text{nat}) \text{ edge} \\
& \Rightarrow ('\text{var}', 'd') \text{ conf} \\
& \Rightarrow ('\text{vert}', '\text{var}', 'd') \text{ pre-RedBlack} \Rightarrow \text{ bool}
\end{align*}
\]

**where**

\[
\begin{align*}
\text{se-extends prb re c prb'} \equiv \\
\quad \text{ui-edge re} & \in \text{ edges (black prb)} \\
\quad \land \text{ArcExt.extends (red prb) re (red prb')} \\
\quad \land \text{src re} & \notin \text{ subsumees (subs prb)} \\
\quad \land \text{se (confs prb (src re)) (labelling (black prb) (ui-edge re)) c} \\
\quad \land \text{prb'} = (\text{\{ red } \quad = \text{ red prb'},} \\
\quad \text{black} & = \text{ black prb,} \\
\quad \text{subs} & = \text{ subs prb,} \\
\quad \text{init-conf} & = \text{ init-conf prb,} \\
\quad \text{confs} & = (\text{confs prb}) (\text{tgt re := c}), \\
\quad \text{marked} & = (\text{marked prb})(\text{tgt re := marked prb (src re))},
\end{align*}
\]
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 ::

\(('\text{vert}','\text{var}',d) \; \text{pre-RedBlack} \Rightarrow ('\text{vert}','\text{var}',d) \; \text{pre-RedBlack} \Rightarrow \text{bool}\)

**where**

\[
\text{se-extends2 prb prb'} = \exists \; re \in \text{edges (red prb')}, \quad \\
\text{ui-edge re} \in \text{edges (black prb)} \quad \land \quad \text{ArcExtextends (red prb) re (red prb')}, \quad \\
\text{src re} \notin \text{subsumees (subs prb)} \quad \land \quad \text{se (confs prb (src re)) (labelling (black prb) (ui-edge re)) (confs prb' (tgt re))}, \quad \\
\text{black prb'} = \text{black prb} \quad \land \quad \text{subs prb'} = \text{subs prb} \quad \land \quad \text{init-conf prb'} = \text{init-conf prb}, \quad \\
\text{confs prb'} = (\text{confs prb}) (\text{tgt re} := \text{confs prb'} (\text{tgt re})), \quad \\
\text{marked prb'} = (\text{marked prb}) (\text{tgt re} := \text{marked prb} (\text{src re})), \quad \\
\text{strengthenings prb'} = \text{strengthenings prb}
\]

**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 ::

\(('\text{vert}','\text{var}',d) \; \text{pre-RedBlack} \Rightarrow ('\text{vert} \times \text{nat}) \Rightarrow ('\text{vert}','\text{var}',d) \; \text{pre-RedBlack} \Rightarrow \text{bool}\)

**where**

\[
\text{mark-extends prb rv prb'} \equiv \quad \\
\text{rv} \in \text{red-vertices prb} \quad \land \quad \text{out-edges (red prb) rv} = \{\} \quad \land \quad \text{rv} \notin \text{subsumees (subs prb)} \quad \land \quad \text{rv} \notin \text{subsumers (subs prb)} \quad \land \quad \neg \text{sat (confs prb rv)} \quad \land \quad \text{prb'} = (\lambda \; \text{red} \quad = \text{red prb}, \quad \\
\text{black} \quad = \text{black prb}, \quad \text{subs} \quad = \text{subs prb}, \quad \text{init-conf} \quad = \text{init-conf prb}, \quad \\
\text{confs} \quad = \text{confs prb}, \quad \text{marked} \quad = (\lambda \; \text{rv}'. if \text{rv'} = \text{rv} \text{ then True else marked prb rv'}),
\]

86
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 ⇒ 'vert sub-t ⇒ ('vert,'var,'d) pre-RedBlack ⇒ bool

where
subsum-extends prb sub prb' ≡
  SubExt.extends (red prb) (subs prb) sub (subs prb')
  ∧ ¬ marked prb (subsumer sub)
  ∧ ¬ marked prb (subsumee sub)
  ∧ confs prb (subsumee sub) ⊑ confs prb (subsumer sub)
  ∧ 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,
            ... = 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-RedBlack
  ⇒ ('vert × nat)
  ⇒ ('var,'d) conf
  ⇒ ('vert,'var,'d) pre-RedBlack
  ⇒ bool

where
abstract-extends prb rv ca prb' ≡
  rv ∈ red-vertices prb
  ∧ ¬ marked prb rv
  ∧ out-edges (red prb) rv = {}
  ∧ rv /∈ subsumees (subs prb)
  ∧ abstract (confs prb rv) ca
  ∧ ca |=c (strengthenings prb rv)
\[ \begin{align*}
\land \text{finite (pred } c_a) \\
\land (\forall e \in \text{pred } c_a. \text{finite (vars } e)) \\
\land \text{prb}^\prime = \{ \begin{array}{l}
\text{red } = \text{red prb}, \\
\text{black } = \text{black prb}, \\
\text{subs } = \text{subs prb}, \\
\text{init-conf } = \text{init-conf prb}, \\
\text{confs } = (\text{confs prb}) (rv := c_a), \\
\text{marked } = \text{marked prb}, \\
\text{strengthenings } = \text{strengthenings prb}, \\
\ldots = \text{more prb} \}
\end{array} \}
\end{align*} \]

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 ::

\[ ('\text{vert}', '\text{var}', 'd') \text{ pre-RedBlack} \]

\[ \Rightarrow ('\text{vert} \times \text{nat}) \]

\[ \Rightarrow ('\text{var}', 'd') \text{ bexp} \]

\[ \Rightarrow ('\text{vert}', '\text{var}', 'd') \text{ pre-RedBlack} \]

\[ \Rightarrow \text{bool} \]

**where**

\[ \text{strengthen-extends prb rv e prb}^\prime \equiv \]

\[ rv \in \text{red-vertices prb} \]

\[ \land rv \not\in \text{subsumees (subs prb)} \]

\[ \land \text{confs prb rv } \models_c e \]

\[ \land \text{prb}^\prime = \{ \begin{array}{l}
\text{red } = \text{red prb}, \\
\text{black } = \text{black prb}, \\
\text{subs } = \text{subs prb}, \\
\text{init-conf } = \text{init-conf prb}, \\
\text{confs } = \text{confs prb}, \\
\text{marked } = \text{marked prb}, \\
\text{strengthenings } = (\text{strengthenings prb}) (rv := (\lambda \sigma. (\text{strengthenings prb}
rv) \sigma \land e \sigma)), \\
\ldots = \text{more prb} \}
\end{array} \} \]

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 ::

88
(′vert,’var,’d) pre-RedBlack ⇒ bool

where
base :
  fst (root (red prb)) = init (black prb)  ⇒
edges (red prb) = {}  ⇒
subs prb = {}  ⇒
(confs prb) (root (red prb)) = init-conf prb ⇒
marked prb = (λ rv. False)  ⇒
strengthenings prb = (λ rv. (λ σ. True))  ⇒ RedBlack prb

| se-step :
  RedBlack prb  ⇒
  se-extends prb re p' prb'  ⇒ RedBlack prb'

| mark-step :
  RedBlack prb  ⇒
  mark-extends prb rv prb'  ⇒ RedBlack prb'

| subsum-step :
  RedBlack prb  ⇒
  subsum-extends prb sub prb'  ⇒ RedBlack prb'

| abstract-step :
  RedBlack prb  ⇒
  abstract-extends prb rv c a prb'  ⇒ RedBlack prb'

| strengthen-step :
  RedBlack prb  ⇒
  strengthen-extends prb rv e prb'  ⇒ 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)
  using  assms by (induct prb) auto

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

lemma
  assumes RedBlack prb
  assumes re ∈ edges (red prb)
shows \( \text{tgt re } \neq \text{root (red prb)} \)
using \text{assms by (induct prb) (auto simp add : vertices-def)}

Red edges are specific versions of black edges.

\textbf{lemma \textit{ui-re-is-be} :}
\textbf{assumes \textit{RedBlack prb} \newline shows \textit{ui-edge re } \in \text{edges (black prb)}}
using \text{assms by (induct rule : RedBlack.induct) auto}

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.

\textbf{lemma \textit{red-OA-subset-black-OA} :}
\textbf{assumes \textit{RedBlack prb} \newline shows \textit{ui-edge ’ out-edges (red prb) rv } \subseteq \text{out-edges (black prb) (fst rv)}}
using \text{assms by (induct prb) (fastforce simp add : vertices-def)}+

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

\textbf{lemma \textit{consistent-roots} :}
\textbf{assumes \textit{RedBlack prb} \newline shows \textit{fst (root (red prb))) = init (black prb)}}
using \text{assms by (induct prb) auto}

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

\textbf{lemma}
\textbf{assumes \textit{RedBlack prb} \newline shows \textit{is-tree (red prb)}}
using \text{assms by (induct prb) (auto simp add : empty-graph-is-tree ArcExt.extends-is-tree)}

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

\textbf{lemma}
\textbf{assumes \textit{RedBlack prb} \newline assumes \textit{wf-lts (black prb)} \newline shows \textit{wf-pre-RedBlack prb}}
\textbf{proof –}
\textbf{have \textit{wf-rgraph (red prb)}
using \text{assms by (induct prb) (force simp add : vertices-def)}+}

\textbf{thus \?thesis}
\textbf{using \text{assms consistent-roots ui-re-is-be}
by (auto simp add : wf-pre-RedBlack-def)}
\textbf{qed}
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)
using assms consistent-roots ui-re-is-be
by (auto simp add : vertices-def image-def Bex-def) fastforce+

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)
using assms unfolding sub-rel-of-def
proof (induct prb)
  case base thus ?case by simp
next
  case se-step thus ?case by (elim conjE) (auto simp add : vertices-def)
next
  case mark-step thus ?case by auto
next
  case subsum-step thus ?case by auto
next
  case abstract-step thus ?case by simp
next
  case strengthen-step thus ?case by simp
qed

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

lemma subs-wf-sub-rel :
  assumes RedBlack prb
  shows  wf-sub-rel (subs prb)
using assms
proof (induct prb)
  case base thus ?case by (simp add : wf-sub-rel-def)
next
  case se-step thus ?case by force
next
  case mark-step thus ?case by (auto simp add : wf-sub-rel-def)
next
  case subsum-step thus ?case by (auto simp add : wf-sub-rel.extends-imp-wf-sub-rel)
next
  case abstract-step thus ?case by simp
next
  case strengthen-step thus ?case by simp

91
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)
- **using** assms subs-sub-rel-of subs-wf-sub-rel by (simp add : wf-sub-rel-of-def) fast

Subsumptions only involve red locations representing the same black location.

**lemma** subs-to-same-BL:
- **assumes** RedBlack prb
- **assumes** sub ∈ subs prb
- **shows** fst (subsume sub) = fst (subsumer sub)
- **using** assms subs-wf-sub-rel unfolding wf-sub-rel-def by fast

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)
- **using** assms
- **proof** (induct res arbitrary : rv1)
  - **case** (Nil rv1) **thus** ?case
    - **using** wf-sub-rel.in-trancl-imp|OF subs-wf-sub-rel| subs-to-same-BL by fastforce
  - **next**
    - **case** (Cons re res rv1)
      - **hence** 1 : rv1 = src re ∨ (rv1, src re) ∈ (subs prb){+}
      - **and** 2 : ces (tgt re) res rv2 (subs prb) by simp-all
      - **have** src (ui-edge re) = fst rv1
        - **using** 1 wf-sub-rel.in-trancl-imp|OF subs-wf-sub-rel|OF assms(1), of rv1 src re]
          - subs-to-same-BL|OF assms(1), of (rv1,src re] by auto
      - **moreover**
        - **have** Graph. ces (tgt (ui-edge re)) (ui-es res) (fst rv2)
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)
- **Using** assms rces-imp-bces ui-rv-is-bv ui-re-is-be
- **Unfolding** subpath-def Graph.subpath-def by (intro conjI) (fast, fast, fastforce)

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)
- **Using** assms
- **Proof** (induct prb arbitrary : rv)
  - **Case** base thus ?case by (simp add : vertices-def finite-RedBlack-def)
  - **Next**
    - **Case** (se-step prb re c′ prb′)

    **Hence** rv ∈ red-vertices prb ∨ rv = tgt re by (auto simp add : vertices-def)

    **Thus** ?case
    - **Proof** (elim disjE)
      - **Assume** rv ∈ red-vertices prb

        **Moreover**
        - **Have** finite-RedBlack prb
          - **Using** se-step(3,4) by (auto simp add : finite-RedBlack-def)

        **Ultimately**
        - **Show** ?thesis
          - **Using** se-step(2,3) by (elim conjE) (auto simp add : vertices-def)
        - **Next**
assume \( rv = \text{tgt re} \)

moreover
have finite-label (labelling (black prb) (ui-edge re))
  using se-step by (auto simp add : finite-RedBlack-def)

moreover
have \( \forall e \in \text{pred (confs prb (src re)). finite (Bexp.vars e)} \)
  using se-step se-step(2)[of src re] unfolding finite-RedBlack-def
by (elim conjE) auto

moreover
have se (confs prb (src re)) (labelling (black prb) (ui-edge re)) c'
  using se-step by auto

ultimately
show ?thesis using se-step se-preserves-finiteness1 by fastforce
qed
next
case mark-step thus ?case by (simp add : finite-RedBlack-def)
next
case subsum-step thus ?case by (simp add : finite-RedBlack-def)
next
case abstract-step thus ?case by (auto simp add : finite-RedBlack-def)
next
case strengthen-step thus ?case by (simp add : finite-RedBlack-def)
qed

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 \( rv \in \text{red-vertices prb} \)
  shows finite (pred (confs prb rv))
  using assms
proof (induct prb arbitrary : rv)
case base thus ?case by (simp add : vertices-def finite-RedBlack-def)
next
case (se-step prb re c' prb')

hence \( rv \in \text{red-vertices prb} \lor rv = \text{tgt re} \)
  by (auto simp add : vertices-def)

thus ?case
proof (elim disjE, goal-cases)
case 1 thus ?thesis
  using se-step(2)[of rv] se-step(3,4)
  by (auto simp add : finite-RedBlack-def)
next
case 2
moreover
hence src re ∈ red-vertices prb
and  finite (pred (confs prb (src re)))
  using se-step(2)[of src re] se-step(3,4)
  by (auto simp add : finite-RedBlack-def)
ultimately
  show ?thesis
  using se-step(3) se-preserves-finiteness2 by auto
qed
next
case mark-step thus ?case by (simp add : finite-RedBlack-def)
next
case subsum-step thus ?case by (simp add : finite-RedBlack-def)
next
case abstract-step thus ?case by (simp add : finite-RedBlack-def)
next
case strengthen-step thus ?case by (simp add : finite-RedBlack-def)
qed

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'
using finite-RedBlack assms
  finite-imp-ex-se-succ[of confs prb rv]
  finite-pred[of prb rv ]
  finite-pred-constr-symvars[of prb rv]
unfolding finite-RedBlack-def by fast

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 \( \exists c'. \, \text{se-star}(\text{confs prb rv}) \) is \( c' \)

using \text{finite-RedBlack assms}
\begin{align*}
\text{finite-imp-ex-se-star-succ}[\alpha \text{confs prb rv ls}] & \\
\text{finite-pred}(\text{OF assms}(1), \alpha \text{rv}) & \\
\text{finite-pred-constr-symvars}(\text{OF assms}(1), \alpha \text{rv}) & 
\end{align*}

unfolding \text{finite-RedBlack-def} by simp

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 \text{finite-RedBlack}) \text{sp-imp-ex-se-star-succ}:

assumes \text{RedBlack prb}
assumes \text{subpath (red prb) rv1 res rv2 (subs prb)}
shows \( \exists c. \, \text{se-star} \)
\begin{align*}
(\text{confs prb rv1}) & \\
(\text{trace (ui-es res) (labelling (black prb)))} & \\
\end{align*}
\( c \)

using \text{finite-RedBlack assms ex-se-star-succ}
by (simp add : subpath-def \text{finite-RedBlack-def})

The configuration associated to a red location \( rl \) is update-able.

