

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

December 14, 2021

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-ity-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 <i>aexp</i> 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 <i>bexp</i> 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 <i>store</i> type-synonym	23
6.1.2	Symbolic variables of a store	23
6.1.3	Fresh symbolic variables	24

6.2	Consistency	24
6.3	Adaptation of an arithmetic expression to a store	25
6.4	Adaptation of a boolean expression to a store	28
7	Configurations, Subsumption and Symbolic Execution	31
7.1	Basic Definitions and Properties	31
7.1.1	Configurations	31
7.1.2	Symbolic variables of a configuration.	32
7.1.3	Freshness.	32
7.1.4	Satisfiability	32
7.1.5	States of a configuration	32
7.1.6	Subsumption	33
7.1.7	Semantics of a configuration	34
7.1.8	Abstractions	34
7.1.9	Entailment	34
7.2	Symbolic Execution	35
7.2.1	Definitions of se and se_star	35
7.2.2	Basic properties of se	36
7.2.3	Monotonicity of se	41
7.2.4	Basic properties of se_star	42
7.2.5	Monotonicity of se_star	43
7.2.6	Existence of successors	43
7.3	Feasibility of a sequence of labels	48
7.4	Concrete execution	50
8	Labelled Transition Systems	53
8.1	Basic definitions	53
8.2	Feasible sub-paths and paths	54
9	Graphs equipped with a subsumption relation	55
9.1	Basic definitions and properties	56
9.1.1	Subsumptions and subsumption relations	56
9.2	Well-formed subsumption relation of a graph	57
9.2.1	Well-formed subsumption relations	57
9.2.2	Subsumption relation of a graph	58
9.2.3	Well-formed sub-relations	59
9.3	Consistent Edge Sequences, Sub-paths	59
9.3.1	Consistency in presence of a subsumption relation	59
9.3.2	Sub-paths	62

10 Extending rooted graphs with edges	67
10.1 Definition and Basic properties	68
10.2 Extending trees	69
10.3 Properties of sub-paths in an extension	72
11 Extending subsumption relations	74
11.1 Definition	74
11.2 Properties of extensions	75
11.3 Properties of sub-paths in an extension	77
12 Red-Black Graphs	82
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	177
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`).

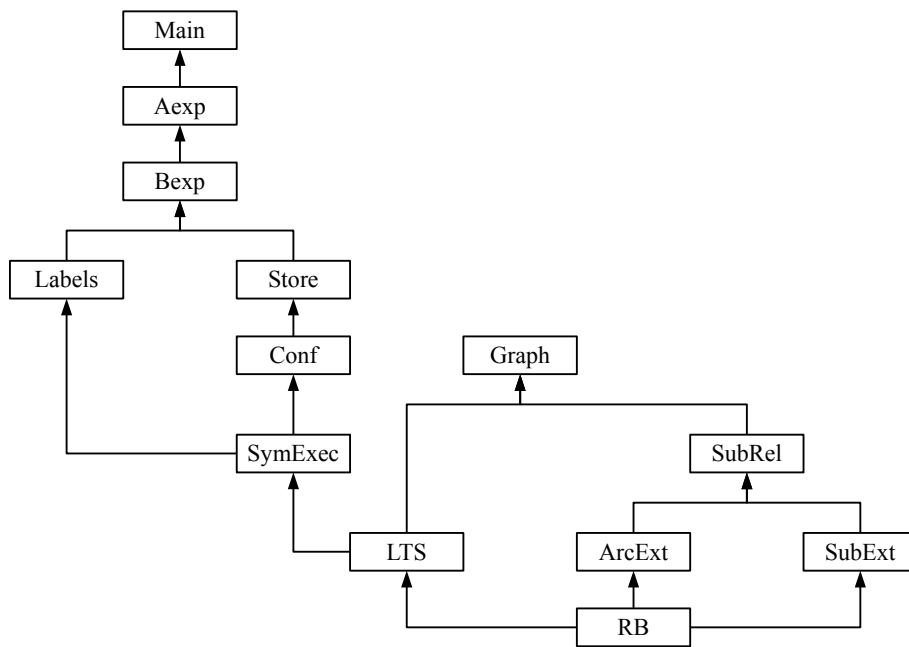


Figure 1: The hierarchy of theories.


```

theory Graph
imports Main
begin

```

2 Rooted Graphs

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

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

```

lemma neq-Nil-conv2 :
   $xs \neq [] = (\exists x xs'. xs = xs' @ [x])$ 
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 =
  src  :: 'v
  tgt  :: 'v

```

2.1.2 Rooted graphs

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

```

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

```

2.1.3 Vertices

The set of vertices of a rooted graph is made of its root and the endpoints of its edges. Isabelle/HOL provides *extensible records*, i.e. it is possible to

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

definition *vertices* ::

$(\prime v, \prime x)$ *rgraph-scheme* $\Rightarrow \prime v$ *set*

where

$vertices\ g \equiv \{root\ g\} \cup src\ \prime edges\ g \cup tgt\ \prime edges\ g$

2.1.4 Basic properties of rooted graphs

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

abbreviation *loop-free* ::

$(\prime v, \prime x)$ *rgraph-scheme* $\Rightarrow bool$

where

$loop-free\ g \equiv \forall\ e \in edges\ g. src\ e \neq tgt\ e$

abbreviation *wf-rgraph* ::