**lemma** (in \text{finite-RedBlack})

assumes \text{RedBlack prb}
assumes \text{rv \in red-vertices prb}
shows \text{updatable (confs prb rv)}

using \text{finite-RedBlack assms}
\begin{align*}
\text{finite-conj}[\text{OF finite-pred}(\text{OF assms}(1))] & \\
\text{finite-pred-constr-symvars}(\text{OF assms}(1)) & 
\end{align*}

unfolding \text{finite-RedBlack-def} by fast

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

**lemma** \text{sub-subsumed}:

assumes \text{RedBlack prb}
assumes \text{sub \in subs prb}
shows \text{confs prb (subsumee sub) \subseteq confs prb (subsumer sub)}

using \text{assms}
proof (induct prb)
case base thus \text{case by simp}

next
case \text{(se-step prb re c' prb')}

moreover
hence \( \text{sub} \in \text{subs prb} \) by \text{auto}

hence \( \text{subsumee sub} \in \text{red-vertices prb} \) and \( \text{subsumer sub} \in \text{red-vertices prb} \) using \text{se-step(1) subs-sub-rel-of}

unfolding \text{sub-rel-of-def by fast+}

moreover
have \( \text{tgt re} \notin \text{red-vertices prb} \) using \text{se-step by auto}

ultimately
show \text{?case by auto}

next
\text{case mark-step thus ?case by simp}

next
\text{case (subsum-step prb sub prb')} thus \text{?case by auto}

next
\text{case (abstract-step prb rv c_a prb')}

hence \( \text{rv} \neq \text{subsumee sub} \) by \text{auto}

show \text{?case}

proof (\text{case-tac rv = subsumer sub})

assume \( \text{rv} = \text{subsumer sub} \)

moreover
hence \( \text{confs prb (subsumer sub)} \sqsubseteq \text{confs prb' (subsumer sub)} \) using \text{abstract-step abstract-def by auto}

ultimately
show \text{?thesis}

using \text{abstract-step}

\begin{align*}
\text{subsums-trans}(\text{confs prb (subsumee sub)}) & \subseteq \text{confs prb' (subsumer sub)} \\
\text{confs prb (subsumer sub)} & \subseteq \text{confs prb' (subsumer sub)}
\end{align*}

by (\text{simp add : subsums-refl})

next
assume \( \text{rv} \neq \text{subsumer sub} \) thus \text{?thesis using abstract-step \( \langle rv \neq \text{subsumee sub} \rangle \) by simp}

qed

next
\text{case strengthen-step thus ?case by simp}

qed
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) =
  \( (rv1 \in \text{red-vertices prb} \land (rv1 = rv2 \lor (rv1, rv2) \in (\text{subs prb}))) \)
- **using** assms subs-wf-sub-rel subs-sub-rel-of wf-sub-rel.Nil-sp by fast

**lemma rb-sp-one :**

- **assumes** RedBlack prb
- **shows** subpath (red prb) rv1 [re] rv2 (subs prb) =
  \( (\text{sub-rel-of (red prb) (subs prb)} \land (rv1 = \text{src re} \lor (rv1, \text{src re}) \in (\text{subs prb})) \land \text{re} \in \text{edges (red prb)} \land (\text{tgt re} = rv2 \lor (\text{tgt re}, rv2) \in (\text{subs prb}))) \)
- **using** assms subs-wf-sub-rel-of wf-sub-rel.of.sp-one by fast

**lemma rb-sp-Cons :**

- **assumes** RedBlack prb
- **shows** subpath (red prb) rv1 (re # res) rv2 (subs prb) =
  \( (\text{sub-rel-of (red prb) (subs prb)} \land (rv1 = \text{src re} \lor (rv1, \text{src re}) \in \text{subs prb}) \land \text{re} \in \text{edges (red prb)} \land \text{subpath (red prb) (tgt re) res rv2 (subs prb)}) \)
- **using** assms subs-wf-sub-rel-of wf-sub-rel.of.sp-Cons by fast

**lemma rb-sp-append-one :**

- **assumes** RedBlack prb
- **shows** subpath (red prb) rv1 (res @ [re]) rv2 (subs prb) =
  \( (\text{subpath (red prb) rv1 res (src re) (subs prb)} \land \text{re} \in \text{edges (red prb)} \land (\text{tgt re} = rv2 \lor (\text{tgt re}, rv2) \in \text{subs prb})) \)
- **using** assms subs-wf-sub-rel-of wf-sub-rel.sp-append-one sp-append-one by fast

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 ⊑ (confs prb rv2)
  using assms finite-RedBlack
**proof** (induct arbitrary : rv1 res rv2 c rule : RedBlack.induct)
  case (base prb rv1 res rv2 c) thus ?case
    by (force simp add : subpath-def Nil-sp subsums-refl)
  next
  case (se-step prb re c' prb' rv1 res rv2 c)
    have rv1 ∈ red-vertices prb'
      and rv2 ∈ red-vertices prb'
      using fst-of-sp-is-vert[OF se-step(4)]
      lst-of-sp-is-vert[OF se-step(4)]
      by simp-all
    hence rv1 ∈ red-vertices prb ∧ rv1 ≠ tgt re ∨ rv1 = tgt re
      and rv2 ∈ red-vertices prb ∧ rv2 ≠ tgt re ∨ rv2 = tgt re
      using se-step by (auto simp add : vertices-def)
  thus ?case
  proof (elim disjE conjE, goal-cases)
    case 1
    moreover
    hence subpath (red prb) rv1 res rv2 (subs prb)
      using se-step(1,3,4)
      sub-rel-of.sp-from-old-verts-imp-sp-in-old
      [OF subs-sub-rel-of, of prb re red prb' rv1 rv2 res]
      by auto

  99
ultimately
show ?thesis using se-step
  by (fastforce simp add : finite-RedBlack-def)

next

case 2

hence  \exists \ res'. \ res = res' @ [re]
  \land \ re \notin set res'
  \land \ subpath (red prb) rv1 res' (src re) (subs prb)
  using se-step
  sub-rel-of.sp-to-new-edge-tgt-imp[OF subs-sub-rel-of, of prb re red
  prb' rv1 res]
by auto

thus ?thesis
proof (elim exE conjE)

fix res'

assume res = res' @ [re]
and \ re \notin set res'
and \ subpath (red prb) rv1 res' (src re) (subs prb)

moreover
then obtain c'
  where se-star (confs prb rv1) (trace (ui-es res') (labelling (black prb))) c'
  and se c' (labelling (black prb) (ui-edge re)) c
  using se-step 2 se-star-append-one by auto blast

ultimately
have c' \subseteq (confs prb (src re)) using se-step by fastforce

thus ?thesis
  using se-step \ (rv1 \neq tgt re) 2
  \ se c' (labelling (black prb) (ui-edge re)) c'
  by (auto simp add : se-mono-for-sub)
qed
next

case 3
moreover
have \( rv1 = rv2 \)
proof –

have \((rv1, rv2) \in (subs \ prb')\)
using se-step 3

sub-rel-of.sp-from-tgt-in-extends-is-Nil

[OF sub-sub-rel-of[OF se-step(1)], of re red prb' res rv2]

rb-Nil-sp[OF RedBlack.se-step[OF se-step(1,3)], of rv1 rv2]

by auto

hence \( rv1 \in subsumees (subs prb) \)
using se-step(3) by force

thus \(?thesis\)
using se-step : \( rv1 = tgt \ re \) sub-sub-rel-of[OF se-step(1)]

by (auto simp add : sub-rel-of-def)

qed

ultimately
show \(?thesis\) by simp

next

moreover

hence \( res = [] \)

using se-step

sub-rel-of.sp-from-tgt-in-extends-is-Nil

[OF sub-sub-rel-of[OF se-step(1)], of re red prb' res rv2]

by auto

ultimately
show \(?thesis\) using se-step by (simp add : subsum-rel)

qed

next

\textbf{case} \((mark-step prb rv prb')\) \textbf{thus} \(?case\) by simp

next

\textbf{case} \((subsum-step prb sub prb' rv1 res rv2 c)\)

have \( RB' : RedBlack prb' \) by (rule RedBlack.subsum-step[OF subsum-step(1,3)])
show \(\exists c\)

proof (case-tac \(rv1 = \text{subsumee sub}\))

assume \(rv1 = \text{subsumee sub}\)

hence \(\text{res} = [] \lor \text{subpath (red prb')} (\text{subsumer sub}) \text{res} \text{rv2 (subs prb')}\)

using subsum-step(3,4)

\(\text{wf-sub-rel-of.sp-in-extends-imp1 [ OF subs-wf-sub-rel-of[OF subsum-step(1)]]}, \) of subsumee sub subsumer sub

by simp

thus \(?\text{thesis}\)

proof (elim disjE)

assume \(\text{res} = []\)

hence \(\text{rv1} = \text{rv2} \lor (\text{rv1}, \text{rv2}) \in (\text{subs prb'})\)

using subsum-step rb-Nil-sp[OF RB'] by fast

thus \(?\text{thesis}\)

proof (elim disjE)

assume \(\text{rv1} = \text{rv2}\)

thus \(?\text{thesis}\)

using subsum-step(5) \(<\text{res} = []>\)

by (simp add : subsums-refl)

next

assume \((\text{rv1}, \text{rv2}) \in (\text{subs prb'})\)

thus \(?\text{thesis}\)

using subsum-step(5) \(<\text{res} = []>\)

\(\text{sub-subsumed[OF RB', of (rv1,rv2)]}\)

by simp

qed

next

assume \(\text{subpath (red prb')} (\text{subsumer sub}) \text{res} \text{rv2 (subs prb')}\)

thus \(?\text{thesis}\)

using subsum-step(5)

proof (induct res arbitrary : rv2 c rule : rev-induct, goal-cases)
proof

have \( rv2 = \text{subsumer sub} \)

proof

have \( (\text{subsumer sub}, rv2) \notin \text{subs prb}' \)

proof (intro notI)

assume \( (\text{subsumer sub}, rv2) \in \text{subs prb}' \)

hence \( \text{subsumer sub} \in \text{subsumees (subs prb')} \) by force

moreover

have \( \text{subsumer sub} \in \text{subsumers (subs prb')} \)

using \( \text{subsum-step(3) by force} \)

ultimately

show False

using \( \text{subs-wf-sub-rel[OF RB']} \)

unfolding \( \text{wf-sub-rel-def} \)
by auto

qed

thus \(?thesis using 1(1) \text{rb-Nil-sp[OF RB']} \) by auto

qed

thus \(?case using \text{subsum-step(3) 1(2) } rv1 = \text{subsume sub} \) by simp

next

case \( (2 \text{ re res } rv2 \) c)

hence \( A : \text{subpath (red prb') (subsumer sub) res (src re) (subs prb')} \)

and \( B : \text{subpath (red prb') (src re) [re] (tgt re) (subs prb')} \)

using \( \text{subs-sub-rel-of[OF RB']} \) by (auto simp add : sp-append-one sp-one)

obtain \( c' \)

where \( C : \text{se-star (confs prb' rv1) (trace (ui-es res) (labelling (black prb')))} \)

and \( D : \text{se c' (labelling (black prb') (ui-edge re)) c} \)

using \( 2 \) by (simp add : se-star-append-one) blast

obtain \( c'' \)
where $E : se (confs prb' (src re)) (labelling (black prb') (ui-edge re)) c''$

using subsum-step(6–8)
  \ensuremath{\langle \text{subpath (red prb') (src re) [re] (tgt re) (subs prb')}, \rangle}

$RB' \text{ finite-RedBlack.cf-ex-se-succ[of prb' src re]}

unfolding finite-RedBlack-def
by (simp add : se-star-one fst-of-sp-is-vert) blast

have $c \sqsubseteq c''$
proof –
  have $c' \sqsubseteq \text{confs prb'} (src re)$ using 2(1) $A B C D$ by fast
thus ?thesis using $D E$ se-mono-for-sub by fast
qed

moreover
have $c'' \sqsubseteq \text{confs prb'} (tgt re)$
proof –
  have \text{subpath (red prb) (src re) [re] (tgt re) (subs prb)}
  proof –
    have $\text{src re} \in \text{red-vertices prb'}$
    and $\text{tgt re} \in \text{red-vertices prb'}$
    and $\text{re} \in \text{edges (red prb')}$
    using $B$ by (auto simp add : vertices-def sp-one)
  hence $\text{src re} \in \text{red-vertices prb'}$
  and $\text{tgt re} \in \text{red-vertices prb'}$
  and $\text{re} \in \text{edges (red prb')}$
  using subsum-step(3) by auto
  thus ?thesis
  using \text{subsum-sub-rel-of[OF subsum-step(1)]}
  by (simp add : sp-one)
qed

thus ?thesis
using \text{subsum-step(2,3,6–8) E}
by (simp add : se-star-one)
qed

moreover
have \text{confs prb'} (tgt re) \sqsubseteq \text{confs prb'} rv2
proof –
  have $\text{tgt re} = rv2 \lor (\text{tgt re},rv2) \in \text{subs prb'}$
    using 2(2) \text{rb-sp-append-one[OF RB']} by auto

thus ?thesis
proof (elim disjE)
  assume tgt re = rv2
  thus ?thesis by (simp add : subsums-refl)
next
  assume (tgt re, rv2) ∈ (subs prb')
  thus ?thesis using sub-subsumed RB' by fastforce
qed
qed

ultimately
show ?case using subsums-trans subsums-trans by fast
qed
qed

next

assume rv1 ≠ subsumee sub

hence subpath (red prb) rv1 res rv2 (subs prb) ∨
  (∃ res1 res2. res = res1 @ res2
    ∧ res1 ≠ []
    ∧ subpath (red prb) rv1 res1 (subsumee sub) (subs prb)
    ∧ subpath (red prb') (subsumee sub) res2 rv2 (subs prb'))
  using subsum-step(3,4)
  wf-sub-rel-of.sp-in-extends-imp2 [OF subs-wf-sub-rel-of[OF sub-
  sum-step(1)]],
  of subsumee sub subsumer sub]
  by auto

thus ?thesis
proof (elim disjE exE conjE)

  assume subpath (red prb) rv1 res rv2 (subs prb)
  thus ?thesis using subsum-step by simp

next

fix res1 res2

define t-res1 where t-res1 = trace (ui-es res1) (labelling (black prb'))
define t-res2 where t-res2 = trace (ui-es res2) (labelling (black prb'))

assume res = res1 @ res2

105
and \( \text{res1} \neq [] \)
and \( \text{subpath} (\text{red prb}) \text{rv1 res1} (\text{subsumee sub}) (\text{subs prb}) \)
and \( \text{subpath} (\text{red prb}') (\text{subsumee sub}) \text{res2 rv2} (\text{subs prb}') \)

then obtain \( c1 \ c2 \)
where \( \text{se-star} (\text{confs prb}' \text{rv1}) \text{t-res1 c1} \)
and \( \text{se-star} \ c1 \ \text{t-res2 c} \)
and \( \text{se-star} (\text{confs prb}' (\text{subsumee sub})) \text{t-res2 c2} \)
using \( \text{subsum-step} (1,3,5,6-8) \ \text{RB}' \)
\( \text{finite-RedBlack.ex-se-star-succ[of prb rv1 t-res1]} \)
\( \text{finite-RedBlack.ex-se-star-succ[of prb' subsumee sub t-res2]} \)
unfolding \( \text{finite-RedBlack-def t-res1-def t-res2-def} \)
by (simp add : \text{fst-of-sp-is-vert se-star-append} ) blast

then have \( c \subseteq c2 \)
proof

have \( c1 \subseteq \text{confs prb}' (\text{subsumee sub}) \)
using \( \text{subsum-step} (2,3,6-8) \)
\( \langle \text{subpath} (\text{red prb}) \text{rv1 res1} (\text{subsumee sub}) (\text{subs prb}) \rangle \)
\( \langle \text{se-star} (\text{confs prb}' \text{rv1}) \text{t-res1 c1} \rangle \)
by (auto simp add : \text{t-res1-def t-res2-def})

thus \( \text{thesis} \)
using \( \langle \text{se-star} \ c1 \ \text{t-res2} \ c \rangle \)
\( \langle \text{se-star} (\text{confs prb}' (\text{subsumee sub})) \text{t-res2 c2} \rangle \)
\( \text{se-star-mono-for-sub} \)
by fast

qed

moreover

have \( c2 \subseteq \text{confs prb'} \text{rv2} \)
using \( \langle \text{subpath} (\text{red prb'}) (\text{subsumee sub}) \text{res2 rv2} (\text{subs prb'}) \rangle \)
unfolding \( \text{t-res2-def} \)
proof (induct \text{res2} arbitrary : \text{rv2} c2 \text{ rule : rev-induct, goal-cases})

case (1 \text{rv2} c2)

hence \( \text{subsumee sub} = \text{rv2} \lor (\text{subsumee sub}, \text{rv2}) \subseteq \text{subs prb}' \)
using \( \text{rb-Nil-sp[OF RB']} \) by simp

thus \( \text{?case} \)
proof (elim disjE)
assume \( \text{subsumee \ sub = rv2} \)

thus \( \text{?thesis} \)

using \( 1(2) \) by \( \text{simp add : subsums-refl} \)

next

assume \( (\text{subsumee \ sub, rv2}) \in \text{subs prb'} \)

thus \( \text{?thesis} \)

using \( 1(2) \)

\[ \text{sub-subsumed}\{OF \ RB', of (subsumee \ sub, rv2)} \]

by simp

qed

next

\( \text{case (2 re res2 rv2 c2)} \)

have \( A : \text{subpath (red prb') (subsumee \ sub) res2 (src re) (subs prb')} \)

and \( B : \text{subpath (red prb') (src re) [re] rv2 (subs prb')} \)

using \( 2(2) \) \( \text{subs-wf-sub-rel[OF \ RB'] subs-wf-sub-rel-of[OF \ RB']} \)

by \( \text{simp-all only: wf-sub-rel.sp-append-one} \)

\[ \text{simp add : wf-sub-rel.of.sp-one wf-sub-rel-of-def} \]

obtain \( c3 \)
where \( C : \text{se-star (confs prb' (subsumee \ sub))} \)

\[ \text{trace (ui-es res2) (labelling (black prb')) (c3)} \]

and \( D : \text{se c3 (labelling (black prb') (ui-edge re)) c2} \)

using \( 2(3) \) \( \text{subsum-step(6−8) RB'} \)

\text{finite-RedBlack.ex-se-succ[of prb' src re]}

by \( \text{simp add : se-star-append-one} \) blast

obtain \( c4 \)
where \( E : \text{se (confs prb' (src re)) (labelling (black prb') (ui-edge re)) c4} \)

using \( \text{subsum-step(6−8) RB'} B \)

\text{finite-RedBlack.ex-se-succ[of prb' src re]}

unfolding \( \text{finite-RedBlack-def} \)

by \( \text{simp add : fst-of-sp-is-vert se-star-append} \) blast

have \( c2 \sqsubseteq c4 \)

proof

have \( c3 \sqsubseteq \text{confs prb' (src re) using } 2(1) \ A \ C \) by fast

thus \( \text{?thesis using } D \ E \text{ se-mono-for-sub by fast} \)
qed

moreover
have \( c_4 \subseteq \text{confs prb}'(\text{tgt re}) \)
proof –
  have subpath (\text{red prb}) (\text{src re}) [\text{re}] (\text{tgt re}) (\text{subs prb})
  proof –
    have src re \( \in \text{red-vertices prb}' \)
    and tgt re \( \in \text{red-vertices prb}' \)
    and re \( \in \text{edges (red prb)} \)
    using \( B \) by (auto simp add : vertices-def sp-one)
  hence src re \( \in \text{red-vertices prb} \)
  and tgt re \( \in \text{red-vertices prb} \)
  and re \( \in \text{edges (red prb)} \)
  using subsum-step(3) by auto
  thus \( \vdash \)thesis
    using subs-sub-rel-of[OF subsum-step(1)]
    by (simp add : sp-one)
  qed

thus \( \vdash \)thesis
  using subsum-step(2,3,6–8) \( E \)
  by (simp add : se-star-one)
  qed

moreover
have \( \text{confs prb}'(\text{tgt re}) \subseteq \text{confs prb}' \text{rv2} \)
proof –
  have tgt re = \text{rv2} \lor (\text{tgt re}, \text{rv2}) \( \in \) (\text{subs prb}')
  using subsum-step 2 rb-sp-append-one[OF RB', of subsume sub res2 re]
    by (auto simp add : vertices-def subpath-def)
  thus \( \vdash \)thesis
  proof (elim disjE)
    assume tgt re = \text{rv2}
    thus \( \vdash \)thesis by (simp add : subsums-refl)
  next
    assume (\text{tgt re}, \text{rv2}) \( \in \) (\text{subs prb}')
    thus \( \vdash \)thesis
      using sub-subsumed RB'
      by fastforce
  qed
ultimately
show ?case using subsums-trans subsums-trans by fast
qed

ultimately
show ?thesis by (rule subsums-trans)
qed

next
case (abstract-step prb rv c a prb' rv1 res rv2 c)

show ?case
proof (case-tac rv1 = rv, goal-cases)

case 1

moreover
hence res = []
  using abstract-step
  sp-from-de-empty[of rv1 subs prb red prb res rv2]
  by simp

moreover
have rv2 = rv
proof –
  have rv1 = rv2 ∨ (rv1, rv2) ∈ (subs prb)
    using abstract-step (res = []),
    rb-Nil-sp[OF RedBlack.abstract-step[OF abstract-step(1,3)]]
    by simp

moreover
have (rv1, rv2) ∉ (subs prb)
  using abstract-step 1
  unfolding Ball-def subsumees-conv
  by (intro notI) blast

ultimately
show ?thesis using 1 by simp
qed

ultimately
show ?thesis using abstract-step(5) by (simp add : subsums-refl)
next

case 2

show \(?thesis
proof \((\text{case-tac } \text{rv2} = \text{rv})\)

assume \(\text{rv2} = \text{rv}\)

hence \(\text{confs prb rv2} \subseteq \text{confs prb'} \text{rv2}\)
  using abstract-step by (simp add : abstract-def)

moreover
  have \(c \subseteq \text{confs prb rv2}\)
  using abstract-step 2 by auto

ultimately
  show \(?thesis using subsums-trans by fast
next
  assume \(\text{rv2} \neq \text{rv}\) thus \(?thesis using abstract-step 2 by simp
qed

qed

next

case strengthen-step thus \(?case by simp
qed

12.6 Properties about marking.

A configuration which is indeed satisfiable can not be marked.

lemma sat-not-marked :
  assumes RedBlack prb
  assumes \(\text{rv} \in \text{red-vertices prb}\)
  assumes sat \((\text{confs prb rv})\)
  shows \(?\neg \text{marked prb rv}\)
using asms
proof \((\text{induct prb arbitrary : rv})\)
  case base thus \(?case by simp
next
  case \(\text{se-step prb re c prb'}\)

  hence \(\text{rv} \in \text{red-vertices prb} \lor \text{rv} = \text{tgt re}\) by (auto simp add : vertices-def)
thus ?case
proof (elim disjE, goal-cases)
  case 1
  moreover
  hence rv ≠ tgt re using se-step(3) by (auto simp add : vertices-def)
  ultimately
  show ?thesis using se-step by (elim conjE) auto
next
  case 2
  moreover
  hence rv ≠ tgt re using se-step(3,5) se-sat-imp-sat by auto

ultimately
  show ?thesis using se-step by (elim conjE) auto
qed
next
  case (mark-step prb rv prb')
  moreover
  hence rv ≠ rv' and (rv,rv') ∉ subs prb
  using sub-subsumed[OF mark-step(1), of (rv,rv')] unsat-subs-unsat by auto

ultimately
  show ?case by auto
next
  case subsum-step thus ?case by auto

next
  case (abstract-step prb rv' c_a prb') thus ?case by (case-tac rv' = rv) simp+
next
  case strengthen-step thus ?case by simp
qed

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

lemma
  assumes RedBlack prb
  assumes rv ∈ red-vertices prb
  assumes marked prb rv
  shows ¬ sat (confs prb rv)
using assms
proof (induct prb arbitrary : rv)
  case base thus ?case by simp
  111
next
case (se-step prb re c prb')

hence \( rv \in \text{red-vertices prb} \lor rv = \text{tgt re} \) by (auto simp add : vertices-def)

thus \(?case\)
proof (elim disjE, goal-cases)
case 1

moreover
hence \( rv \neq \text{tgt re} \) using se-step(3) by auto
hence marked prb rv using se-step by auto

ultimately
have \( \neg \text{sat (confs prb rv)} \) by (rule se-step(2))

thus \(?thesis \) using se-step(3) \( \langle rv \neq \text{tgt re} \rangle \) by auto
next
case 2

moreover
hence marked prb (src re) using se-step(3,5) by auto

ultimately
have \( \neg \text{sat (confs prb (src re))} \) using se-step(2,3) by auto

thus \(?thesis \) using se-step(3) \( \langle rv = \text{tgt re} \rangle \) unsat-imp-se-unsat by (elim conjE)
auto
qed
next
case (mark-step prb rv' prb') thus \(?case\) by (case-tac rv' = rv) auto
next
case subsum-step thus \(?case\) by simp
next
case (abstract-step - rv' -) thus \(?case\) by (case-tac rv' = rv) simp+
next
case strengthen-step thus \(?case\) by simp
qed

Red vertices involved in subsumptions are not marked.

lemma subsumee-not-marked :
assumes RedBlack prb
assumes sub \( \in \) subs prb
shows \( \neg \) marked prb (subsume sub)
using assms
proof (induct prb)
  case base thus ?case by simp
next
  case (se-step prb re c prb')
moreover
hence subsume sub \( \neq \) tgt re
using subs-wf-sub-rel-of[OF se-step(1)]
by (elim conjE, auto simp add : wf-sub-rel-of-def sub-rel-of-def)
ultimately
show ?case by auto
next
  case mark-step thus ?case by auto
next
  case subsum-step thus ?case by auto
next
  case abstract-step thus ?case by auto
next
  case strengthen-step thus ?case by simp
qed

lemma subsumer-not-marked :
  assumes RedBlack prb
  assumes sub \( \in \) subs prb
  shows \( \neg \) marked prb (subsumer sub)
using assms
proof (induct prb)
  case base thus ?case by simp
next
  case (se-step prb re c prb')
moreover
hence subsumer sub \( \neq \) tgt re
using subs-wf-sub-rel-of[OF se-step(t)]
by (elim conjE, auto simp add : wf-sub-rel-of-def sub-rel-of-def)
ultimately
show ?case by auto
next
case (mark-step prb rv prb') thus ?case by auto
next
case (subsum-step prb sub' prb') thus ?case by auto
next
case abstract-step thus ?case by simp
next
case strengthen-step thus ?case by simp
qed

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)
using assms
proof (induct prb arbitrary : re)
case base thus ?case by simp
next
case se-step thus ?case by (force simp add : vertices-def image-def)
next
case (mark-step prb rv prb' re) thus ?case by (case-tac tgt re = rv) auto
next
case subsum-step thus ?case by simp
next
case abstract-step thus ?case by simp
next
case strengthen-step thus ?case by simp
qed

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
using assms
proof (induct res arbitrary : rv1)
case Nil
hence \( rv1 = rv2 \lor (rv1, rv2) \in \text{subsp prb by (simp add : rb-Nil-sp)} \)

thus ?case
proof (elim disjE, goal-cases)
case 1 thus ?case using Nil by simp
next
case 2 show ?case using Nil subsume-not-marked[OF Nil(1) 2] by simp
qed
next
case (Cons re res)

thus ?case
unfolding rb-sp-Cons[OF Cons(2), of rv1 re res rv2]
proof (elim conjE disjE, goal-cases)
case 1

moreover hence \( \neg \text{marked prb (tgt re)} \) by simp

moreover have \( re \in \text{edges (red prb)} \) using Cons(3) rb-sp-Cons[OF Cons(2), of rv1 re res rv2] by fast

ultimately show ?thesis using tgt-not-marked-imp[OF Cons(2)] by fast
next
case 2 thus ?thesis using subsume-not-marked[OF Cons(2)] by fastforce
qed
qed

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 ::**

\[
\text{fringe prb} \equiv \{ rv \in \text{red-vertices prb}. \quad \begin{align*}
rv & \notin \subsume (\subm prb) \\
\neg \text{marked prb} \; rv & \\
\text{ui-edge } \; \text{out-edges } (\text{red prb}) \; rv & \subset \text{out-edges } (\text{black prb}) \; (\text{fst \; rv})
\end{align*}\}
\]

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 \[\text{edges } (\text{red prb}) = \{\}\]
assumes \[\text{subs prb} = \{\}\]
assumes \[\text{marked prb} = (\lambda \; rv. \; \text{False})\]
assumes \[\text{out-edges } (\text{black prb}) \; (\text{fst } (\text{root } \; \text{red prb})) \neq \{\}\]
shows \[\text{fringe prb} = \{\text{root } \; \text{red prb}\}\]
using \text{assms by (auto simp add : fringe-def vertices-def)}

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 \[\text{sub-rel-of } (\text{red prb}) \; (\text{subs prb})\]
assumes \[\text{se-extends prb re c'} \; prb'\]
assumes \[\neg \text{marked prb} \; (\text{src re})\]
assumes \[\text{ui-edge } \; \text{(out-edges } (\text{red prb'}) \; (\text{src re})) \subset \text{out-edges } (\text{black prb}) \; (\text{fst } (\text{src re}))\]
assumes \[\text{out-edges } (\text{black prb}) \; (\text{fst } (\text{tgt re})) \neq \{\}\]
shows \[\text{fringe prb'} = \text{fringe prb} \cup \{\text{tgt re}\}\]
unfolding \text{set-eq-iff Un-iff singleton-iff}
proof (intro allI iffI, goal-cases)
  case (1 rv)

  moreover
  hence rv ∈ red-vertices prb ∨ rv = tgt re
  using assms(2) by (auto simp add : fringe-def vertices-def)

ultimately
  show ?case using assms(2) by (auto simp add : fringe-def)
next
  case (2 rv)

  hence rv ∈ red-vertices prb' using assms(2) by (auto simp add : fringe-def vertices-def)

moreover
  have rv ∉ subsumees (subs prb')
  using 2
  proof (elim disjE)
    assume rv ∈ fringe prb thus ?thesis using assms(2) by (auto simp add : fringe-def)
  next
    assume rv = tgt re thus ?thesis
    using assms(1,2) unfolding sub-rel-of-def by force
  qed

moreover
  have ui-edge (out-edges (red prb') rv) ⊂ out-edges (black prb') (fst rv)
  using 2
  proof (elim disjE)
    assume rv ∈ fringe prb

    thus ?thesis
    proof (case-tac rv = src re)
      assume rv = src re thus ?thesis using assms(2,4) by auto
    next
      assume rv ≠ src re thus ?thesis
      using assms(2) (rv ∈ fringe prb)
      by (auto simp add : fringe-def)
    qed
  next
    assume rv = tgt re thus ?thesis
    using assms(2,5) extends-tgt-out-edges[of re red prb red prb'] by (elim conjE)
  auto
  qed
moreover
have ¬ marked prb' rv
using 2
proof (elim disjE, goal-cases)
case 1

moreover
hence rv ≠ tgt re using assms(2) by (auto simp add : fringe-def)

ultimately
show ?thesis using assms(2) by (auto simp add : fringe-def)
next
case 2 thus ?thesis using assms(2,3) by auto
qed

ultimately
show ?case by (simp add : fringe-def)
qed

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}
unfolding set-eq_iff Un iff singleton iff Diff iff
proof (intro allI iffI, goal-cases)
case (1 rv)

hence rv ∈ red-vertices prb ∨ rv = tgt re
and rv ≠ src re
using assms(2,3,4,5) by (auto simp add : fringe-def vertices-def)

with 1 show ?case using assms(2) by (auto simp add : fringe-def)

next
case (2 rv)
hence \( rv \in \text{red-vertices prb'} \) using \( \text{assms(2)} \) by (auto simp add : fringe-def vertices-def)

moreover
have \( rv \notin \text{subsumees (subs prb')} \)
using 2
proof (elim disjE)
  assume \( rv \in \text{fringe prb} \land rv \neq \text{src re} \)
  thus \( ?\text{thesis using assms(2)} \) by (auto simp add : fringe-def)
next
  assume \( rv = \text{tgt re} \) thus \( ?\text{thesis} \)
  using \( \text{assms(1,2) unfolding sub-rel-of-def by fastforce} \)
qed

moreover
have \( \text{ui-edge ' (out-edges (red prb') rv) } \subset \text{out-edges (black prb') (fst rv)} \)
using 2
proof (elim disjE)
  assume \( rv \in \text{fringe prb} \land rv \neq \text{src re} \) thus \( ?\text{thesis} \)
  using \( \text{assms(2)} \) by (auto simp add : fringe-def)
next
  assume \( rv = \text{tgt re} \) thus \( ?\text{thesis} \)
  using \( \text{assms(2,5) extends-tgt-out-edges[of re red prb red prb']} \) by (elim conjE)
auto
qed

moreover
have \( \neg \text{marked prb'} rv \)
using 2
proof (elim disjE, goal-cases)
  case 1

  moreover
  hence \( rv \neq \text{tgt re} \) using \( \text{assms by (auto simp add : fringe-def)} \)

ultimately
show \( ?\text{thesis} \)
using \( \text{assms 1 by (auto simp add : fringe-def)} \)
next
  case 2 thus \( ?\text{thesis using assms by auto} \)
qed

ultimately
show \( ?\text{case by (simp add : fringe-def)} \)
qed
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

**unfolding** set-eq-iff Un-iff singleton-iff

**proof** (intro allI iffI, goal-cases)
- case (1 rv)

**thus** ?case

**unfolding** fringe-def mem-Collect-eq

**using** assms

**proof** (intro conjI, goal-cases)
- case 1 **thus** ?case by (auto simp add : fringe-def vertices-def)
- next
- case 2 **thus** ?case by auto
- next
- case 3

moreover

**hence** rv ≠ tgt re by auto

ultimately

**show** ?case by auto
next
- case 4 **thus** ?case by auto
qed
next
- case (2 rv)

**thus** ?case unfolding fringe-def mem-Collect-eq

**using** assms

**proof** (intro conjI, goal-cases)
- case 1 **thus** ?case by (auto simp add : vertices-def)
- next
- case 2 **thus** ?case by auto
next
- case 3
moreover

**hence** rv ≠ tgt re by auto
ultimately

**show** ?case by auto
next
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`

**unfolding** `set-eq-iff Un-iff singleton-iff`

**proof** (intro allI iffI, goal-cases)

- **case** (1 rv)
  - **thus** ?case using assms(1,3)
  - **unfolding** `fringe-def mem-Collect-eq`
  - **proof** (intro conjI, goal-cases)
    - **case** 1 thus ?case by (auto simp add : fringe-def vertices-def)
    - **next**
    - **case** 2 thus ?case by (auto simp add : fringe-def)
    - **next**
    - **case** 3 thus ?case by (case-tac rv = tgt re) (auto simp add : fringe-def)
    - **next**
    - **case** 4 thus ?case by (auto simp add : fringe-def)
  - **qed**

- **next**
  - **case** (2 rv)

- **moreover**
  - **hence** `rv ∈ red-vertices prb'`
  - **and** `rv ≠ tgt re`
  - **using** assms(1) by (auto simp add : fringe-def vertices-def)

- **moreover**
  - **have** `ui-edge (out-edges (red prb') rv) ⊂ out-edges (black prb) (fst rv)`
  - **proof** (case-tac rv = src re)
    - **assume** `rv = src re` thus ?thesis using assms(2) by simp
    - **next**
    - **assume** `rv ≠ src re`
    - **thus** ?thesis using assms(1) 2
      - by (auto simp add : fringe-def)
  - **qed**
ultimately
show ?case using assms(1) by (auto simp add : fringe-def)
qed

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}
unfolding set-eq-iff Un-iff singleton-iff Diff-iff
proof (intro allI iffI, goal-cases)
case (1 rv)

moreover
have rv ∈ red-vertices prb and rv ≠ src re
using 1 assms by (auto simp add : fringe-def vertices-def)

moreover
have ¬ marked prb rv
proof (intro notI)
  assume marked prb rv

  have marked prb' rv
  proof –
    have rv ≠ tgt re using assms(1) (rv ∈ red-vertices prb) by auto
    thus ?thesis using assms(1) (marked prb rv) by auto
  qed

  thus False using 1 by (auto simp add : fringe-def)
  qed

ultimately
show ?case using assms(1) by (auto simp add : fringe-def)

next
case (2 rv)

hence rv ∈ red-vertices prb' using assms(1) by (auto simp add : fringe-def
vertices-def

moreover
have \( rv \notin \text{subsumees} (\text{subs prb}') \) using 2 assms(1) by (auto simp add : fringe-def)

moreover
have ui-edge ' (out-edges (red prb') rv) \( \subseteq \) out-edges (black prb') (fst rv)
using 2 assms(1) by (auto simp add : fringe-def)

moreover
have \( \neg \text{marked prb}' rv \)
proof –
  have \( rv \neq \text{tgt re} \) using assms(1) 2 by (auto simp add : fringe-def)

  thus ?thesis using assms(1) 2 by (auto simp add : fringe-def)
qed

ultimately
show ?case by (simp add : fringe-def)
qed

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 – \{\text{subsumee sub}\}
using assms by (auto simp add : fringe-def)

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 \Rightarrow (\'vert \times \text{nat}) \Rightarrow \text{\'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** —
show ?thesis
unfolding set-eq_iff
proof (intro allI iffI)

fix bes
assume bes ∈ RedBlack-subpaths-from prb (root (red prb))
thus bes ∈ Graph.paths (black prb)
unfolding RedBlack-subpaths-from-def Un-iff
proof (elim disjE exE conjE, goal-cases)
case 1

hence bes = [] using assms by (auto simp add: subpath-def)

thus ?thesis
by (auto simp add : Graph.subpath-def vertices-def)
next
  case 2

  then obtain $res1 \ bes2 \ rv$ where $bes = ui-es \ res1 \ & \ bes2$
  and $rv \in fringe \ prb$
  and subpath $(red \ prb) (root \ (red \ prb)) \ res1 \ rv \ (subs \ prb)$
  and $Graph.\ subpath-from \ (black \ prb) \ (fst \ rv) \ bes2$
  by blast

  moreover
  hence $res1 = []$ using assms by (simp add : subpath-def)

  ultimately
  show $\vdash thesis$ using assms $\langle rv \in fringe \ prb \rangle$ by (simp add : fringe-def vertices-def)
  qed

next
  fix $bes$
  assume $bes \in Graph.\ paths \ (black \ prb)$
  show $bes \in RedBlack-subpaths-from \ prb \ (root \ (red \ prb))$
  proof (case-tac out-edges $(black \ prb) \ (init \ (black \ prb)) = \{\})$
    assume $out-edges \ (black \ prb) \ (init \ (black \ prb)) = \{\}$
    show $\vdash thesis$
    unfolding RedBlack-subpaths-from-def Un-iff mem-Collect-eq
    apply (intro disjI1)
    apply (rule-tac $?x=[\ ]$ in exI)
    apply (intro conjI)
    apply (rule-tac $?x=root \ (red \ prb)$ in exI)
    proof (intro conjI)
      show subpath $(red \ prb) \ (root \ (red \ prb)) \ [\ ] \ (root \ (red \ prb)) \ (subs \ prb)$
      using assms(3) by (simp add : sub-rel-of-def subpath-def vertices-def)
      next
      show $\neg \ marked \ prb \ (root \ (red \ prb))$ using assms(5) by simp
      next
      show $bes = ui-es \ []$
      using $\langle bes \in Graph.\ paths \ (black \ prb) \rangle$
      $\langle out-edges \ (black \ prb) \ (init \ (black \ prb)) = \{\}\rangle$
      by (cases bes) (auto simp add : Graph.sp-Cons)
    qed
    next
    show $\vdash thesis$
    unfolding RedBlack-subpaths-from-def Un-iff mem-Collect-eq
    proof (intro disjI2, rule-tac $?x=[\ ]$ in exI, rule-tac $?x=\text{bes}$ in exI, intro conjI, goal-cases)
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)