$(\prime v, \prime x)$ *rgraph-scheme* $\Rightarrow bool$

where

$wf-rgraph\ g \equiv root\ g \in src\ \prime edges\ g = (edges\ g \neq \{\})$

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

2.1.5 Out-going edges

This abbreviation will prove handy in the following.

abbreviation *out-edges* ::

$(\prime v, \prime x)$ *rgraph-scheme* $\Rightarrow \prime v \Rightarrow \prime v$ *edge set*

where

$out-edges\ g\ v \equiv \{e \in edges\ g. src\ e = v\}$

2.2 Consistent Edge Sequences, Sub-paths and Paths

2.2.1 Consistency of a sequence of edges

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

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

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

2.2.2 Sub-paths and paths

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

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

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

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

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

```
lemma fst-of-sp-is-vert :  
  assumes subpath g v1 es v2  
  shows v1 ∈ vertices g  
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 *subpath-from* whenever this final vertex has no importance, and *subpath* otherwise.

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

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

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

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

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

The empty sequence is a path of any rooted graph.

lemma
 $[] \in \text{paths } g$

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

Some useful simplification lemmas for *subpath*.

lemma *sp-one* :

$subpath\ g\ v1\ [e]\ v2 = (src\ e = v1 \wedge e \in edges\ g \wedge tgt\ e = v2)$

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

lemma *sp-Cons* :

$subpath\ g\ v1\ (e\#es)\ v2 = (src\ e = v1 \wedge e \in edges\ g \wedge subpath\ g\ (tgt\ e)\ es\ v2)$

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

lemma *sp-append-one* :

$subpath\ g\ v1\ (es@[e])\ v2 = (subpath\ g\ v1\ es\ (src\ e) \wedge e \in edges\ g \wedge tgt\ e = v2)$

by (*induct es arbitrary : v1, auto simp add : subpath-def vertices-def*)

lemma *sp-append* :

$subpath\ g\ v1\ (es1@es2)\ v2 = (\exists\ v.\ subpath\ g\ v1\ es1\ v \wedge subpath\ g\ v\ es2\ v2)$

by (*induct es1 arbitrary : v1*)

(*(simp add : subpath-def, fast,*

(auto simp add : fst-of-sp-is-vert sp-Cons))

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*)

(*auto simp add : sp-Cons subpath-def vertices-def*)

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  $e \in \text{set } es$ 
shows  $\exists es'. es = es' @ [e] \wedge e \notin \text{set } es'$ 
using assms(2,3)
proof (induction es arbitrary : v1)
  case Nil thus ?case by simp
next
  case (Cons e' es)

```

hence $e = e' \vee (e \neq e' \wedge e \in \text{set } es)$ **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  $\Rightarrow$   $'v$  edge  $\Rightarrow$  ( $'v, 'x$ ) rgraph-scheme
where
  add-edge g e  $\equiv$  rgraph.edges-update ( $\lambda$  edges. edges  $\cup$   $\{e\}$ ) g

```

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

```

lemma sp-ends-in-tgt-imp-mem :
  assumes  $\text{tgt } e \notin \text{vertices } g$ 
  assumes  $v \neq \text{tgt } e$ 
  assumes subpath (add-edge g e) v es (tgt e)
  shows  $e \in \text{set } es$ 
proof –
  have  $es \neq []$  using assms(2,3) by (auto simp add : subpath-def)

  then obtain  $e' es'$  where  $es = es' @ [e']$  by (simp add : neq-Nil-conv2) blast

```

```

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

```

The empty graph is thus a tree.

```

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

```

```

end
theory Aexp
imports Main
begin

```

3 Arithmetic Expressions

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

3.1 Variables and their domain

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

over symbolic variables. This distinction eases some parts of the following formalization.

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

type-synonym $'v \text{ symvar} = 'v \times \text{nat}$

lemma

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

by ($\text{simp add : finite-prod}$)

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 *aexp* type-synonym

Arithmetic (and boolean, see `Bexp.thy`) expressions are represented by their semantics, i.e. total functions giving values in D to states. This way of

representing expressions has the benefit that it is not necessary to define the syntax of terms (and formulae) appearing in program statements and path predicates.

type-synonym $(\text{'v}, \text{'d}) \text{ aexp} = (\text{'v}, \text{'d}) \text{ state} \Rightarrow \text{'d}$

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

3.4 Variables of an arithmetic expression

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

definition $\text{vars} ::$

$(\text{'v}, \text{'d}) \text{ aexp} \Rightarrow \text{'v set}$

where

$\text{vars } e = \{v. \exists \sigma \text{ val. } e (\sigma(v := \text{val})) \neq e \sigma\}$

lemma $\text{vars-example-1} :$

fixes $e :: (\text{'v}, \text{integer}) \text{ aexp}$

assumes $e = (\lambda \sigma. \sigma x - \sigma y)$

assumes $x \neq y$

shows $\text{vars } e = \{x, y\}$

unfolding set-eq-iff

proof (intro allI iffI)

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

then obtain $\sigma \text{ val}$

where $e (\sigma(v := \text{val})) \neq e \sigma$

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  $e::('v, \text{integer}) \text{ aexp}$ 
  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 ::
   $'v \Rightarrow ('v, 'd) \text{ aexp} \Rightarrow \text{bool}$ 
where
   $\text{fresh } v \ e \equiv v \notin \text{vars } e$ 