**Unfolding** subset-iff mem-Collect-eq RedBlack-subpaths-from-def Un-iff image-def Bex-def
proof (intro allI impI, elim disjE exE conjE, goal-cases)
case (1 bes res rv) thus ?case using assms red-sp-imp-black-sp by blast
next
case (2 bes res1 bes2 rv1 bv2) thus ?case
using red-sp-imp-black-sp[of assms, of rv res1 rv1]
by (rule-tac ?x=bv2 in exI) (auto simp add : Graph.sp-append)
qed

lemma RedBlack-paths-are-black-paths :
assumes RedBlack prb
shows RedBlack-paths prb ⊆ Graph.paths (black prb)
using assms
RedBlack-subpaths-are-black-subpaths[of prb root (red prb)]
consistent-roots[of prb]
by simp

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
using assms finite-RedBlack
proof (induct prb arbitrary : rv)

case (base prb rv)

moreover
hence $rv = \text{root} (\text{red prb})$ by (simp add : vertices-def)

moreover
hence feasible-subpaths-from (black prb) (confs prb rv) (fst rv)
= feasible-paths (black prb) (confs prb (root (red prb)))
using base by simp

moreover
have out-edges (black prb) (fst (root (red prb))) = {} ∨
ui-edge 'out-edges(red prb)(root (red prb)) ⊂ out-edges(black prb)(fst (root (red prb)))

  using base by auto

ultimately

show ?case
  using finite-RedBlack.base-RedBlack-paths[of prb]
  by (auto simp only : finite-RedBlack-def)

next


unfolding Un-iff image-def Bex-def mem-Collect-eq
proof (elim disjE exE conjE)

fix res rv'

assume bes = ui-es res
and subpath (red prb) rv res rv' (subs prb)
and ¬ marked prb rv'

moreover
hence ¬ marked prb' rv'
using se-step(3) lst-of-sp-is-vert[of red prb rv res rv' subs prb]
by (elim conjE) auto

ultimately
show ?thesis
using se-step(3) sp-in-extends-w-sub[of red prb red prb' rv res rv' subs]
unfolding RedBlack-subpaths-from-def Un-iff image-def Bex-def mem-Collect-eq
by (intro disjI1, rule-tac ?x=res in exI, intro conjI)
(rule-tac ?x=rv' in exI, auto)

next

fix res1 bes2 rv1 bl

assume A : bes = ui-es res1 @ bes2
and B : rv1 ∈ fringe prb
and C : subpath (red prb) rv res1 rv1 (subs prb)
and E : ¬ (∃ res21 bes22. bes2 = ui-es res21 @ bes22
∧ res21 ≠ []
∧ subpath-from (red prb) rv1 res21 (subs prb))
and F : Graph.subpath (black prb) (fst rv1) bes2 bl

hence rv1 ≠ tgt re using se-step by (auto simp add : fringe-def)

show ?thesis
proof (case-tac rv1 = src re)

assume rv1 = src re

show ?thesis
proof (case-tac bes2 = [])
assume bes2 = []

show ?thesis

  unfolding RedBlack-subpaths-from-def Un-iff image-def Bex-def mem-Collect-eq
  apply (intro disjI1)
  apply (rule-tac ?x=resI in exI)
  apply (intro conjI)
  apply (rule-tac ?x=rv1 in exI)
  apply (intro conjI)
  proof –
  show subpath (red prb' rv res1 rv1 (subs prb'))
    using se-step(3) C by (auto simp add : sp-in-extends-wsubs)
  next
  have rv1 ≠ tgt re using se-step(3) (rv1 = src re) by auto
  thus ¬ marked prb' rv1 using se-step(3) B by (auto simp add :
    fringe-def)
  next
  show bes = ui-es res1 using A (bes2 = []); by simp
  qed

next

assume bes2 ≠ []
then obtain be bes2' where bes2 = be # bes2' unfolding neq-Nil-conv
by blast
show ?thesis
proof (case-tac be = ui-edge re)

  assume be = ui-edge re

  show ?thesis
  proof (case-tac out-edges (black prb) (fst (tgt re)) = {})

    assume out-edges (black prb) (fst (tgt re)) = {} 
    show ?thesis

      unfolding RedBlack-subpaths-from-def Un-iff image-def Bex-def mem-Collect-eq
      apply (intro disjI1)
      apply (rule-tac ?x=resI @ [re] in exI)
      apply (intro conjI)
      apply (rule-tac ?x=tgt re in exI)
      proof (intro conjI)
      show subpath (red prb') rv (resI @ [re]) (tgt re) (subs prb')

130
using se-step(3) (rv1 = src re) C
  sp-in-extends-w-vars[of re red prb red prb' rv res1 rv1 subs
prb]
  rb-sp-append-one[OF RB', of rv res1 re tgt re]
by auto

next
show ~ marked prb' (tgt re)
using se-step(3) (rv1 = src re) B
by (auto simp add : fringe-def)

next
have bes2' = []
using F ⟨bes2 = be # bes2'⟩
  ⟨be = ui-edge re⟩ ⟨out-edges (black prb) (fst (tgt re)) = {}⟩
by (cases bes2') (auto simp add: Graph.sp-Cons)

thus bes = ui-es (res1 @ [re])
using ⟨bes = ui-es res1 @ bes2⟩ ⟨bes2 = be # bes2'⟩ ⟨be = ui-edge re⟩ by simp
qed

next

assume out-edges (black prb) (fst (tgt re)) ≠ {}
show ?thesis

unfolding RedBlack-subpaths-from-def Un-iff mem-Collect-eq
apply (intro disjI2)
apply (rule-tac ?x=res1@[re] in exI)
apply (rule-tac ?x=bes2' in exI)
proof (intro conjI, goal-cases)
  show bes = ui-es (res1 @ [re]) @ bes2'
  using ⟨bes = ui-es res1 @ bes2⟩ ⟨bes2 = be # bes2'⟩ ⟨be = ui-edge re⟩ by simp
next

  case 2 show ?case
  proof (rule-tac ?x=tgt re in exI, intro conjI)
    have ~ marked prb (src re)
    using B ⟨rv1 = src re⟩ by (simp add : fringe-def)
    thus tgt re ∈ fringe prb'
    using se-step(3) out-edges (black prb) (fst (tgt re)) ≠ {},
    seE-fringe1 [OF subs-sub-rel-of[OF se-step(1)] se-step(3)]
    seE-fringe4 [OF subs-sub-rel-of[OF se-step(1)] se-step(3)]
    by auto
next

131
show subpath (red prb') rv (res1 @ [re]) (tgt re) (subs prb')
using se-step(3) rv1 = src re C
sp-in-extends-w-sub[s of re red prb red prb'
rv res1 rv1 subs prb]
rb-sp-append-one[OF RB', of rv res1 re tgt re]
by auto

next
show ¬ (∃ res21 bes22. bes2' = ui-es res21 @ bes22
∧ res21 ≠ []
∧ subpath-from (red prb') (tgt re) res21 (subs prb'))
proof (intro notI, elim exE conjE)
fix res21 bes22 rv2
assume bes2' = ui-es res21 @ bes22
and res21 ≠ []
and subpath (red prb') (tgt re) res21 rv2 (subs prb')
thus False
using se-step(3)
sub-rel-of.sp-from-tgt-in-extends-is-Nil
[OF subs-sub-rel-of[OF se-step(1)], of re red prb' res21 rv2]
by auto
qed
next
show Graph.subpath-from (black prb') (fst (tgt re)) bes2'
using se-step(3) F (bes2 = be # bes2' ∧ be = ui-edge re)
by (auto simp add : Graph.sp-Cons)
qed
qed
qed

next

assume be ≠ ui-edge re

show ?thesis
unfolding RedBlack-subpaths-from-def Un-iff mem-Collect-eq
unfolding RedBlack-subpaths-from-def Un-iff mem-Collect-eq
apply (intro disjI2)
apply (rule-tac ?x=res1 in exI)
apply (rule-tac ?x=bes2 in exI)
apply (intro conjI)
apply (rule (bes = ui-es res1 @ bes2))
apply (rule-tac ?x(rv1 in exI)
proof (intro conjI)
show \( rv1 \in {\text{fringe prb}}' \)

unfolding fringe-def mem-Collect-eq
proof (intro conjI)
  show \( rv1 \in {\text{red-vertices prb}}' \)
using se-step(3) \( B \) by (auto simp add : fringe-def vertices-def)
next
  show \( rv1 \notin {\text{subsumees (subs prb)}}' \)
using se-step(3) \( B \) by (auto simp add : fringe-def)
next
  show \( \neg \text{marked prb}' rv1 \)
using \( B \) se-step(3) \( rv1 \neq \text{tgt re} \) \( rv1 = \text{src re} \)
by (auto simp add : fringe-def)
next
  have \( be \notin {\text{ui-edge ' out-edges (red prb)' rv1}} \)
proof (intro notI)
    assume \( be \in {\text{ui-edge ' out-edges (red prb)' rv1}} \)
    then obtain \( re' \ where be = ui-edge re' \)
    and \( re' \in {\text{out-edges (red prb)' rv1}} \)
by blast

show \( False \)
using \( E \)
apply (elim notE)
apply (rule-tac ?x=\[re\]' \in exI)
apply (rule-tac ?x=bes2' \in exI)
proof (intro conjI)
  show \( \text{bes2} = ui-es \[re\]' @ bes2' \)
using \( \text{bes2} = be \neq bes2' \) \( be = ui-edge re' \)
by simp
next
  show \( \[re\]' \neq [] \)
by simp
next
  have \( re' \in {\text{edges (red prb)}}' rv1 \)
using se-step(3) \( rv1 = \text{src re} \) \( re' \in {\text{out-edges (red prb)' rv1}} \)
  \( be \neq ui-edge re\) \( be = ui-edge re' \)
by (auto simp add : vertices-def)
thus subpath-from (red prb) \( rv1 \ [re]' \) (subs prb)
using \( \text{re'} \in {\text{out-edges (red prb)' rv1}} \)
subssub-rel-of[OF se-step(1)]
by (rule-tac ?x=tgt re' \in exI)
  (simp add : rb-sp-one[OF se-step(1)])
qed

moreover
have \( be \in \text{out-edges} (\text{black prb} (\text{fst rv1}) \)
using \( F (\text{bes2} = be \# \text{bes2'} \) by (simp add : Graph.sp-Cons)

ultimately
show \( \text{ui-edge} \ ' \text{out-edges} (\text{red prb'})) \text{rv1} \subset \text{out-edges} (\text{black prb'}) (\text{fst rv1}) \)
using se-step(3) red-OA-subset-black-OA[OF RB', of rv1]
by auto

qed

next
show \( \text{subpath} (\text{red prb'}) \text{rv res1 rv1} (\text{subs prb'}) \)
using se-step(3) C by (auto simp add : sp-in-extends-w-subs)

next
show \( \neg (\exists \text{res21} \text{bes22}. \text{bes2} = \text{ui-es res21} @ \text{bes22} \)
\& \text{res21} \neq [] \)
\& \text{subpath-from} (\text{red prb'}) \text{rv1 res21} (\text{subs prb'}) 
apply (intro notI)
apply (elim exE conjE)
proof –
fix \text{res21} \text{bes22} \text{rv3}
assume \text{bes2} = \text{ui-es res21} @ \text{bes22}
and \text{res21} \neq []
and \text{subpath} (\text{red prb'}) \text{rv1 res21} \text{rv3} (\text{subs prb'})
moreover
then obtain \text{re'} \text{res21'} \text{where res21} = \text{re'} \# \text{res21'}
and \text{be} = \text{ui-edge re'}
using \( \text{bes2} = \text{be \# \text{bes2'}} \) unfolding neq-Nil-conv by (elim exE) simp

ultimately
have \text{re'} \in \text{edges} (\text{red prb'}) by (simp add : sp-Cons)
moreover
have \text{re'} \notin \text{edges} (\text{red prb})
using \( E \)
apply (intro notI)
apply (elim notE)
apply (rule-tac \( ?x=[\text{re'}] \) in exI)
apply (rule-tac \( ?x=\text{bes2'} \) in exI)
proof (intro conjI)
show \text{bes2} = \text{ui-es [re']} @ \text{bes2'}
using \( \text{bes2} = \text{be \# \text{bes2'}} \) \text{be} = \text{ui-edge re'} by simp
next

134
show [re'] ≠ [] by simp

next
assume re' ∈ edges (red prb)
thus subpath-from (red prb) rv1 [re'] (subs prb)
  using subs-sub-rel-of[OF se-step(1)]
  {subpath (red prb') rv1 res21 rv3 (subs prb')
   res21 = re' ≠ res21'}
apply (rule-tac ?x=tgt re' in ex1)
apply (simp add: rb-sp-Cons[OF RB'])
apply (simp add: rb-sp-one[OF se-step(1)])
using se-step(3) by auto
qed

ultimately
show False
  using se-step(3) \(\langle be \neq ui\text{-}edge re, be = ui\text{-}edge re'\rangle\) by auto
qed

next
show Graph.subpath-from (black prb') (fst rv1) bes2
  using se-step(3) F by auto
qed

next
assume rv1 ≠ src re

show \(?thesis
unfolding RedBlack-subpaths-from-def Un-iff mem-Collect-eq
apply (intro disjI2)
apply (rule-tac \(?x=res1 in ex1)
apply (rule-tac \(?x=bes2 in ex1)
apply (intro conjI, goal-cases)
proof –
show bes = ui-es res1 @ bes2 by (rule \(\langle bes = ui-es res1 @ bes2\rangle\))
next
  case 2 show \(?case
apply (rule-tac \(?x=rv1 in ex1)
proof (intro conjI, goal-cases)
show rv1 ∈ fringe prb'
  using se-step(3) B \(\langle rv1 \neq src re, rv1 \neq tgt re,\rangle\)
  seE-fringe1[OF subs-sub-rel-of[OF se-step(1)] se-step(3)]
  seE-fringe2[OF se-step(3)]
  seE-fringe3[OF se-step(3)]
  seE-fringe4[OF subs-sub-rel-of[OF se-step(1)] se-step(3)]
seE-fringe5[OF se-step(3)]
apply (case-tac marked prb (src re))
apply simp
apply (case-tac ui-edge ' out-edges (red prb') (src re) ⊂ out-edges (black prb) (fst (src re)))]
apply (case-tac out-edges (black prb) (fst (tgt re)) = { })
apply simp
apply simp
apply (case-tac out-edges (black prb) (fst (tgt re)) = { })
apply simp
apply simp
done
next
show subpath (red prb') rv res1 rv1 (subs prb')
using se-step(3) C by (auto simp add :sp-in-extends-w-sub)
next
show ¬ (∃ res21 bes22. bes2 = ui-es res21 @ bes22
∧ res21 ≠ [] ∧ subpath-from (red prb') rv1 res21 (subs prb'))
proof (intro notI, elim exE conjE)
fix res21 bes22 rv2
assume bes2 = ui-es res21 @ bes22
and res21 ≠ []
and subpath (red prb') rv1 res21 rv2 (subs prb')
then obtain re' res21' where res21 = re' # res21'
using (res21 ≠ []) unfolding neq-Nil-conv by blast
have rv1 = src re' ∨ (rv1, src re') ∈ subs prb
and re' ∈ edges (red prb')
using se-step(3) rb-sp-Cons[OF RB']
⟨subpath (red prb') rv1 res21 rv2 (subs prb')⟩ ⟨res21 = re' # res21'⟩
by auto
moreover
have re' ∈ edges (red prb)
proof –
have re' ≠ re
using ⟨rv1 = src re' ∨ (rv1, src re') ∈ subs prb⟩
proof (elim disjE, goal-cases)
case 1 thus ?thesis using ⟨rv1 ≠ src re⟩ by auto
next
case 2 thus ?case
using B unfolding fringe-def subsumees-conv

by fast 
  qed
  thus thesis using se-step(3) \(re' \in \text{edges }\text{(red prb)}\) by simp 
  qed

show False 
  using E 
  apply (elim notE) 
  apply (rule-tac \(?x = [re'] in exI\)) 
  apply (rule-tac \(?x = \text{ui-es res21' @ bes22 in exI}\)) 
  proof (intro conjI)
  show bes2 = ui-es [re'] @ ui-es res21' @ bes22 
    using \(\text{bes2} = \text{ui-es res21 @ bes22}\) \(\text{res21} = re' \neq\)
  next 
  show \([re'] \neq []\) by simp 
  next 
  show subpath-from (red prb) rv1 [re'] (subs prb) 
    using se-step(1) 
    \(\text{rv1} = \text{src }re' \lor (\text{rv1,src }re') \in \text{subs prb}\) 
    \(re' \in \text{edges }\text{(red prb)}\) 
    rb-sp-one subs-sub-rel-of
    by fast 
  qed
  next 
  case 4 show ?case using se-step(3) F by auto 
  qed 
  qed 
  qed 
  qed

next 

assume rv = tgt re

show thesis 
  proof (case-tac out-edges (black prb) (fst (tgt re)) = {}) 
  assume out-edges (black prb) (fst (tgt re)) = {}
  show thesis 
    unfolding RedBlack-subpaths-from-def Un-iff image-def Bex-def mem-Collect-eq
apply \(\text{intro disjI1}\)
apply \(\text{rule-tac } ?x=[\ ]\ \text{in exI}\)
proof \(\text{intro conjI, rule-tac } ?x=\text{tgt re in exI, intro conjI}\)
show subpath \(\text{red prb'}\ \text{rv} [\ ] \ (\text{tgt re}) \ (\text{subs prb'})\)
  using se-step(3) \(\langle \text{rv} = \text{tgt re} \rangle \ \text{rb-Nil-sp[OF RB'] by (auto simp add : vertices-def)}\)
next
have sat \(\langle \text{confs prb'} \ (\text{tgt re}) \rangle\)
  using \(\langle \text{bes} \in \text{feasible-subpaths-from} \ (\text{black prb'}) \ (\text{confs prb'} \ \text{rv}) \ (\text{fst} \ \text{rv})\rangle\)
  \(\langle \text{rv} = \text{tgt re} \rangle \ \text{se-star-sat-imp-sat}\)
  by (auto simp add : feasible-def)
thus \(\neg \ \text{marked prb'} \ (\text{tgt re})\)
  using se-step(3) \(\text{sat-not-marked[OF RB', of tgt re]}\)
  by (auto simp add : vertices-def)
next
show \(\text{bes} = \text{ui-es} [\ ]\)
  using se-step(3) \(\langle \text{rv} = \text{tgt re} \rangle \ \text{out-edges} \ (\text{black prb'}) \ (\text{fst} \ (\text{tgt re})) = \{\}\)
  \(\langle \text{bes} \in \text{feasible-subpaths-from} \ (\text{black prb'}) \ (\text{confs prb'} \ \text{rv}) \ (\text{fst} \ \text{rv})\rangle\)
  by (cases \text{bes}) (auto simp add : Graph.sp-Cons)
qued
next
assume out-edges \(\ (\text{black prb}) \ (\text{fst} \ (\text{tgt re})) \neq \{\}\)
show \text{thesis}
unfolding RedBlack-subpaths-from-def Un-iff mem-Collect-eq
apply \(\text{intro disjI2}\)
apply \(\text{rule-tac } ?x=[\ ]\ \text{in exI}\)
apply \(\text{rule-tac } ?x=\text{bes in exI}\)
proof \(\text{intro conjI, goal-cases}\)
show \(\text{bes} = \text{ui-es} [\ ] \ @ \ \text{bes} \ \text{by simp}\)
next
case 2
show \text{case}
apply \(\text{rule-tac } ?x=\text{rv in exI}\)
proof \(\text{intro conjI}\)
  have \(\neg \ \text{marked prb} \ (\text{src re})\)
  proof –
  have sat \(\langle \text{confs prb'} \ (\text{tgt re}) \rangle\)
    using \(\langle \text{bes} \in \text{feasible-subpaths-from} \ (\text{black prb'}) \ (\text{confs prb'} \ \text{rv}) \ (\text{fst} \ \text{rv})\rangle\)
    \(\langle \text{rv} = \text{tgt re} \rangle \ \text{se-star-sat-imp-sat}\)
    by (auto simp add : feasible-def)
hence \( \text{sat} (\text{confs prb}' (\text{src re})) \)
   \text{using } \text{se-step se-sat-imp-sat by auto}

moreover
\text{have } \text{src re} \neq \text{tgt re} \text{ using se-step by auto}

ultimately
\text{have } \text{sat} (\text{confs prb} (\text{src re}))
   \text{using se-step(3) by (auto simp add : vertices-def)}

thus \(?thesis
   \text{using se-step sat-not-marked}[OF se-step(1), of src re] \text{ by fast}
qued

cnext

\text{thus } \text{rv} \in \text{fringe prb}'
   \text{using se-step(3) } \langle \text{rv} = \text{tgt re}\rangle \langle \text{out-edges (black prb) (fst (tgt re))} \neq \langle {} \rangle \rangle

\begin{align*}
& \text{seE-fringe1}[OF subs-sub-rel-of[OF se-step(1)] se-step(3)] \\
& \text{seE-fringe4}[OF subs-sub-rel-of[OF se-step(1)] se-step(3)]
\end{align*}
   \text{by auto}
cnext

show \text{subpath (red prb) rv} \langle \text{rv} \rangle \langle \text{rv} (\text{subs prb'}) \rangle
   \text{using se-step(3) } \langle \text{rv} = \text{tgt re}\rangle \text{ subs-sub-rel-of}[OF RB']
   \text{by (auto simp add : subpath-def vertices-def)}
cnext

\text{show } \neg (\exists \text{res2}1 \text{ bes2}2. \text{bes} = \text{ui-es res2}1 \odot \text{bes2}2
   \land \text{res2}1 \neq []
   \land \text{subpath-from (red prb') rv res2}1 (\text{subs prb'})
\text{proof (intro notI, elim exE conjE)}
   \text{fix res}1 \text{ bes2}2 \text{ rv}'

\text{assume bes} = \text{ui-es res}1 \odot \text{bes2}2
\text{and } \text{res}1 \neq []
\text{and } \text{subpath (red prb') rv res}1 \text{ rv}' (\text{subs prb'})

\text{have } \text{out-edges (red prb') (tgt re) } \neq \langle {} \rangle \lor \text{tgt re} \in \text{subsumees (subs prb')}
\text{proof -}
   \text{obtain re' res2 where res}1 = \text{re}'\#\text{res2}
      \text{using } \langle \text{res}1 \neq [] \rangle \text{ unfolding neq-Nil-conv by blast}
hence\( rv = src\ re' \lor (rv, src\ re') \in subs\ prb \)

using\( se\text{-}step(3)\ (subpath\ (red\ prb')\ rv\ res1\ rv'\ (subs\ prb'))\)

\[ rb\text{-}sp\text{-}Cons(OF\ RB',\ of\ rv\ re'\ res2\ rv') \]

by\( auto \)

thus\ ?thesis

proof\ (elim\ disjE)
assumption\( rv = src\ re' \)

moreover
hence\( re' \in out\text{-}edges\ (red\ prb')\ (tgt\ re) \)

using\( (subpath\ (red\ prb')\ rv\ res1\ rv'\ (subs\ prb'))\)

\[ (res1 = re'\#res2)\ (rv = tgt\ re) \]

by\( (auto\ simp\ add: sp\text{-}Cons) \)

ultimately

show\ ?thesis\ using\ se\text{-}step(3)\ by\ auto

next
assumption\( (rv, src\ re') \in subs\ prb \)

hence\( tgt\ re \in red\text{-}vertices\ prb \)

using\( se\text{-}step(3)\ (rv = tgt\ re)\ subsumees\subsumees\OF\ se\text{-}step(1))\)

unfolding\ sub-rel-of-def\ by\ force

thus\ ?thesis\ using\ se\text{-}step(3)\ by\ auto

qed

qed

thus\ False

proof\ (elim\ disjE)
assumption\( out\text{-}edges\ (red\ prb')\ (tgt\ re) \neq \{\} \)

thus\ ?thesis\ using\ se\text{-}step(3)\)

by\( (auto\ simp\ add: vertices\text{-}def\ image\text{-}def) \)

next
assumption\( tgt\ re \in subsumees\ (subs\ prb') \)

hence\( tgt\ re \in red\text{-}vertices\ prb \)

using\( se\text{-}step(3)\ subsumees\subsumees\OF\ se\text{-}step(1))\)

unfolding\ subsumees\text{-}conv\ sub-rel-of-def\ by\ fastforce

thus\ ?thesis\ using\ se\text{-}step(3)\ by\ (auto\ simp\ add: vertices\text{-}def) 

qed

qed

next
show \( \text{Graph.subpath-from (black prb')} (\text{fst rv}) \text{ bes} \)
using se-step(3)
\( \langle \text{bes} \in \text{feasible-subpaths-from (black prb')} (\text{confs prb'} \text{ rv}) (\text{fst rv}) \rangle \)
by simp
qed
qed
qed
qed
next

\text{case (mark-step prb rv2 prb'} \text{ rv1})
have finite-RedBlack prb using mark-step by (auto simp add : finite-RedBlack-def)
show ?case
unfolding subset-iff
proof (intro allI impI)

fix \text{bes}
assume \( \text{bes} \in \text{feasible-subpaths-from (black prb')} (\text{confs prb'} \text{ rv1}) (\text{fst rv1}) \)
then obtain \( c \) where se-star (confs prb rv1) (trace bes (labelling (black prb')))
\( c \)
and \( \text{sat c} \)
using mark-step(3) \( \langle \text{bes} \in \text{feasible-subpaths-from (black prb')} (\text{confs prb'} \text{ rv1}) (\text{fst rv1}) \rangle \)
by (simp add : feasible-def) blast

have \( \text{bes} \in \text{RedBlack-subpaths-from prb rv1} \)
using mark-step(2) [af rv1] mark-step(3−7)
\( \langle \text{bes} \in \text{feasible-subpaths-from (black prb')} (\text{confs prb'} \text{ rv1}) (\text{fst rv1}) \rangle \)
by auto

thus \( \text{bes} \in \text{RedBlack-subpaths-from prb'} \text{ rv1} \)
apply (subst (asm) RedBlack-subpaths-from-def)
unfolding Un-iff image-def Bex-def mem-Collect-eq
proof (elim disjE exE conjE)

fix \text{res rv3}
assume \( \text{bes} = \text{ui-es res} \)
and subpath (red prb) rv1 res rv3 (subs prb)
and \( \text{marked prb rv3} \)
show ?thesis
unfolding RedBlack-subpaths-from-def Un-iff image-def Bex-def mem-Collect-eq
proof (intro disjI1,rule-tac ?x=\text{res} in exI,intro conjI)
show \( \exists rv'. \text{subpath} (\text{red prb}') rv1 \text{ res } rv' (\text{subs prb}') \land \neg \text{marked prb'} rv' \)
apply \((\text{rule-tac } \exists x = rv3 \text{ in } exl)\)
proof \((\text{intro conj1})\)
show \(\text{subpath} (\text{red prb}') rv1 \text{ res } rv3 (\text{subs prb}')\)
using mark-step(3) \(<\text{subpath} (\text{red prb}) rv1 \text{ res } rv3 (\text{subs prb})>\)
by auto

next

show \(\neg \text{marked prb'} rv3\)
proof –

have \(\text{sat} (\text{confs prb rv3})\)
proof –

have \(c \subseteq \text{confs prb rv3}\)
using mark-step(1) \(<\text{subpath} (\text{red prb}) rv1 \text{ res } rv3 (\text{subs prb})>\)
<bes = ui-es res> \(<\text{se-star} (\text{confs prb rv1}) (\text{trace bes} (\text{labelling} (\text{black prb})))>\)

<finite-RedBlack prb>
finite-RedBlack.SE-rel
by simp

thus \(?\text{thesis}\
using <\text{se-star} (\text{confs prb rv1}) (\text{trace bes} (\text{labelling} (\text{black prb})))>
<sat c>
sat-sub-by-sat
by fast

qed
thus \(?\text{thesis}\
using mark-step(3) <\text{subpath} (\text{red prb}) rv1 \text{ res } rv3 (\text{subs prb})>
lst-of-sp-is-vert[of \text{red prb rv1 res rv3 subs prb}]
sat-not-marked[OF RedBlack.mark-step[OF mark-step(1,3)]]
by auto

qed

next

show \(\text{bes} = \text{ui-es res}\) by \((\text{rule } \langle\text{bes} = \text{ui-es res}\rangle)\)

qed

next

142
fix res1 bes2 rv3 bl

assume A : bes = ui-es res1 @ bes2
and B : rv3 ∈ fringe prb
and C : subpath (red prb) rv1 res1 rv3 (subs prb)
and E : ¬ (∃ res21 bes22. bes2 = ui-es res21 @ bes22
∧ res21 ≠ []
∧ subpath-from (red prb) rv3 res21 (subs prb))
and F : Graph.subpath (black prb) (fst rv3) bes2 bl

show ?thesis
unfolding RedBlack-subpaths-from-def Un-iff mem-Collect-eq
apply (intro disjI2)
apply (rule-tac ?x=res1 in exI)
apply (rule-tac ?x=bes2 in exI)
proof (intro conjI, goal-cases)
  show bes = ui-es res1 @ bes2 by (rule ‹bes = ui-es res1 @ bes2›)
next
case 2 show ?case
  apply (rule-tac ?x=rv3 in exI)
  proof (intro conjI)
    have sat (confs prb rv3)
    proof
      obtain c' where se-star (confs prb rv1) (trace (ui-es res1) (labelling (black prb)))
      c'
      and se-star c' (trace bes2 (labelling (black prb))) c
      and sat c'
        using A ‹se-star (confs prb rv1) (trace bes (labelling (black prb)))›
      c) (sat c)
        by (simp add : se-star-append se-star-sat-imp-sat) blast
    moreover
    hence c' ⊑ confs prb rv3
      using ‹finite-RedBlack prb› mark-step(1) C finite-RedBlack.SE-rel
    by fast
    ultimately
    show ?thesis by (simp add : sat-sub-by-sat)
  qed
thus rv3 ∈ fringe prb' using mark-step(3) B by (auto simp add : fringe-def)
next

  show subpath (red prb') rv1 res1 rv3 (subs prb')
  using mark-step(3) subpath (red prb) rv1 res1 rv3 (subs prb)
  by auto

next

  show \neg (\exists res21 bes22. bes2 = ui-es res21 @ bes22
  \land res21 \neq []
  \land subpath-from (red prb') rv3 res21 (subs prb'))
  proof (intro notI, elim exE conjE)

  fix res21 bes22 rv4

  assume bes2 = ui-es res21 @ bes22
  and res21 \neq []
  and subpath (red prb') rv3 res21 rv4 (subs prb')

  show False
  using E
  proof (elim notE, rule-tac ?x=res21 in exI,
  rule-tac ?x=bes22 in exI, intro conjI)
  show bes2 = ui-es res21 @ bes22 by (rule \langle bes2 = ui-es res21
  @ bes22,\rangle)

  next
  show res21 \neq [] by (rule \langle res21 \neq []\rangle)

  next
  show subpath-from (red prb) rv3 res21 (subs prb)
  using mark-step(3)
  \langle subpath (red prb') rv3 res21 rv4 (subs prb'),
  by (simp del : split-paired-Ex) blast
  qed

  qed

  next
  show Graph.subpath-from (black prb') (fst rv3) bes2 using mark-step(3)
  F by simp blast
  qed

next
  case (subsum-step prb sub prb' rv)
hence finite-RedBlack prb by (auto simp add : finite-RedBlack-def)

have RB' : RedBlack prb' by (rule RedBlack.subsum-step[OF subsum-step(1,3)])

show ?case
unfolding subset-iff
proof (intro allI impI)

fix bes
assume bes ∈ feasible-subpaths-from (black prb') (confs prb' rv) (fst rv)

hence bes ∈ RedBlack-subpaths-from prb rv
  using subsum-step(2)[of rv] subsum-step(3−7) by auto

thus bes ∈ RedBlack-subpaths-from prb' rv
apply (subst (asm) RedBlack-subpaths-from-def)
unfolding Un-iff image-def Bex-def mem-Collect-eq
proof (elim disjE exE conjE)

fix res rv'
assume bes = ui-es res
and subpath (red prb) rv res rv' (subs prb)
and ¬ marked prb rv'
thus bes ∈ RedBlack-subpaths-from prb' rv
  using subsum-step(3) sp-in-extends[of sub red prb]
  by (simp (no-asm) only : RedBlack-subpaths-from-def Un-iff image-def
       Bex-def mem-Collect-eq,
       intro disjI1, rule-tac ?x= res in exI, intro conjI)
      (rule-tac ?x= rv' in exI, auto)

next

fix res1 bes2 rv' bl
assume A : bes = ui-es res1 @ bes2
and B : rv' ∈ fringe prb
and C : subpath (red prb) rv res1 rv' (subs prb)

and E : ¬ (∃ res21 bes22. bes2 = ui-es res21 @ bes22
          ∧ res21 ≠ []
          ∧ subpath-from (red prb) rv' res21 (subs prb))
and F : Graph. subpath (black prb) (fst rv') bes2 bl
show bes ∈ RedBlack-subpaths-from prb' rv
proof (case-tac rv' = subsumee sub)
assume \( rv' = \text{subsume} \ sub \)

show \(?thesis\)
using \( \langle \text{bes} \in \text{feasible-subpaths-from} (\text{black prb}') (\text{confs prb}' rv) (\text{fst rv}) \rangle \)
\( A \ C F \)

proof (induct \( \text{bes2} \) arbitrary : \( \text{bes bl} \) rule : \text{rev-induct}, \text{goal-cases})

\begin{quote}
\text{case (1 bes bl) thus ?case}
\end{quote}

\begin{quote}
using \text{subsum-step(3)} \ B \text{ sp-in-extends[af sub red prb]}
\end{quote}

\begin{quote}
by (simp (no-asm) only : \text{RedBlack-subpaths-from-def Un-iff image-def Bex-def mem-Collect-eq,}
\end{quote}

\begin{quote}
\text{intro disjI1, rule-tac } ?x=\text{res1} \text{ in exI, intro conjI)
\end{quote}

\begin{quote}
(rule-tac \( ?x=rv' \) \text{ in exI, auto simp add : fringe-def})
\end{quote}

\begin{quote}
next
\end{quote}

\begin{quote}
\text{case (2 be bes2 bes bl)}
\end{quote}

\begin{quote}
then obtain \( c1 \ c2 \ c3 \)
\end{quote}

\begin{quote}
where \text{se-star (confs prb}' rv) (\text{trace } (\text{ui-es res1}) (\text{labelling (black prb)}))
\end{quote}

\begin{quote}
c1
\end{quote}

\begin{quote}
and \text{se-star c1 (trace bes2 (labelling (black prb))) c2}
\end{quote}

\begin{quote}
and \text{sat c3}
\end{quote}

\begin{quote}
using \text{subsum-step(3)}
\end{quote}

\begin{quote}
by (simp add : \text{feasible-def se-star-append se-star-append-one se-star-one})
\end{quote}

\begin{quote}
blast
\end{quote}

\begin{quote}
\text{have ui-es res1 @ bes2 } \in \text{RedBlack-subpaths-from prb}' rv
\end{quote}

\begin{quote}
proof –
\end{quote}

\begin{quote}
\text{have ui-es res1 @ bes2 } \in \text{feasible-subpaths-from (black prb']}' (\text{confs prb}' rv) (\text{fst rv})
\end{quote}

\begin{quote}
proof –
\end{quote}

\begin{quote}
\text{have Graph.subpath-from (black prb)' (fst rv) (ui-es res1 @ bes2)}
\end{quote}

\begin{quote}
using \text{subsum-step 2(5) red-sp-imp-black-sp[OF subsum-step(1) C]}
\end{quote}