```

```

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.

type-synonym $(\prime v, \prime d) \text{ bexp} = (\prime v, \prime d) \text{ state} \Rightarrow \text{bool}$

definition *vars* ::

$(\prime v, \prime d) \text{ bexp} \Rightarrow \prime v \text{ set}$

where

$\text{vars } e = \{v. \exists \sigma \text{ val. } e (\sigma(v := \text{val})) \neq e \sigma\}$

abbreviation *fresh* ::

$\prime v \Rightarrow (\prime v, \prime d) \text{ bexp} \Rightarrow \text{bool}$

where

$\text{fresh } v \ e \equiv v \notin \text{vars } e$

4.1.2 Satisfiability of an expression

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

definition *sat* ::

$(\prime v, \prime d) \text{ bexp} \Rightarrow \text{bool}$

where

$\text{sat } e = (\exists \sigma. e \ \sigma)$

4.1.3 Entailment

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

definition *entails* ::

$(\prime v, \prime d) \text{ bexp} \Rightarrow (\prime v, \prime d) \text{ bexp} \Rightarrow \text{bool}$ (**infixl** \models_B 55)

where

$\varphi \models_B \psi \equiv (\forall \sigma. \varphi \ \sigma \longrightarrow \psi \ \sigma)$

4.1.4 Conjunction

In the following, path predicates are represented by sets of boolean expressions. We define the conjunction of a set of boolean expressions E as the

expression that associates *true* to a state σ if, for all elements e of E , e associates *true* to σ .

definition *conjunct* ::

$(\text{'v, 'd}) \text{ bexp set} \Rightarrow (\text{'v, 'd}) \text{ bexp}$

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.

lemma *vars-of-conj* :

$\text{vars } (\lambda \sigma. e1 \sigma \wedge e2 \sigma) \subseteq \text{vars } e1 \cup \text{vars } e2$

(**is** $\text{vars } ?e \subseteq \text{vars } e1 \cup \text{vars } e2$)

unfolding *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 *vars-def* **by** *blast*

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

thus $v \in \text{vars } e1 \cup \text{vars } e2$ **unfolding** *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 *?case* **by** (*simp add : vars-def conjunct-def*)
next
case (*2 e E*)

thus *?case*
using *vars-of-conj[of e conjunct E]*
by (*rule-tac rev-finite-subset, auto simp add : 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 vars ?e $\subseteq \text{Aexp.vars } e1 \cup \text{Aexp.vars } e2$)
unfolding *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 *vars-def* **by** *blast*

hence $e1 (\sigma (v := \text{val})) \neq e1 \sigma \vee e2 (\sigma (v := \text{val})) \neq e2 \sigma$
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 `jump`, `break`, etc) are labelled by a *Skip*.

```

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

```

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

```

definition finite-label ::
  ('v,'d) label  $\Rightarrow$  bool
where
  finite-label l  $\equiv$  case l of
    Assume e  $\Rightarrow$  finite (Bexp.vars e)
  | Assign - e  $\Rightarrow$  finite (Aexp.vars e)
  | -  $\Rightarrow$  True

```

```

abbreviation finite-labels ::
  ('v,'d) label list  $\Rightarrow$  bool
where
  finite-labels ls  $\equiv$  ( $\forall l \in$  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 $'a \text{ store} = 'a \Rightarrow \text{nat}$

6.1.2 Symbolic variables of a store

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

definition *symvar* ::

$'a \Rightarrow 'a \text{ store} \Rightarrow 'a \text{ symvar}$

where

$\text{symvar } v \ s \equiv (v, s \ v)$

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

lemma

$\text{inj } (\lambda v. \text{symvar } v \ s)$

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 \Rightarrow *'a symvar set*

where

symvars s = $(\lambda v. \text{symvar } v \text{ } s)$ ' (*UNIV::'a set*)

6.1.3 Fresh symbolic variables

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

definition *fresh-symvar* ::

'v symvar \Rightarrow *'v store* \Rightarrow *bool*

where

fresh-symvar sv s = $(sv \notin \text{symvars } s)$

6.2 Consistency

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

definition *consistent* ::

('v,'d) state \Rightarrow *('v symvar, 'd) state* \Rightarrow *'v store* \Rightarrow *bool*

where

consistent $\sigma \sigma_{sym} s \equiv (\forall v. \sigma_{sym} (\text{symvar } v \text{ } s) = \sigma v)$

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

lemma

$\exists \sigma \sigma_{sym}. \text{consistent } \sigma \sigma_{sym} s$

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 $\sigma \sigma_{sym} s = (\forall sv \in \text{symvars } s. \sigma_{sym} sv = \sigma (fst \text{ } sv))$

by (*auto simp add : consistent-def symvars-def symvar-def*)

lemma *consistent-eq2* :

consistent $\sigma \sigma_{sym} \text{ } store = (\sigma = (\lambda v. \sigma_{sym} (\text{symvar } v \text{ } store)))$

by (*auto simp add : consistent-def*)

lemma *consistentI1* :
consistent σ $(\lambda sv. \sigma (fst sv))$ *store*
using *consistent-eq1* **by** *fast*

lemma *consistentI2* :
consistent $(\lambda v. \sigma_{sym} (symvar v store))$ 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* ::
 $(v, d) aexp \Rightarrow v store \Rightarrow (v symvar, d) aexp$
where
 $adapt-aexp e s = (\lambda \sigma_{sym}. e (\lambda v. \sigma_{sym} (symvar v s)))$

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

lemma *adapt-aexp-is-subst* :
assumes *consistent* σ σ_{sym} s
shows $(adapt-aexp e s) \sigma_{sym} = e \sigma$
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 $v \in Aexp.vars\ e$
shows $symvar\ v\ s \in Aexp.vars\ (adapt\ aexp\ e\ s)$ (**is** $?sv \in Aexp.vars\ ?e'$)
proof –
obtain $\sigma\ val$ **where** $e\ (\sigma\ (v := val)) \neq e\ \sigma$
using *assms unfolding Aexp.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 *Aexp.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-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* :
 $Aexp.vars\ (adapt\ aexp\ e\ s) \subseteq symvars\ s$ (**is** $Aexp.vars\ ?e' \subseteq symvars\ s$)
unfolding *subset-iff*
proof (*intro allI impI*)
fix sv

assume $sv \in Aexp.vars\ ?e'$

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

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

hence $?e'\ (\sigma_{sym}\ (sv := val)) = ?e'\ \sigma_{sym}$
by (*simp add : adapt-aexp-def*)

```

thus False
using ⟨?e' ( $\sigma_{sym} (sv := val)$ )  $\neq$  ?e'  $\sigma_{sym}$ ⟩
by (elim notE)
qed

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 ∈ symvars s by (simp add : symvars-def)
qed

lemma symvar-var-imp-var :
  assumes sv ∈ Aexp.vars (adapt-axp e s) (is sv ∈ Aexp.vars ?e')
  shows fst sv ∈ Aexp.vars e
proof –
  obtain v where sv = (v, s v)
  using assms(1) symvars-of-adapt-axp
  unfolding symvars-def symvar-def
  by blast

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

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

  ultimately
  show ?thesis
  using ⟨sv = (v, s v)⟩
    consistentI2[of  $\sigma_{sym} s$ ]
    consistentI2[of  $\sigma_{sym} (sv := val) s$ ]
  unfolding Aexp.vars-def
  by (simp add : adapt-axp-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* :
 $Aexp.vars (adapt-aexp e s) = (\lambda v. symvar v s) \text{ ` } Aexp.vars e$
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* ::
 $(\text{'}v, \text{'}d) bexp \Rightarrow \text{'}v \text{ store} \Rightarrow (\text{'}v \text{ symvar}, \text{'}d) bexp$
where
 $adapt-bexp e s = (\lambda \sigma. e (\lambda x. \sigma (symvar x s)))$

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

lemma *var-imp-symvar-var2* :
 assumes $v \in Bexp.vars e$
 shows $symvar v s \in Bexp.vars (adapt-bexp e s)$ (**is** $?sv \in Bexp.vars ?e'$)
proof –
 obtain $\sigma \text{ val}$ **where** $A : e (\sigma (v := val)) \neq e \sigma$

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.vars (adapt-bexp e s) \subseteq symvars s (is Bexp.vars ?e' \subseteq ?SV)

proof

fix *sv*

assume *sv* \in *Bexp.vars ?e'*

then obtain $\sigma_{sym}\ val$

where $?e' (\sigma_{sym} (sv := val)) \neq ?e' \sigma_{sym}$

by *(simp add : Bexp.vars-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.vars (adapt-bexp e s)* **(is** *sv* \in *Bexp.vars ?e')*

shows *fst sv* \in *Bexp.vars e*

proof –

obtain v **where** $sv = (v, s v)$
using *assms symvars-of-adapt-bexp*
unfolding *symvars-def symvar-def*
by *blast*

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

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

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

lemma *adapt-bexp-vars* :
 $Bexp.vars (adapt-bexp e s) = (\lambda v. symvar v s) \text{ ` } Bexp.vars e$
(is 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 $fst sv \in vars e$ **by** *(rule symvar-var-imp-var2)*

moreover
have $sv = symvar (fst sv) s$
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** $v \in vars e$ $sv = symvar v s$ **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 =

```

$store :: 'v store$
 $pred :: ('v symvar, 'd) bexp set$

7.1.2 Symbolic variables of a configuration.

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

definition $symvars ::$

$('v, 'd) conf \Rightarrow 'v symvar set$

where

$symvars c = Store.symvars (store c) \cup Bexp.vars (conjunct (pred c))$

7.1.3 Freshness.

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

definition $fresh-symvar ::$

$'v symvar \Rightarrow ('v, 'd) conf \Rightarrow bool$

where

$fresh-symvar sv c = (sv \notin symvars c)$

7.1.4 Satisfiability

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

abbreviation $sat ::$

$('v, 'd) conf \Rightarrow bool$

where

$sat c \equiv Bexp.sat (conjunct (pred c))$

7.1.5 States of a configuration

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

definition $states ::$

$('v, 'd) conf \Rightarrow ('v, 'd) state set$

where

$states c = \{\sigma. \exists \sigma_{sym}. consistent \sigma \sigma_{sym} (store c) \wedge conjunct (pred c) \sigma_{sym}\}$