\begin{quote}
by (simp add : Graph.sp-append) blast
\end{quote}

\begin{quote}
moreover
\text{have feasible (confs prb}' rv)
\end{quote}

\begin{quote}
(trace (ui-es res1 @ bes2) (labelling (black prb')))\end{quote}

\begin{quote}
proof –
\end{quote}

\begin{quote}
\text{have se-star (confs prb}' rv)
\end{quote}

\begin{quote}
(trace (ui-es res1@bes2) (labelling (black prb')))\end{quote}

\begin{quote}
c2\end{quote}

146
using \texttt{subsum-step}
\begin{align*}
\langle  \texttt{se-star} & (\texttt{confs prb}' \texttt{ rv}) (\texttt{trace} (\texttt{ui-es res1}) \\
& (\texttt{labelling} (\texttt{black prb}))) (c1) \\
\langle  \texttt{se-star} c1 (\texttt{trace} \texttt{bes2} (\texttt{labelling} (\texttt{black prb}))) c2 \rangle \\
\texttt{by} & (\texttt{simp add} : \texttt{se-star-append}) \texttt{ blast} \\
\end{align*}

moreover
have \texttt{sat} c2
\begin{align*}
\langle  \texttt{se} c2 (\texttt{labelling} (\texttt{black prb}) \texttt{ be}) c3 \rangle \langle  \texttt{sat} c3 \rangle \\
\texttt{by} & (\texttt{simp add} : \texttt{se-sat-imp-sat})
\end{align*}

ultimately
show \texttt{?thesis} by \texttt{(simp add : feasible-def)} \texttt{ blast}

qed

ultimately
show \texttt{?thesis} by \texttt{simp}

qed

moreover
have \texttt{Graph.subpath-from} (\texttt{black prb}) (\texttt{fst rv}') \texttt{bes2}
\begin{align*}
\texttt{using} & \texttt{2(5)} \texttt{ by} (\texttt{auto simp add} : \texttt{ Graph.sp-append-one})
\end{align*}

ultimately
show \texttt{?thesis} using \texttt{2(1,4)} by(\texttt{auto simp add} : \texttt{Graph.sp-append-one})

qed

thus \texttt{?case}
\begin{align*}
\texttt{apply} & (\texttt{subst (asm) RedBlack-subpaths-from-def}) \\
\texttt{unfolding} & \texttt{ Un-iff image-def Bex-def mem-Collect-eq} \\
\texttt{proof} & (\texttt{elim disjE exE conjE, goal-cases})
\end{align*}

\begin{align*}
\texttt{case} & (1 \texttt{ res rv''}) \\
\texttt{show} & \texttt{?thesis} \\
\texttt{proof} & (\texttt{case-tac be} \in \texttt{ui-edge }' \texttt{ out-edges} (\texttt{red prb'}) \texttt{ rv''})
\end{align*}

\begin{align*}
\texttt{assume} & \texttt{be} \in \texttt{ui-edge }' \texttt{ out-edges} (\texttt{red prb'}) \texttt{ rv''} \\
\texttt{then obtain} & \texttt{re where} \texttt{be} = \texttt{ui-edge re} \\
\texttt{and} & \texttt{re} \in \texttt{out-edges} (\texttt{red prb'}) \texttt{ rv''} \\
\texttt{by} & \texttt{blast}
\end{align*}

\begin{align*}
\texttt{show} & \texttt{?thesis} \\
\texttt{unfolding} & \texttt{RedBlack-subpaths-from-def Un-iff image-def Bex-def mem-Collect-eq} \\
\texttt{apply} & (\texttt{intro disjI1})
\end{align*}
apply (rule-tac \(?x=res \circ [re]\) in exI)
proof (intro conjI, rule-tac \(?x=tgt\ re\) in exI, intro conjI)
  show subpath \((\text{red prb'})\ rv (\text{res} \circ [re])\) \((\text{tgt}\ re)\) \((\text{subs} \text{prb'})\)
    using 1(2) \(<re \in \text{out-edges} \ (\text{red prb'})\ rv'>\)
    by (simp add : sp-append-one)

next
show \(\neg\ \text{marked prb'}\ (\text{tgt}\ re)\)
proof –
  have sat \((\text{confs prb'}\ rv)\)
    proof –
      have subpath \((\text{red prb'})\ rv (\text{res} \circ [re])\) \((\text{tgt}\ re)\) \((\text{subs} \text{prb'})\)
        using 1(2) \(<re \in \text{out-edges} \ (\text{red prb'})\ rv''\>
        by (simp add : sp-append-one)
      then obtain \(c\)
        where se-star \((\text{confs prb'}\ rv)\)
          \((\text{trace} \ (\text{ui-es} \ (\text{res} \circ [re])) \ (\text{labelling} \ (\text{black prb}))))\)
          \(c\)
        using subsum-step(3,5,6,7) \(RB'\)
          finite-RedBlack.sp-imp-ex-se-star-succ
            [of prb' rv res@[re] tgt re]
        unfolding finite-RedBlack-def
        by simp blast
    hence sat \(c\)
      using 1(1)
      \(<\text{se-star} \ (\text{confs prb'}\ rv)\ (\text{trace} \ (\text{ui-es} \ (\text{res} \circ [re])) \ (\text{labelling} \ (\text{black prb}))\) \(c1)\>
        \(<\text{se-star} \ c1 \ (\text{trace} \ (\text{bes}2 \ (\text{labelling} \ (\text{black prb})))\) c2)\>
        \(<\text{se} \ c2 \ (\text{labelling} \ (\text{black prb}) \ be) \ c3)\>
        \(<\text{sat} \ c3) \ (\text{be} = \text{ui-edge} \ re)\>
        
    se-star-succ-states
      [of \text{confs prb'} rv
        trace(\text{ui-es}(res@[re]))(\text{labelling}(black prb))
        c3]
      apply (subst (asm) eq-commute)
      by (auto simp add : se-star-append-one se-star-append
        se-star-one sat-eq)
    moreover
    have \(c \sqsubseteq \text{confs prb'}\ (\text{tgt}\ re)\)
      using subsum-step(3,5,6,7)
      \(<\text{subpath} \ (\text{red prb'})\ rv (\text{res} \circ [re])\) \((\text{tgt}\ re)\) \((\text{subs} \text{prb'})\),
        \(<\text{se-star} \ (\text{confs prb'}\ rv)(\text{trace} \ (\text{ui-es} \ (\text{res} \circ [re]))\))\>

148
(labelling (black prb))(c)\)

\(\text{finite-RedBlack.SE-rel[of prb']} RB'\)

by \(\text{simp add : finite-RedBlack-def}\)

ultimately
show \(\text{thesis by simp add: sat-by-sat}\)
qed

thus \(\text{thesis}\)

using \(\langle \text{re } \in \text{ out-edges (red prb')} \text{ rv''}\rangle,\)

sat-not-marked[OF RB', of tgt re]

by \(\text{auto simp add : vertices-def}\)

qed

next

show \(\text{bes = ui-es (res@[re]) using 1(1) 2(3) be = ui-edge re}\) by simp

qed

next

assume \(\text{be } \notin \text{ ui-edge ' out-edges (red prb')} \text{ rv''}\)

show \(\text{thesis}\)

proof \(\text{(case-tac rv'' } \in \text{ subsumees (subs prb')}\)

assume \(\text{rv'' } \in \text{ subsumees (subs prb')}\)

then obtain \(\text{arv'' where (rv'',arv'') } \in \text{ (subs prb')}\) by auto

hence subpath \(\text{(red prb')} \text{ rv res arv'' (subs prb')}\)

using \(\langle \text{subpath (red prb')} \text{ rv res rv'' (subs prb')}\rangle\)

by \(\text{simp add : sp-append-sub}\)

show \(\text{thesis}\)

proof \(\text{(case-tac be } \in \text{ ui-edge ' out-edges (red prb')} \text{ arv''}\)

assume \(\text{be } \in \text{ ui-edge ' out-edges (red prb')} \text{ arv''}\)

then obtain \(\text{re where re } \in \text{ out-edges (red prb')} \text{ arv''}\)

and \(\text{be = ui-edge re}\)

by blast

show \(\text{thesis}\)

unfolding RedBlack-subpaths-from-def Un-iff image-def

Bex-def mem-Collect-eq

149
apply (intro disjI)
apply (rule-tac ?x=res@[re] in exI)
proof (intro conjI, rule-tac ?x=tgt re in exI, intro conjI)

show subpath (red prb') rv (res@[re]) (tgt re) (subs prb')
  using (subpath (red prb') rv res arv'' (subs prb'))
  (re ∈ out-edges (red prb') arv''),
  by (simp add : sp-append-one)

next

have sat (confs prb' (tgt re))
proof –
  have subpath (red prb') rv (res@[re]) (tgt re) (subs prb')
    using (subpath (red prb') rv res arv'' (subs prb'))
    (re ∈ out-edges (red prb') arv''),
    by (simp add : sp-append-one)
then obtain c
where se: se-star (confs prb' rv) (trace (ui-es (res@[re])))
  (labelling (black prb))) (c)
  using subsum-step(3,5,6,7) RB'
finite-RedBlack, sp-imp-ex-se-star-succ
  [of prb' rv res@[re] tgt re]
unfolding finite-RedBlack-def
by simp blast

hence sat c
  using I(1)
  ⟨se-star (confs prb' rv) (trace (ui-es res I))
    (labelling (black prb))) (c I)⟩,
  ⟨se-star c1 (trace bes2 (labelling (black prb))) c2⟩,
  ⟨se c2 (labelling (black prb) be) c3⟩, ⟨sat c3⟩,
  ⟨be = ui-edge re⟩
  se-star-succ-states
  [of confs prb' rv
    trace (ui-es(res@[re]))
    (labelling (black prb))]
  c3⟩
apply (subst (asm) eq-commute)
by (auto simp add : se-star-append-one se-star-append
se-star-one sat-eq)

moreover
have c ⊑ confs prb' (tgt re)

150
\textbf{using} \texttt{subsum-step}(3, 5, 6, 7) \texttt{se RB'}
\texttt{finite-RedBlack.SE-rel[af prb']}
\texttt{subpath (red prb') rv (res@[re]) (tgt re) (subs prb')}
\texttt{by (simp add : finite-RedBlack-def)}

ultimately
\texttt{show ?thesis by (simp add: sat-sub-by-sat)}
\texttt{qed}

thus \texttt{¬ marked prb' (tgt re)}
\texttt{using \langle re ∈ out-edges (red prb') arv'';}
\texttt{sat-not-marked[OF RB', of tgt re]}
\texttt{by (auto simp add : vertices-def)}

next
\texttt{show bes = ui-es (res @ [re])}
\texttt{using \langle bes = ui-es res1 @ bes2 @ [be]\rangle}
\texttt{\langle ui-es res1 @ bes2 = ui-es res\rangle}
\texttt{\langle be = ui-edge re\rangle}
\texttt{by simp}

\texttt{qed}

next

\texttt{assume A : be `∈ ui-edge ` out-edges (red prb') arv''}

\texttt{have src be = fst arv''}
\texttt{proof –}
\texttt{have Graph.subpath (black prb') (fst rv) (ui-es res1 @ bes2)}
\texttt{\langle ui-es res1 @ bes2 = ui-es res\rangle}
\texttt{\langle subpath (red prb') rv res arv'' (subs prb')\rangle}
\texttt{red-sp-imp-black-sp[OF RB']}
\texttt{by auto}

moreover
\texttt{have Graph.subpath (black prb') (fst rv) (ui-es res1 @ bes2)}
\texttt{\langle bes ∈ feasible-subpaths-from (black prb') (confs prb' rv)\rangle}
\texttt{\langle bes = ui-es res1 @ bes2 @ [be]\rangle}
\texttt{by (auto simp add : Graph.sp-append Graph.sp-append-one}
ultimately
show ?thesis
using sp-same-src-imp-same-tgt by fast
qed

show ?thesis
unfolding RedBlack-subpaths-from-def Un-iff mem-Collect-eq
apply (intro disjI2)
apply (rule-tac ?x=res in exI)
apply (rule-tac ?x=[be] in exI)
proof (intro conjI, goal-cases)

show bes = ui-es res @ [be]
using (bes = ui-es res1 @ bes2 @ [be])
(by simp)

next

case 2 show ?case
apply (rule-tac ?x=arv'' in exI)
proof (intro conjI)

show arv'' ∈ fringe prb'
unfolding fringe-def mem-Collect-eq
proof (intro conjI)
show arv'' ∈ red-vertices prb'
using (subpath (red prb') rv res arv'' (subs prb'))
by (simp add : lst-of-sp-is-vert)
next
show arv'' ∈ subsumees (subs prb')
using ((rv'',arv'') ∈ subs prb'' subs-wf-sub-rel[OF RB'])
unfolding wf-sub-rel-def Ball-def
by (force simp del : split-paired-All)
next
show ¬ marked prb' arv''
using ((rv'',arv'') ∈ (subs prb'') subsumer-not-marked[OF RB'])
by fastforce
next
have be ∈ edges (black prb')
using subsum-step(3)
(by Graph.subpath (black prb) (fst rv') (bes2 @ [be]) bl)
by (simp add : Graph.sp-append-one)

\[ \text{thus } \text{ui-edge ' out-edges (red prb')} \text{ arv'' } \subset \text{ out-edges (black } (\text{fst arv''}) \]

using \(<\text{src be } = \text{fst arv''}> \text{ A red-\text{OA-subset-black-\text{OA}[OF}}

RB', of arv'']

by auto

c qed

next

show subpath (red prb') rv res arv'' (subs prb')

by (rule \(<\text{subpath (red prb')} \text{ rv res arv'' (subs prb')}>)

next

show \neg (\exists \text{res21 bes22. [be] } = \text{ui-es res21 } @ \text{bes22}

& res21 \neq []

& subpath-from (red prb') arv'' res21 (subs prb'))

proof (intro notI, elim exE conjE, goal-cases)

case (1 res21 bes22 rv''')

have be \in \text{ui-edge ' out-edges (red prb')} \text{ arv''}

proof –

obtain re res21' where res21 = re # res21'

using 1(2) unfolding neq-Nil-conv by blast

have be = \text{ui-edge re and } re \in \text{out-edges (red prb')} \text{ arv''}

proof –

show be = \text{ui-edge re using 1(1) \(<\text{res21 } = \text{re } # \text{res21'}>}

by simp

next

have re \in \text{edges (red prb')}

using 1(3) \(<\text{res21 } = \text{re } # \text{res21'}> by (simp add :}

sp-Cons)

moreover

have src re = arv''

proof –

have \(<\text{arv''}\text{,src re} > \notin \text{subs prb'}

using \(<\text{rv''}\text{,arv''} > \in \text{subs prb'}\text{ subs-wf-sub-rel[OF}

RB']

unfolding wf-sub-rel-def Ball-def

153
by (force simp del : split-paired-All)

thus ?thesis
  using 1(3) \langle res21 = re \# res21' \rangle
  by (simp add : rb-sp-Cons[OF RB'])
qed

ultimately
show \( re \in \text{out-edges (red prb')} \) arv'' by simp
qed

thus ?thesis by auto
qed

thus False using A by (elim notE)
qed

next

show Graph.subpath-from (black prb') (fst arv'') [be]
  using subsum-step(3)
    \langle Graph.subpath (black prb) (fst rv') (bes2 @ [be]) bl \rangle
    \langle (rv'',arv'') \in \text{subs prb'} \rangle
    \langle subpath (red prb') rv res arv'' (subs prb') \rangle
    \langle src be = fst arv'' \rangle
    RB'' \text{red-sp-imp-black-sp subs-to-same-BL}
by (simp add : Graph.sp-append-one Graph.sp-one)
qed
qed

next

assume rv'' \notin \text{subsumees (subs prb')}

show ?thesis
proof (case-tac be \in ui-edge \{ out-edges (red prb') rv'' \})
  assume be \in ui-edge \{ out-edges (red prb') rv'' \}

then obtain re where be = ui-edge re
  and \( re \in \text{out-edges (red prb') rv''} \)
  by blast

show ?thesis
unfolding RedBlack-subpaths-from-def Un-iff image-def

Bex-def mem-Collect-eq

apply (intro disjI)
apply (rule-tac ?x=res @ re in exI)
apply (intro conjI)

proof (rule-tac ?x=tgt re in exI, intro conjI)
show subpath (red prb) rv (res @ re) (tgt re) (subs prb)'
  using (subpath (red prb) rv res rv'' (subs prb))
  \( \mbox{by (simp add : sp-append-one)} \)

next
show \( \neg \) marked prb' (tgt re)
proof –
  have sat (confs prb' (tgt re))
  proof –
  have subpath (red prb) rv (res@re) (tgt re) (subs prb)'
    using (subpath (red prb) rv res rv'' (subs prb'))
    \( \mbox{by (simp add : sp-append-one)} \)
  then obtain c
  where se : se-star (confs prb' rv) (trace (ui-es (res@re))
    (labelling (black prb))))(c)
    using subsum-step(3,5,6,7) RB'
    finite-RedBlack,sp-imp-ex-se-star-succ
    \[ \mbox{of prb' rv res@re tgt re} \]
    unfolding finite-RedBlack-def
    by simp blast

hence sat c
  using 1(1)
  \( \langle \mbox{se-star (confs prb' rv) (trace (ui-es res1) \}
    (labelling (black prb)))) \( \mbox{c1} \) \)
  \( \langle \mbox{se-star c1 (trace bes2 (labelling (black prb))) } \mbox{c2} \) \)
  \( \langle \mbox{se c2 (labelling (black prb) be) } \mbox{c3 } \langle \mbox{sat c3} \) \)
  \( \langle \mbox{be = ui-edge re} \) \)
  se-star-succ-states
  \[ \mbox{of confs prb' rv} \)
  trace (ui-es (res@re)) (labelling (black prb))
    \( \mbox{c3} \)
  apply (subst (asm) eq-commute)
  by (auto simp add : se-star-append-one se-star-append
    se-star-one sat-eq)
moreover
have $c \subseteq \text{cons prb'} (\text{tgt re})$
  using subsum-step(3,5,6,7) se RB'
  finite-RedBlack.SE-rel[of prb']
  subpath (red prb') ru (res@[re]) (tgt re) (subs
prb'),

by (simp add : finite-RedBlack-def)

ultimately
show ?thesis by (simp add: sat-by-sat)
qed

thus ?thesis
  using $\langle re \in \text{out-edges (red prb')} ru'' \rangle$
  sat-not-marked[OF RB', of tgt re]
  by (auto simp add : vertices-def)
qed