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

lemma *sat-eq* :
 sat c = (states c ≠ {})
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* \Rightarrow ('v,'d) *conf* \Rightarrow *bool* (**infixl** \sqsubseteq 55)
where
 $c_2 \sqsubseteq c_1 \equiv (\text{states } c_2 \subseteq \text{states } c_1)$

The subsumption relation is reflexive and transitive.

lemma *subsums-refl* :
 $c \sqsubseteq c$
by (*simp only : subsums-def*)

lemma *subsums-trans* :
 $c_1 \sqsubseteq c_2 \Longrightarrow c_2 \sqsubseteq c_3 \Longrightarrow c_1 \sqsubseteq c_3$
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 $c_1 \sqsubseteq c_2$
assumes $c_2 \sqsubseteq c_1$
shows $\text{states } c_1 = \text{states } c_2$
using *assms* **by** (*simp add : subsums-def*)

A satisfiable configuration can only be subsumed by satisfiable configurations.

lemma *sat-sub-by-sat* :
assumes *sat c2*
and $c_2 \sqsubseteq c_1$
shows *sat c1*
using *assms sat-eq[of c1] sat-eq[of c2]*
by (*simp add : subsums-def*) *fast*

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

lemma *unsat-subs-unsat* :
assumes $\neg \text{sat } c1$
assumes $c2 \sqsubseteq c1$
shows $\neg \text{sat } c2$
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 $\text{sem } c_1$ entails $\text{sem } c_2$. The following theorem shows that the way we detect subsumption in practice is correct.

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

theorem
 $c_2 \sqsubseteq c_1 \iff \text{sem } c_2 \models_B \text{sem } c_1$
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* ::
 $(v, d) \text{ conf} \Rightarrow (v, d) \text{ conf} \Rightarrow \text{bool}$
where
 $\text{abstract } c \ c_a \equiv c \sqsubseteq c_a$

7.1.9 Entailment

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

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

lemma
 $\text{sem } c \models_B e \longleftrightarrow (\forall \sigma \in \text{states } c. e \sigma)$
by (*auto simp add : states-def sem-def entails-def*)

end
theory *SymExec*
imports *Conf Labels*
begin

7.2 Symbolic Execution

We model symbolic execution by an inductive predicate *se* which takes two configurations c_1 and c_2 and a label l and evaluates to *true* if and only if c_2 is a *possible result* of the symbolic execution of l from c_1 . We say that c_2 is a possible result because, when l is of the form *Assign v e*, we associate a fresh symbolic variable to the program variable v , but we do not specify how this fresh variable is chosen (see the two assumptions in the third case). We could have model *se* (and *se_star*) by a function producing the new configuration, instead of using inductive predicates. However this would require to provide the two said assumptions in each lemma involving *se*, which is not necessary using a predicate. Modeling symbolic execution in this way has the advantage that it simplifies the following proofs while not requiring additional lemmas.

7.2.1 Definitions of *se* and *se_star*

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

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

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

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

inductive se ::

$(v, d) conf \Rightarrow (v, d) label \Rightarrow (v, d) conf \Rightarrow bool$

where

$se\ c\ Skip\ c$

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

| $fst\ sv = v \implies$

$fresh\text{-}symvar\ sv\ c \implies$

$se\ c\ (Assign\ v\ e)\ (\mid\ store = (store\ c)(v := snd\ sv),$

$pred = pred\ c \cup \{(\lambda\ \sigma.\ \sigma\ sv = (adapt\text{-}aexp\ e\ (store\ c))\ \sigma)\}\ \mid)$

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

inductive $se\text{-}star$:: $(v, d) conf \Rightarrow (v, d) label\ list \Rightarrow (v, d) conf \Rightarrow bool$ **where**

$se\text{-}star\ c\ []\ c$

| $se\ c1\ l\ c2 \implies se\text{-}star\ c2\ ls\ c3 \implies se\text{-}star\ c1\ (l\ \# \ ls)\ c3$

7.2.2 Basic properties of se

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

lemma $se\text{-}sat\text{-}imp\text{-}sat$:

assumes $se\ c\ l\ c'$

assumes $sat\ c'$

shows $sat\ c$

using $assms$ **by** $cases\ (auto\ simp\ add : sat\text{-}def\ conjunct\text{-}def)$

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

lemma $unsat\text{-}imp\text{-}se\text{-}unsat$:

assumes $se\ c\ l\ c'$

assumes $\neg sat\ c$

shows $\neg sat\ c'$

using $assms$ **by** $cases\ (simp\ add : sat\text{-}def\ conjunct\text{-}def)+$

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 .

lemma $[simp]$:

$se\ c\ Skip\ c' = (c' = c)$
by (*simp add : se.simps*)

lemma *se-Assume-eq* :
 $se\ c\ (Assume\ e)\ c' = (c' = (\parallel\ store = store\ c,\ pred = pred\ c \cup \{adapt_bexp\ e\ (store\ c)\}\ \parallel))$
by (*simp add : se.simps*)

lemma *se-Assign-eq* :
 $se\ c\ (Assign\ v\ e)\ c' =$
 $(\exists\ sv.\ fresh_symvar\ sv\ c$
 $\wedge\ fst\ sv = v$
 $\wedge\ c' = (\parallel\ store = (store\ c)(v := snd\ sv),$
 $\quad pred = insert\ (\lambda\sigma.\ \sigma\ sv = adapt_aexp\ e\ (store\ c)\ \sigma)\ (pred\ c)\parallel))$
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 $conjunct\ (pred\ c') =$
 $(\lambda\ \sigma.\ conjunct\ (pred\ c)\ \sigma \wedge adapt_bexp\ e\ (store\ c)\ \sigma)$
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 $\exists\ sv.\ conjunct\ (pred\ c') =$
 $(\lambda\ \sigma.\ conjunct\ (pred\ c)\ \sigma \wedge \sigma\ sv = adapt_aexp\ e\ (store\ c)\ \sigma)$
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 $states\ c' = \{\sigma \in states\ c.\ e\ \sigma\}$
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 *Assign v e*. We want to express the set of states of c' as a function of the set of states of c . Since the proof requires a number of details, we split into two sub lemmas.

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

lemma *states-of-se-assign1* :

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

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

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

proof –

obtain σ_{sym}

where $1 : \text{consistent } \sigma' \sigma_{sym} (\text{store } c')$

and $2 : \text{conjunct } (\text{pred } c') \sigma_{sym}$

using *assms(2) unfolding states-def by blast*

then obtain σ

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

using *consistentI2 by blast*

moreover

have *conjunct (pred c) σ_{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 $\sigma' v = \sigma_{sym} (\text{symvar } v (\text{store } c'))$

using 1 **by** (*simp add : consistent-def*)

moreover

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

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

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

moreover

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

```

using  $\exists$  by (rule adapt-aexp-is-subst)

ultimately
show ?thesis by simp
qed

moreover
have  $\forall x. x \neq v \longrightarrow \sigma' x = \sigma x$ 
proof (intro allI impI)
  fix  $x$ 

  assume  $x \neq v$ 

  moreover
  hence  $\sigma' x = \sigma_{sym} (\text{symvar } x (\text{store } c))$ 
  using assms(1) 1 unfolding consistent-def symvar-def
  by (drule-tac ?x=x in spec) (auto simp add : se-Assign-eq)

  moreover
  have  $\sigma_{sym} (\text{symvar } x (\text{store } c)) = \sigma x$ 
  using  $\exists$  by (auto simp add : consistent-def)

  ultimately
  show  $\sigma' x = \sigma x$  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  $\exists \sigma \in \text{states } c. \sigma' = \sigma (v := e \sigma)$ 
  shows  $\sigma' \in \text{states } c'$ 
proof -
  obtain  $\sigma$ 
  where  $\sigma \in \text{states } c$ 
  and  $\sigma' = \sigma (v := e \sigma)$ 
  using assms(2) by blast

```

```

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

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

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

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

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

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

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

ultimately
show ?thesis using 5 by (simp add: conjunct-def)
qed

ultimately
show ?thesis unfolding states-def by blast
qed

```


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

theorem *states-of-se-assign* :
assumes $se\ c\ (Assign\ v\ e)\ c'$
shows $states\ c' = \{\sigma\ (v := e\ \sigma) \mid \sigma.\ \sigma \in states\ c\}$
using *assms states-of-se-assign1 states-of-se-assign2* **by** *fast*

7.2.3 Monotonicity of se

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

theorem *se-mono-for-sub* :
assumes $se\ c1\ l\ c1'$
assumes $se\ c2\ l\ c2'$
assumes $c2 \sqsubseteq c1$
shows $c2' \sqsubseteq c1'$
using *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 *se-mono-for-states-eq* :
assumes $states\ c1 = states\ c2$
assumes $se\ c1\ l\ c1'$
assumes $se\ c2\ l\ c2'$
shows $states\ c2' = states\ c1'$
using *assms(1)*
se-mono-for-sub[OF assms(2,3)]
se-mono-for-sub[OF assms(3,2)]
by (*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 *se-succs-states* :
assumes $se\ c\ l\ c1$
assumes $se\ c\ l\ c2$
shows $states\ c1 = states\ c2$
using *assms se-mono-for-states-eq* **by** *fast*

7.2.4 Basic properties of *se__star*

Some simplification lemmas for *se-star*.

```
lemma [simp] :  
  se-star c [] c' = (c' = c)  
by (subst se-star.simps) auto
```

```
lemma se-star-Cons :  
  se-star c1 (l # ls) c2 = (∃ c. se c1 l c ∧ se-star c ls c2)  
by (subst (1) se-star.simps) blast
```

```
lemma se-star-one :  
  se-star c1 [l] c2 = se c1 l c2  
using se-star-Cons by force
```

```
lemma se-star-append :  
  se-star c1 (ls1 @ ls2) c2 = (∃ c. se-star c1 ls1 c ∧ se-star c ls2 c2)  
by (induct ls1 arbitrary : c1, simp-all add : se-star-Cons) blast
```

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

```
lemma unsat-imp-se-star-unsat :  
  assumes se-star c ls c'  
  assumes ¬ sat c  
  shows ¬ sat c'  
using assms  
by (induct ls arbitrary : c)  
  (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.