next
show bes = ui-es (res @ [re])
  using $\langle bes = ui-es res1 @ bes2 @ [be] \rangle$
  $\langle ui-es res1 @ bes2 = ui-es res \rangle$
  $\langle be = ui-edge re \rangle$

  by simp
qed

next
assume A : be /$\notin$ ui-edge ' out-edges (red prb') ru''

show ?thesis
  unfolding RedBlack-subpaths-from-def Un-iff Bex-def
  mem-Collect-eq
  apply (intro disjI2)
  apply (rule-tac $?x=\text{res}$ in exI)
  apply (rule-tac $?x=[\text{be}]$ in exI)
  proof (intro conj1, goal-cases)
    show bes = ui-es res @ [be]
      using $\langle ui-es res1 @ bes2 = ui-es res \rangle$
      $\langle bes = ui-es res1 @ bes2 @ [be] \rangle$
      by simp
  next
  case 2
    have src be = fst ru''
    proof –
have \( \text{Graph.subpath}(\text{black } \text{prb}') (\text{fst } \text{rv}) (\text{ui-es } \text{res}) (\text{src } \text{be}) \)

using \( \langle \text{bes} \in \text{feasible-subpaths-from}(\text{black } \text{prb}') \rangle (\text{cons } \text{prb}' \text{ rv}) (\text{fst } \text{rv}) \)
\( \langle \text{bes} = \text{ui-es } \text{res}1 \otimes \text{bes}2 \otimes [\text{be}] \rangle \)
\( \langle \text{ui-es } \text{res}1 \otimes \text{bes}2 = \text{ui-es } \text{res} \rangle \)
red-sp-imp-black-sp
\[ \langle \text{OF } \text{RB}' \langle \text{subpath}(\text{red } \text{prb}') \text{ rv res rv''}(\text{subs } \text{prb}') \rangle \rangle \]
by (subst (asm)(2) eq-commute)
(auto simp add: Graph.sp-append Graph.sp-one)

thus \( ?\text{thesis} \)
using red-sp-imp-black-sp
\[ \langle \text{OF } \text{RB}' \langle \text{subpath}(\text{red } \text{prb}') \text{ rv res rv''}(\text{subs } \text{prb}') \rangle \rangle \]
by (rule sp-same-src-imp-same-tgt)
qed

show \( ?\text{case} \)
apply (rule-tac \( ?x=\text{rv''} \) in \( \text{exI} \))
proof (intro conjI)

show \( \text{rv''} \in \text{fringe } \text{prb}' \)

unfolding fringe-def mem-Collect-eq
proof (intro conjI)

show \( \text{rv''} \in \text{red-vertices } \text{prb}' \)
using \( \langle \text{subpath}(\text{red } \text{prb}') \text{ rv res rv''}(\text{subs } \text{prb}') \rangle \)
by (simp add : lst-of-sp-is-vert)
next
show \( \text{rv''} \notin \text{subsumees}(\text{subs } \text{prb}') \)
by (rule \( \langle \text{rv''} \notin \text{subsumees}(\text{subs } \text{prb}') \rangle \))
next
show \( \neg \text{marked } \text{prb}' \text{ rv''} \) by (rule \( \neg \text{marked } \text{prb}' \text{ rv''} \))
next
have \( \text{be} \in \text{edges}(\text{black } \text{prb}') \)
using subsum-step(3)
\( \langle \text{Graph.subpath}(\text{black } \text{prb}) (\text{fst } \text{rv}') (\text{bes2 } \otimes [\text{be}]) \text{ bl} \rangle \)
by (simp add : Graph.sp-append-one)

thus \( \text{ui-edge } \cdot \text{out-edges}(\text{red } \text{prb}') \text{ rv''} \subset \text{out-edges}(\text{black } \text{prb}') (\text{fst } \text{rv}'') \)
using \( \langle \text{src be} = \text{fst } \text{rv''}, \text{A} \rangle \text{ red-OA-subset-black-OA}[\text{OF } \text{RB}', \text{ of } \text{rv''}] \)
by auto

157
qed

next

show subpath (red prb') rv res rv'' (subs prb')
  by (rule (subpath (red prb') rv res rv'' (subs prb')))

next

show \( \neg(\exists res21 \ bes22. \ [be] = u_i-es res21 @ bes22 \wedge res21 \neq [] \wedge SubRel.subpath-from (red prb') (rv'') (res21) (subs prb')) \)
proof (intro notI, elim exE conjE, goal-cases)
  case (1 res21 bes22 rv''')
  have be \( \in \) ui-edge ' out-edges (red prb') rv''
  proof
    obtain re res21' where res21 = re # res21'
      using 1(2) unfolding neq-Nil-conv by blast
    have be = ui-edge re
    and re \( \in \) out-edges (red prb') rv''
    proof
      show be = ui-edge re using 1(1) \( \langle \) res21 = re#res21' \( \rangle \)
      by simp
    next
      have re \( \in \) edges (red prb')
      using 1(3) \( \langle \) res21 = re # res21' \( \rangle \) by (simp add : sp-Cons)
    moreover
      have src re = rv''
      proof
        have \( \langle rv'', src re \rangle \notin subs prb' \)
          using \( \langle rv'' \notin subsumees (subs prb') \rangle \) by force
        thus \( \langle \) thesis \( \rangle \)
        using 1(3) \( \langle \) res21 = re # res21' \( \rangle \)
        by (simp add : rb-sp-Cons[OF RB'])
    qed
  next
    have re \( \in \) out-edges (red prb') rv'' by simp
  qed

158
thus \(?thesis\) by auto
qed

thus \(False\) using \(A\) by \((elim\ notE)\)
qed

next

\textbf{show} \(\text{Graph.subpath-from}\ (\text{black prb'}) (fst rv') [be]\)
\textbf{using} \(\text{subsum-step(3)}\)
\hspace{1cm} \langle \text{Graph.subpath (black prb') (fst rv') (bes2 @ [be])} \rangle
\hspace{1cm} \langle \text{src be = fst rv'} \rangle
\textbf{by} \((\text{rule-tac ?x=tgt be \textbf{in} exI})\)
\hspace{1cm} (\text{simp add : Graph.sp-append-one Graph.sp-one})

qed
\hspace{0.5cm} qed
\hspace{0.5cm} qed
\hspace{0.5cm} qed
\hspace{0.5cm} qed

next

\textbf{case} \((2\ \text{res1'}\ \text{bes2'}\ \text{rv''}\ \text{bl'})\)

\textbf{show} \(?thesis\)
\textbf{proof} \((\text{case-tac bes2'} = []\))

assume \(\text{bes2'} = []\)

\textbf{have} \(\text{Graph.subpath (black prb') (fst rv) (ui-es res1' @ [be]) bl}\)
\textbf{proof} \(\phantom{\langle \text{src be = fst rv'} \rangle}\)
\hspace{1cm} \textbf{have} \(\text{Graph.subpath (black prb') (fst rv) (ui-es res1') (src be)}\)
\textbf{proof} \(\phantom{\langle \text{src be = fst rv'} \rangle}\)
\hspace{1cm} \textbf{have} \(\text{Graph.subpath (black prb') (fst rv') bes2 (src be)}\)
\hspace{1cm} using \(\text{subsum-step(3)}\)
\hspace{1cm} \langle \text{Graph.subpath (black prb') (fst rv') (bes2@[be]) bl} \rangle
\hspace{1cm} by \((\text{simp add : Graph.sp-append-one})\)

moreover
\textbf{have} \(\text{subpath (red prb') rv res1 rv' (subs prb')}\)
\hspace{1cm} using \(\text{subsum-step(3)} \langle \text{subpath (red prb') rv res1 rv' (subs prb')} \rangle\)

159
by (auto simp add : sp-in-extends)

hence Graph.subpath (black prb') (fst rv) (ui-es res1) (fst rv') using RB' by (simp add : red-sp-imp-black-sp)

ultimately
show ?thesis
  using (ui-es res1 @ bes2 = ui-es res1' @ bes2' \(\text{\{bes2' = []\}}\)) by (subst (asm) eq-commute) (auto simp add : Graph.sp-append)
qed

moreover
have Graph.subpath (black prb') (src be) [be] bl using subsum-step(3) \(\langle\text{Graph.subpath (black prb) (fst rv)}\rangle\)
  by (simp add : Graph.sp-append-one Graph.sp-one)

ultimately
show ?thesis by (auto simp add : Graph.sp-append)
qed

hence Graph.subpath (black prb') (fst rv) (ui-es res1') (src be)
and be \(\in\) edges (black prb')
and tgt be = bl
  by (simp-all add : Graph.sp-append-one)

have fst rv'' = src be

proof
  have Graph.subpath (black prb') (fst rv) (ui-es res1') (fst rv'')
    using (subpath (red prb') rv res1' rv'' (subs prb'),
      red-sp-imp-black-sp[OF RB']
    by fast

  thus ?thesis
    using (Graph.subpath (black prb') (fst rv) (ui-es res1') (src be))
    by (simp add : sp-same-src-imp-same-tgt)
qed

show ?thesis

proof (case-tac be \(\in\) ui-edge ' out-edges (red prb') rv'')

assume be \(\in\) ui-edge ' out-edges (red prb') rv''

then obtain re where be = ui-edge re
and \( re \in \text{out-edges} (\text{red prb}') rv'' \)
by blast

show \(?thesis\)
unfolding RedBlack-subpaths-from-def Un-iff
image-def Bex-def mem-Collect-eq
apply (intro disjI1)
apply (rule-tac \(?x=\text{res1}'@[re] \in exI\))
apply (intro conjI)
apply (rule-tac \(?x=\text{tgt} re \in exI\))
proof (intro conjI)
show subpath (\text{red prb}') rv (\text{res1}'@[re]) (\text{tgt} re) (\text{subs} \text{prb}')
using \(\text{subpath} (\text{red prb}') rv \text{res1}' rv'' (\text{sub} \text{prb}')\)
\( re \in \text{out-edges} (\text{red prb}') rv'' \)
by (simp add : sp-append-one)

next
show \(\neg \text{marked} \text{prb}' (\text{tgt} re)\)
proof
have sat (confs \text{prb}' (\text{tgt} re))
proof
have subpath (\text{red prb}') rv (\text{res1}'@[re]) (\text{tgt} re) (\text{subs} \text{prb}')
using \(\text{subpath} (\text{red prb}') rv \text{res1}' rv'' (\text{sub} \text{prb}')\)
\( re \in \text{out-edges} (\text{red prb}') rv'' \)
by (simp add : sp-append-one)

then obtain \( c\)
where se : se-star (confs \text{prb}' rv) (trace (ui-es (\text{res1}'@[re])))
(using (labelling (black \text{prb}))) (c)
using subsum-step(3,5,6,7) RB'
finite-RedBlack.sp-imp-ex-se-star-succ
[of \text{prb}' rv \text{res1}'@[re] \text{tgt} re]
unfolding finite-RedBlack-def
by simp blast

hence sat c
proof
have bes = ui-es (\text{res1}'@[re])
using \(\text{bes} = \text{ui-es} \text{res1} @ \text{bes2} @[\text{be}]\)
\(\text{be} = \text{ui-edge} \text{re} (\text{bes2}' = []\)
\(\text{ui-es} \text{res1} @ \text{bes2} = \text{ui-es} \text{res1}' @ \text{bes2}'\)
by simp
thus \( ?\text{thesis} \)
\[
\begin{align*}
\text{using } & \text{subsum-step(3) } \text{se-star-succs-states[OF } \text{se}] \\text{ } \\
& \langle \text{bes } \in \text{feasible-subpaths-from (black } \text{prb}') \text{ (confs prb') rv} \rangle \\
& \langle \text{fst rv} \rangle \\
\text{by (auto simp add : feasible-def sat-eq) }
\end{align*}
\]
\qedsymbol 

moreover have \( c \sqsubseteq \text{confs prb'} (\text{tgt re}) \)
\[
\begin{align*}
\text{using } & \text{subsum-step(3,5,6,7) se } \\
& \text{finite-RedBlack.SE-rel[of prb'] RB'} \\
& \langle \text{subpath (red prb') (rv) (res1'@[re])} \rangle \\
& \langle \text{tgt re} \rangle \text{ (subs prb')} \rangle \\
\text{by (simp add : finite-RedBlack-def) }
\end{align*}
\]
ultimately show \( ?\text{thesis} \) by (simp add: sat-sub-by-sat)
\qedsymbol 

thus \( ?\text{thesis} \)
\[
\begin{align*}
\text{using } & \text{re } \in \text{out-edges (red prb') rv''}, \\
& \text{sat-not-marked[OF } \text{RB'}, of tgt re] } \\
\text{by (auto simp add : vertices-def) }
\end{align*}
\]
\qedsymbol 

next show \( \text{bes } = \text{ui-es (res1'@[re])} \)
\[
\begin{align*}
\text{using } & \langle \text{bes } = \text{ui-es res1 @ bes2 @ [be]} \rangle \\
& \langle \text{ui-es res1 @ bes2 } = \text{ui-es res1' @ bes2'} \rangle \\
& \langle \text{bes2'} = [], \text{be = ui-edge re} \rangle \\
\text{by simp}
\end{align*}
\]
\qedsymbol 

next

assume \( A : \text{be } \notin \text{ui-edge ' out-edges (red prb') rv''} \)
show \( ?\text{thesis} \)

unfolding \( \text{RedBlack-subpaths-from-def Un-iff mem-Collect-eq} \)
apply (intro disjI2)
apply (rule-tac \( ?x=\text{res1'} \) in \( \text{exI} \))
apply (rule-tac \( ?x=[\text{be}] \) in \( \text{exI} \))
proof (intro conjI, goal-cases)
show \( \text{bes} = \text{ui}-\text{es res1}' \@ \text{[be]} \)
\begin{align*}
\text{using} & \langle \text{bes} = \text{ui}-\text{es res1} \@ \text{bes2} \@ \text{[be]} \rangle, \\
& \langle \text{ui}-\text{es res1} \@ \text{bes2} = \text{ui}-\text{es res1}' @ \text{bes2}' \rangle, \\
& \langle \text{bes2}' = [] \rangle,
\end{align*}
by \( \text{simp} \)

next

\text{case 2 show } \text{?case} \\
\text{apply } (\text{rule-tac } ?x=rv''\text{ in exI}) \\
\text{proof } (\text{intro conjI})

\begin{align*}
\text{show } rv'' \in \text{fringe prb'} \text{ by } & (\text{rule } rv'' \in \text{fringe prb'}) \\
\text{next}
\end{align*}

\begin{align*}
\text{show } & \text{subpath (red prb') rv res1'} rv'' (\text{subs prb'}) \\
& \text{by } (\text{rule } \langle \text{subpath (red prb') rv res1'} rv'' (\text{subs prb'}) \rangle)
\end{align*}

next

\begin{align*}
\text{show } & \neg (\exists \text{res21 bes22. [be] = ui}-\text{es res21} \@ \text{bes22} \\
& \quad \land \text{res21} \neq [] \\
& \quad \land \text{subpath-from (red prb') (rv'')} \\
& \quad \langle \text{res21} \rangle (\text{subs prb'}) \rangle) \\
\text{proof } (\text{intro notI}, \text{clim exE conjE, goal-cases}) \\
\text{case } & (1 \text{ res21 bes22 rv''})
\end{align*}

\begin{align*}
\text{then obtain } & \text{re res21'} \text{ where } \text{be} = \text{ui-edge re} \\
& \quad \text{and } \text{res21} = \text{re} \# \text{res21'}
\end{align*}

\text{unfolding } \text{neq-Nil-conv by auto}

moreover

\begin{align*}
\text{hence } & \text{re} \in \text{out-edges (red prb') rv''} \\
& \text{using } 1(3) \langle rv'' \in \text{fringe prb'} \rangle \text{ RB'} \\
& \text{unfolding } \text{subsumees-conv by } (\text{force simp add : fringe-def}) \\
& \text{rb-sp-Cons})
\end{align*}

ultimately

\text{show } False \text{ using A by auto}

qed
next

show Graph.subpath-from (black prb') (fst rv'') [be]
  using \langle Graph.subpath (black prb') (fst rv) (ui-es res1'@[be]) \rangle bl

  \langle fst rv'' = src be \rangle
  by (auto simp add : Graph.sp-append-one Graph.sp-one)

qed

qed

next

assume bes2' ≠ []

then obtain be' bes2'' where bes2' = be' ≠ bes2''

unfolding neq-Nil-conv by blast

show ?thesis

unfolding RedBlack-subpaths-from-def Un-iff mem-Collect-eq
apply (intro disjI2)
apply (rule-tac \( x=res1' \) in exI)
apply (rule-tac \( x=bes2'@[be] \) in exI)
proof (intro conjI, goal-cases)

  show bes = ui-es res1' @ bes2' @ [be]
  using \langle bes = ui-es res1 @ bes2 @ [be] \rangle
  \langle ui-es res1 @ bes2 = ui-es res1' @ bes2' \rangle
  by simp

next

case 2 show ?case

  apply (rule-tac \( x=rv'' \) in exI)
  proof (intro conjI)

    show rv'' ∈ fringe prb' by (rule \( rv'' ∈ fringe prb' \))

next

show subpath (red prb') rv res1' rv'' (subs prb')
  by (rule \langle subpath (red prb') rv res1' rv'' (subs prb') \rangle)
next

\[ \begin{align*}
\text{\textbf{show}} & \quad \neg (\exists \text{res21 \ bes22}. \ \text{bes2'} \ \text{[be]} = \text{ui-es res21} \ \text{[@bes22]} \\
& \quad \land \text{res21} \neq [] \\
& \quad \land \text{subpath-from (red prb') (rv'')} \\
& \quad (\text{res21}) (\text{subs prb'})
\end{align*} \]

\textbf{proof} (\textit{intro notI, elim exE conjE, goal-cases})
\textbf{case} (1 \res21 \bes22 \ \rv''')
then obtain \re \res21' where \res21 = \re \# \res21'
and \be' = \text{ui-edge} \re
using \langle \text{bes2'} = \be' \# \bes2'' \rangle unfolding \text{neq-Nil-conv}
by auto