```
lemma se-star-sat-imp-sat :  
  assumes se-star c ls c'  
  assumes sat c'  
  shows sat c  
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 $c2 \sqsubseteq c1$
shows $c2' \sqsubseteq c1'$
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* ::

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

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

lemma *updatable-imp-ex-se-suc* :
assumes *updatable c*
shows $\exists c'. se \ c \ l \ c'$
using *assms*
by (*cases l, simp-all add : se-Assume-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 *store* component has no influence here, since its set of symbolic variables is always a strict subset of the set of symbolic variables (i.e. there always exist fresh symbolic variables for a store). To establish this proof, we need the following intermediate lemma.

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

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

proof –
have *finite (snd ‘ {sv ∈ SV. fst sv = v})*
and *snd ‘ {sv ∈ SV. fst sv = v} ≠ {}*
using *assms by auto*

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

ultimately

```

obtain  $n$ 
where  $n \in \text{snd } \{sv \in SV. \text{fst } sv = v\}$ 
and  $\forall m \in \text{snd } \{sv \in SV. \text{fst } sv = v\}. m \leq n$ 
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 ?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 (Bexp.vars (conjunct (pred c))) (is finite ?V)
  shows updatable c
unfolding updatable-def
proof (intro allI)
  fix  $v$ 

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

  then obtain  $max\text{-}sv$ 
  where  $max\text{-}sv \in ?V$ 
  and  $\text{fst } max\text{-}sv = v$ 
  and  $max : \forall sv' \in \{sv \in ?V. \text{fst } sv = v\}. \text{snd } sv' \leq \text{snd } max\text{-}sv$ 
  using assms finite-symvars-imp-ex-greatest-symvar [of ?V v]
  by blast

  show ?thesis
  using  $max$ 
unfolding fresh-symvar-def symvars-def Store.symvars-def symvar-def
proof (case-tac  $\text{snd } max\text{-}sv \leq \text{store } c \ v, \text{goal-cases}$ )
  case 1 thus ?case by (rule-tac  $?x=(v, \text{Suc } (\text{store } c \ v))$ ) in exI auto
next
  case 2 thus ?case by (rule-tac  $?x=(v, \text{Suc } (\text{snd } max\text{-}sv))$ ) in exI auto

```

```

qed
next
  case 2 thus ?thesis
  by (rule-tac ?x=(v, Suc (store c v)) in exI)
      (auto simp add : fresh-symvar-def symvars-def Store.symvars-def symvar-def)
qed
qed

```

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

```

lemma finite-imp-ex-se-succ :
  assumes finite (pred c)
  assumes  $\forall e \in \text{pred } c. \text{finite } (Bexp.vars \ e)$ 
  shows  $\exists c'. se \ c \ l \ c'$ 
using 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 *along a sequence of labels*, it is not sufficient for the first configuration to be updatable. It must also be such that (all) its successors are updatable. A sufficient condition for this is that the set of variables of its path predicate is finite and that the sub-expression of the label that is executed also has a finite set of variables. Under these assumptions, symbolic execution preserves finiteness of the *pred* component and of the sets of variables of its elements. Thus, successors *se* are also updatable because they also have a path predicate with a finite set of variables. In the following, to prove this we need two intermediate lemmas:

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

```

lemma se-preserves-finiteness1 :
  assumes finite-label l
  assumes se c l c'
  assumes  $\forall e \in \text{pred } c. \text{finite } (Bexp.vars \ e)$ 

```

```

  shows  $\forall e \in \text{pred } c'. \text{finite } (\text{Bexp.vars } e)$ 
proof (cases l)
  case Skip thus ?thesis using assms by (simp add : )
next
  case (Assume e) thus ?thesis
  using 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-symvar sv c
  and fst sv = v
  and  $c' = \langle \text{store} = (\text{store } c)(v := \text{snd } sv),$ 
            $\text{pred} = \text{insert } (\lambda\sigma. \sigma \text{ sv} = \text{adapt-axp } e (\text{store } c) \sigma) (\text{pred } c)\rangle$ 
  using assms(2) se-Assign-eq[of c v e c'] by blast

  moreover
  have finite (Bexp.vars ( $\lambda\sigma. \sigma \text{ sv} = \text{adapt-axp } e (\text{store } c) \sigma$ ))
  proof –
  have finite (Aexp.vars ( $\lambda\sigma. \sigma \text{ sv}$ ))
  by (auto simp add : Aexp.vars-def)

  moreover
  have finite (Aexp.vars (adapt-axp e (store c)))
  using assms(1) Assign finite-vars-imp-finite-adapt-a
  by (auto simp add : finite-label-def)

  ultimately
  show ?thesis using finite-vars-of-a-eq by auto
qed

  ultimately
  show ?thesis using assms by auto
qed

lemma se-preserves-finiteness2 :
  assumes se c l c'
  assumes finite (pred c)
  shows finite (pred c')
using 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 *pred* component of the “initial configuration” is finite, as well as the set of variables of its elements, and that the sub-expression of the label that is executed also has a finite set of variables.

```

lemma finite-imp-ex-se-star-succ :
  assumes finite (pred c)
  assumes  $\forall e \in \text{pred } c. \text{finite } (\text{Bexp.vars } e)$ 
  assumes finite-labels ls
  shows  $\exists c'. \text{se-star } c \text{ ls } c'$ 
using assms
proof (induct ls arbitrary : c, goal-cases)
  case 1 show ?case using se-star.simps by blast
next
  case (2 l ls c)

  then obtain c1 where se c l c1 using finite-imp-ex-se-succ by blast

  hence finite (pred c1)
  and  $\forall e \in \text{pred } c1. \text{finite } (\text{Bexp.vars } e)$ 
  using 2 se-preserves-finiteness1 se-preserves-finiteness2 by fastforce+

  moreover
  have finite-labels ls using 2 by simp

  ultimately
  obtain c2 where se-star c1 ls c2 using 2 by blast

  thus ?case using  $\langle \text{se } c \text{ l } c1 \rangle$  using se-star-Cons by blast
qed