\textbf{show} False
\textbf{using} \langle \neg (\exists \text{res21 \ bes22}. \ \text{bes2'} = \text{ui-es res21} \ \text{[@bes22]} \\
& \quad \land \text{res21} \neq [] \\
& \quad \land \text{subpath-from (red prb') (rv'')} \\
& \quad (\text{res21}) (\text{subs prb'})\rangle \}
apply (\textit{elim notE})
apply (\textit{rule-tac ?x=[re in exI})
apply (\textit{rule-tac ?x=bes2'' in exI})
proof (\textit{intro conjI})
\textbf{show} \text{bes2'} = \text{ui-es [re] @ bes2''}
\textbf{using} \langle \text{bes2'} \ \text{[be]} = \text{ui-es res21} \ \text{[@bes22]}, \\
& \langle \text{bes2'} = \be' \# \bes2''\rangle \\
& \langle \be' = \text{ui-edge} \ \re\rangle
\text{by simp}
next
\textbf{show} \re \neq [] \text{by simp}
next
\textbf{show} \text{subpath-from (red prb') rv'' [re] (subs prb')} \\
\textbf{using} \langle \text{subpath (red prb') rv'' res21 rv'''}(\text{subs prb'})\rangle
\langle \text{res21} = \re \# \res21'\rangle
\text{by (fastforce simp add : sp-Cons Nil-sp vertices-def)}
qed

qed

next

\textbf{show} Graph.\text{subpath-from (black prb') (fst rv'')} \langle \text{bes2'} @

165
proof
have Graph.subpath (black prb') (fst rv)
  (ui-es res1' @ bes2') (src be)

proof
have Graph.subpath (black prb') (fst rv)
  (ui-es res1 @ bes2) (src be)
  using \bes \in \text{feasible-subpaths-from} (black prb')
  (confs prb' rv)
  (fst rv)
  \bes = ui-es res1 @ bes2 \in [be]
by (auto simp add : Graph.sp-append Graph.sp-one)

thus \?thesis using \ui-es res1 @ bes2 = ui-es
  by simp
qed

moreover
have Graph.subpath (black prb')(fst rv)(ui-es res1' @
  bes2') bl'
  using \Graph.subpath (black prb') (fst rv'') bes2' bl'
  red-sp-imp-black-sp(OF RB'
  \subpath (red prb')(rv)(res1')
  (rv'') (subs prb'))
by (auto simp add : Graph.sp-append)

ultimately
have src be = bl' by (rule sp-same-src-imp-same-tgt)

moreover
have Graph.subpath (black prb') (src be) [be] (tgt be)
  using subsum-step(3)
  \Graph.subpath (black prb) (fst rv') (bes2@[be])
  by (auto simp add : Graph.sp-append-one Graph.sp-one)

ultimately
show \?thesis
  using \Graph.subpath (black prb') (fst rv'') bes2' bl'
  by (simp add : Graph.sp-append-one Graph.sp-one)
qed
qed
qed
qed

next

assume \( rv' \neq \text{subsumee sub} \)

show \( ?\text{thesis} \)
  unfolding \( \text{RedBlack-subpaths-from-def Un-iff mem-Collect-eq} \)
  apply (intro disjI2)
  apply (rule-tac \( ?x=\text{res1 in exI} \))
  apply (rule-tac \( ?x=\text{bes2 in exI} \))
  proof (intro conjI, goal-cases)
    show \( \text{bes} = \text{ui-es res1 @ bes2} \) by (rule \( \langle \text{bes} = \text{ui-es res1 @ bes2} \rangle \))
  next
    case 2 show \( ?\text{case} \)
      apply (rule-tac \( ?x=rv' \) in exI)
      proof (intro conjI)
        show \( rv' \in \text{fringe prb'} \)
        using subsum-step(3) \( \text{subsumE-fringe[OF subsum-step(3)]} \) \( B \langle rv' \neq \text{subsumee sub} \rangle \)
        by simp
      next
        show \( \text{subpath (red prb')} rv \text{ res1 rv' (subs prb')} \)
        using subsum-step(3) \( C \) by (auto simp add: sp-in-extends)
    next
    show \( \neg (\exists \text{res21 bes22. bes2} = \text{ui-es res21 @ bes22} \)
      \wedge \text{res21} \neq [] \)
      \wedge \text{subpath-from (red prb')} \( rv' \text{ res21 (subs prb')} \))
      proof (intro notI, elim exE conjE)
        fix \( \text{res21 bes22 rv''} \)
        assume \( \text{bes2 = ui-es res21 @ bes22} \)
        and \( \text{res21} \neq [] \)
        and \( \text{subpath (red prb')} \( rv' \text{ res21 rv'' (subs prb')} \))
        then obtain \( re \text{ res21'} \) where \( \text{res21} = re \# \text{res21'} \)
        unfolding neq-Nil-conv by blast
        have \( \text{subpath (red prb')} \( rv' [re] (tgt re) \) (subs prb) \)
        proof –
          have \( \neg \text{uses-sub rv'} [re] (tgt re) \) sub using \( \langle rv' \neq \text{subsumee sub} \rangle \)
by auto

thus ?thesis
using subsum-step(3)
    ⟨subpath (red prb') rv' res21 rv'' (subs prb')⟩ ⟨res21 = re ≠ res21'⟩

wf-sub-rel-of.sp-in-extends-not-using-sub
[OF subs-wf-sub-rel-of[OF subsum-step(1)],
of subsumee sub subsumer sub subs prb' rv' [re] tgt re]
rb-sp-Cons[OF RB', of rv' re res21' rv'']
rb-sp-one[OF subsum-step(1), of rv' re tgt re]
subs-sub-rel-of[OF subsum-step(1)]
by auto
qed

show False
using E
apply (elim notE)
apply (rule-tac ?z=[re] in exI)
apply (rule-tac ?z=ui-es res21'@bes22 in exI)
proof (intro conjI)
    show bes2 = ui-es [re] @ ui-es res21' @ bes22
using ⟨bes2 = ui-es res21 @ bes22⟩ ⟨res21 = re ≠ res21'⟩ by simp
next
    show [re] ≠ [] by simp
next
    show subpath-from (red prb) rv' [re] (subs prb)
using subsum-step(3)
    ⟨rv' ≠ subsumee sub⟩ ⟨subpath (red prb') rv' res21 rv'' (subs prb')⟩
⟨res21 = re ≠ res21'⟩,
rb-sp-Cons[OF RB', of rv' re res21' rv'']
rb-sp-one[OF subsum-step(1), of rv' re tgt re]
subs-sub-rel-of[OF subsum-step(1)] subs-sub-rel-of[OF RB']
by fastforce
qed
qed
next
show Graph.subpath-from (black prb') (fst rv') bes2
using subsum-step(3) F by simp blast
qed
qed

168
qed
qed

next

\textbf{case} (abstract-step prb rv2 c a prb' rv1)
\textbf{have} RB': RedBlack prb' \textbf{by} (rule RedBlack.abstract-step[OF abstract-step(1,3)])
\textbf{have} finite-RedBlack prb \textbf{using} abstract-step \textbf{by} (auto simp add : finite-RedBlack-def)
\textbf{show} ?case
\textbf{unfolding} subset-iff
\textbf{proof} (intro allI impI)

\textbf{fix} bes

\textbf{assume} bes \(\in\) feasible-subpaths-from (black prb') (confs prb' rv1) (fst rv1)

\textbf{show} bes \(\in\) RedBlack-subpaths-from prb' rv1
\textbf{proof} (case-tac rv2 = rv1)

\textbf{assume} rv2 = rv1

\textbf{show} ?thesis
\textbf{proof} (case-tac out-edges (black prb') (fst rv1) = {})
\textbf{assume} out-edges (black prb') (fst rv1) = {}
\textbf{show} ?thesis
\textbf{unfolding} RedBlack-subpaths-from-def Un-iff image-def Bex-def mem-Collect-eq
\textbf{apply} (intro disjI1)
\textbf{apply} (rule-tac ?x=[] in exI)
\textbf{apply} (intro conjI)
\textbf{apply} (rule-tac ?x=rv1 in exI)
\textbf{proof} (intro conjI)
\textbf{show} subpath (red prb') rv1 [] rv1 (subs prb')
\textbf{using} abstract-step(4) rb-Nil-sp[OF RB'] \textbf{by} fast
\textbf{next}
\textbf{show} ¬ marked prb' rv1 using abstract-step(3) (rv2 = rv1)
\textbf{by} simp

\textbf{next}
\textbf{show} bes = ui-es []
\textbf{using} \(\vdash\) bes \(\in\) feasible-subpaths-from (black prb') (confs prb' rv1)
\(\langle\text{fst rv1}\rangle\)

\textbf{by} simp

\textbf{qed}
assume \( \text{out-edges (black prb)} (\text{fst rv1}) \neq \{\} \)

show \(?thesis\)
unfolding RedBlack-subpaths-from-def Un-iff mem-Collect-eq
apply (intro disjI2)
apply (rule-tac \(?x=[]\) in exI)
apply (rule-tac \(?x=\text{bes}\) in exI)
proof (intro conjI, goal-cases)

show \(\text{bes} = \text{ui-es }[] \at \text{bes}\) by simp

next

\begin{itemize}
\item \text{case 2 show ?case}
\item apply (rule-tac \(?x=\text{rv1}\) in exI)
\item proof (intro conjI)
\end{itemize}

show \(\text{rv1} \in \text{fringe prb}'\)
using abstract-step(1,3) \(\text{rv2} = \text{rv1}\) \(\text{out-edges (black prb)} (\text{fst rv1}) \neq \{\}\)
by (auto simp add : fringe-def)

next

show \(\text{subpath (red prb')} \text{rv1} [] \text{rv1} (\text{subs prb'})\)
using abstract-step(3) \(\text{rv2} = \text{rv1}\)
\(\text{rb-Nil-sp[OF RedBlack.abstract-step[OF abstract-step(1,3)]]}\)
by auto

next

show \(\neg (\exists \text{res21 bes22. bes} = \text{ui-es res21} @ \text{bes22} \wedge \text{res21} \neq [] \wedge \text{subpath-from (red prb')} \text{rv1 res21 (subs prb')})\)
proof (intro notI, elim exE conjE)
fix \text{res21 rv3}

assume \(\text{res21} \neq []\)
and \(\text{subpath (red prb')} \text{rv1 res21 rv3 (subs prb')}\)

moreover
then obtain \(\text{re res21}'\) where \(\text{res21} = \text{re} \# \text{res21}'\)
unfolding neq-Nil-conv by blast

ultimately
have re ∈ out-edges (red prb′) rv1
using abstract-step(3) ‹rv2 = rv1›
rb-sp-Cons[OF RedBlack.abstract-step[OF abstract-step(1,3)],
of rv1 re res21 rv3]
unfolding subsumees-conv by fastforce

thus False using abstract-step(3) ‹rv2 = rv1› by auto
qed

next

show Graph.subpath-from (black prb) (fst rv1) bes
using ‹bes ∈ feasible-subpaths-from (black prb) (confs prb rv1)›
by simp

qed

qed

next

assume rv2 ≠ rv1

moreover
hence feasible (confs prb rv1) (trace bes (labelling (black prb)))
using abstract-step(3)
⟨bes ∈ feasible-subpaths-from (black prb) (confs prb’ rv1) (fst rv1)›
by simp

ultimately
have bes ∈ RedBlack-subpaths-from prb rv1
using abstract-step(2)[of rv1] abstract-step(3–7)
⟨bes ∈ feasible-subpaths-from (black prb) (confs prb’ rv1) (fst rv1)›
by auto

thus ?thesis
apply (subst (asm) RedBlack-subpaths-from-def)
unfolding Un-iff image-def Bex-def mem-Collect-eq
proof (elim disjE exE conjE)

fix res rv3
assume \( \text{bes} = \text{ui-es res} \)
and \( \text{subpath (red prb) rv1 res rv3 (subs prb)} \)
and \( \neg \text{marked prb rv3} \)

thus \( \neg \text{thesis} \)
using abstract-step(3)
unfolding RedBlack-subpaths-from-def Un-iff image-def Bex-def mem-Collect-eq
by (intro disjI1, rule-tac \( ?x = \text{res} \) in \( \text{exI} \), intro conjI)
(rule-tac \( ?x = \text{rv3} \) in \( \text{exI} \), simp-all)

next

fix \( \text{res1 bes2 rv3 bl} \)
assume \( A : \text{bes} = \text{ui-es res1} \ @ \text{bes2} \)
and \( B : \text{rv3} \in \text{fringe prb} \)
and \( C : \text{subpath (red prb) rv1 res1 rv3 (subs prb)} \)
and \( E : \neg (\exists \text{res21 bes22} . \text{bes2} = \text{ui-es res21} \ @ \text{bes22}) \)
and \( \neg \text{subpath-from (red prb) rv3 res21 (subs prb)} \)
and \( F : \text{Graph.subpath (black prb) (fst rv3) bes2 bl} \)

show \( \neg \text{thesis} \)
unfolding RedBlack-subpaths-from-def Un-iff mem-Collect-eq
apply (intro disjI2)
apply (rule-tac \( ?x = \text{res1} \) in \( \text{exI} \))
apply (rule-tac \( ?x = \text{bes2} \) in \( \text{exI} \))
proof (intro conjI, goal-cases)
show \( \text{bes} = \text{ui-es res1} \ @ \text{bes2} \) by (rule \( \text{bes} = \text{ui-es res1} \ @ \text{bes2} \))
next
case 2 show \( \neg \text{case} \)
using abstract-step(3) \( \text{B C E F unfolding fringe-def} \)
by (rule-tac \( ?x = \text{rv3} \) in \( \text{exI} \)) auto
qed

qed

next

case (strengthen-step prb rv2 e prb' rv1)
show \( \neg \text{case} \)
unfolding subset-iff
proof (intro allI impI)
fix bes
assume bes ∈ feasible-subpaths-from (black prb') (confs prb' rv1) (fst rv1)

hence bes ∈ RedBlack-subpaths-from prb rv1

using strengthen-step(2) [of rv1] strengthen-step(3–7) by auto

thus bes ∈ RedBlack-subpaths-from prb' rv1
apply (subst (asm) RedBlack-subpaths-from-def)
unfolding Un-iff image-def Bex-def mem-Collect-eq
proof (elim disjE exE conjE)

next

fix res rv2
assume bes = ui-es res
and subpath (red prb) rv1 res rv2 (subs prb)
and ¬ marked prb rv2
thus ?thesis

using strengthen-step(3)
unfolding RedBlack-subpaths-from-def Un-iff image-def Bex-def mem-Collect-eq
by (intro disjI1) fastforce

next

fix res1 bes2 rv3 bl
assume A : bes = ui-es res1 @ bes2
and B : rv3 ∈ fringe prb
and C : subpath (red prb) rv1 res1 rv3 (subs prb)
and E : ¬ (∃ res21 bes22. bes2 = ui-es res21 @ bes22
∧ res21 ≠ []
∧ subpath-from (red prb) rv3 res21 (subs prb))
and F : Graph.subpath (black prb) (fst rv3) bes2 bl

show ?thesis

unfolding RedBlack-subpaths-from-def Un-iff mem-Collect-eq
apply (intro disjI2)
apply (rule-tac ?x = res1 in exI)
apply (rule-tac ?x = bes2 in exI)
proof (intro conjI, goal-cases)
show bes = ui-es res1 @ bes2 by (rule (bes = ui-es res1 @ bes2))
next
case 2
show ?case
using strengthen-step(3) B C E F unfolding fringe-def
by (rule-tac ?x = rv3 in exI) auto
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

using feasible-subpaths-preserved [OF assms, of root (red prb)] consistent-roots [OF assms]

by (simp add : vertices-def)

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 ⊆ confs prb (root (red prb))

using assms

proof (induct prb)

case base thus ?case by (simp add: subsums-refl)

next

case se-step thus ?case by (force simp add : vertices-def)

next

case mark-step thus ?case by simp

next

case subsum-step thus ?case by simp

next

case (abstract-step prb rv c a prb')

thus ?case by (auto simp add : abstract-def subsums-trans)

next

case strengthen-step thus ?case by simp

qed

**Theorem** (in finite-RedBlack) *feasible-path-inclusion-from-init* :

174
assumes RedBlack prb
shows feasible-paths (black prb) (init-conf prb) ⊆ RedBlack-paths prb
unfolding subset-iff mem-Collect-eq
proof (intro allI impI, elim exE conjE, goal-cases)
case (1 es bl)

hence es ∈ feasible-subpaths-from (black prb) (init-conf prb) (fst (root (red prb)))
    using consistent-roots[OF assms] by simp blast

hence es ∈ feasible-subpaths-from (black prb) (confs prb (root (red prb))) (fst(root(red prb)))
    unfolding mem-Collect-eq
    proof (elim exE conjE, goal-cases)
      case (1 bl')
      show ?case
      proof (rule-tac ?x = bl' in exI, intro conjI)
        show Graph.subpath (black prb) (fst (root (red prb))) es bl' by (rule 1(1))
      next
        have finite-labels (trace es (labelling (black prb)))
            using finite-RedBlack by auto

      moreover
        have finite (pred (confs prb (root (red prb))))
            using finite-RedBlack finite-pred[OF assms]
            by (auto simp add : vertices-def finite-RedBlack-def)

      moreover
        have finite (pred (init-conf prb))
            using assms by (intro finite-init-pred)

      moreover
        have ∀ e∈pred (confs prb (root (red prb))), finite (Bexp.vars e)
            using finite-RedBlack finite-pred-constr-symvars[OF assms]
            by (fastforce simp add : finite-RedBlack-def vertices-def)

      moreover
        have ∀ e∈pred (init-conf prb), finite (Bexp.vars e)
            using assms by (intro finite-init-pred-symvars)

      moreover
        have init-conf prb ⊆ confs prb (root (red prb))
            using assms by (rule init-subsumed)
ultimately
show feasible (confs prb (root (red prb))) (trace es (labelling (black prb)))

using 1(2) by (rule subsums-imp-feasible)
qed
qed

thus ?case
using feasible-subpaths-preserved[OF assms, of root (red prb)]
by (auto simp add : vertices-def)
qed

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 subsum-
tions. It would be possible to generate the core operations on red-black
graphs by the Isabelle code-generator, by introducing un-interpreted func-
tion 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

Symbolic Execution Tool for Verification,” in Proceedings of CAV ’12,

Model Checker Blast,” STTT, vol. 9, no. 5-6, pp. 505–525, 2007.

SAT-Based Predicate Abstraction for ANSI-C,” in Proceedings of
TACAS ’05, 2005, pp. 570–574.

P. Ashar, “F-Soft: Software Verification Platform,” in Proceedings of

thesizing Software Verifiers from Proof Rules,” in Proceedings of PLDI

123–136.

symbolic execution techniques: Proof of correctness and preservation

paths via graph transformations and symbolic execution: a method

S. Peyronnet, “Coverage-biased Random Exploration of large Models

178