```

7.3 Feasibility of a sequence of labels

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

```

definition feasible :: ('v,'d) conf  $\Rightarrow$  ('v,'d) label list  $\Rightarrow$  bool where
  feasible c ls  $\equiv$   $(\exists c'. \text{se-star } c \text{ ls } c' \wedge \text{sat } c')$ 

```

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

```

lemma feasible-Cons :
  feasible c (l#ls) =  $(\exists c'. \text{se } c \text{ l } c' \wedge \text{sat } c' \wedge \text{feasible } c' \text{ ls})$ 
proof (intro iffI, goal-cases)

```



```

case 1 thus ?case
using se-star-sat-imp-sat by (simp add : feasible-def se-star-Cons) blast
next
case 2 thus ?case
unfolding feasible-def se-star-Cons by blast
qed

```

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

```

lemma subsums-imp-feasible :
assumes finite-labels ls
assumes finite (pred c1)
assumes finite (pred c2)
assumes  $\forall e \in \text{pred } c1. \text{finite } (Bexp.vars e)$ 
assumes  $\forall e \in \text{pred } c2. \text{finite } (Bexp.vars e)$ 
assumes  $c2 \sqsubseteq c1$ 
assumes feasible c2 ls
shows feasible c1 ls
using assms
proof (induct ls arbitrary : c1 c2)
case Nil thus ?case by (simp add : feasible-def sat-sub-by-sat)
next
case (Cons l ls c1 c2)

then obtain  $c2'$  where se c2 l c2'
and sat c2'
and feasible c2' ls
using feasible-Cons by blast

obtain  $c1'$  where se c1 l c1'
using finite-conj[OF Cons(3,5)]
finite-pred-imp-se-updatable
updatable-imp-ex-se-suc
by blast

moreover
hence sat c1'

```

```

using se-mono-for-sub[OF - ⟨se c2 l c2'⟩ Cons(7)]
      sat-sub-by-sat[OF ⟨sat c2'⟩]
by fast

moreover
have feasible c1' ls
proof –

  have finite-label l
  and finite-labels ls using Cons(2) by simp-all

  have finite (pred c1')
  by (rule se-preserves-finiteness2[OF ⟨se c1 l c1'⟩ Cons(3)])

  moreover
  have finite (pred c2')
  by (rule se-preserves-finiteness2[OF ⟨se c2 l c2'⟩ Cons(4)])

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

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

  moreover
  have  $c2' \sqsubseteq c1'$ 
  by (rule se-mono-for-sub[OF ⟨se c1 l c1'⟩ ⟨se c2 l c2'⟩ Cons(7)])

  ultimately
  show ?thesis using Cons(1) ⟨feasible c2' ls⟩ ⟨finite-labels ls⟩ by fast
qed

  ultimately
  show ?case by (auto simp add : feasible-Cons)
qed

```

7.4 Concrete execution

We illustrate our notion of symbolic execution by relating it with *ce*, an inductive predicate describing concrete execution. Unlike symbolic execution, concrete execution describes program behavior given program states, i.e. concrete valuations for program variables. The goal of this section is

to show that our notion of symbolic execution is correct, that is: given two configurations such that one results from the symbolic execution of a sequence of labels from the other, then the resulting configuration represents the set of states that are reachable by concrete execution from the states of the original configuration.

inductive *ce* ::

$('v, 'd) \text{ state} \Rightarrow ('v, 'd) \text{ label} \Rightarrow ('v, 'd) \text{ state} \Rightarrow \text{bool}$

where

$ce \ \sigma \ \text{Skip} \ \sigma$

| $e \ \sigma \Longrightarrow ce \ \sigma \ (\text{Assume } e) \ \sigma$

| $ce \ \sigma \ (\text{Assign } v \ e) \ (\sigma(v := e \ \sigma))$

inductive *ce-star* :: $('v, 'd) \text{ state} \Rightarrow ('v, 'd) \text{ label list} \Rightarrow ('v, 'd) \text{ state} \Rightarrow \text{bool}$ **where**

$ce\text{-star} \ c \ [] \ c$

| $ce \ c1 \ l \ c2 \Longrightarrow ce\text{-star} \ c2 \ ls \ c3 \Longrightarrow ce\text{-star} \ c1 \ (l \# \ ls) \ c3$

lemma [*simp*] :

$ce \ \sigma \ \text{Skip} \ \sigma' = (\sigma' = \sigma)$

by (*auto simp add : ce.simps*)

lemma [*simp*] :

$ce \ \sigma \ (\text{Assume } e) \ \sigma' = (\sigma' = \sigma \wedge e \ \sigma)$

by (*auto simp add : ce.simps*)

lemma [*simp*] :

$ce \ \sigma \ (\text{Assign } v \ e) \ \sigma' = (\sigma' = \sigma(v := e \ \sigma))$

by (*auto simp add : ce.simps*)

lemma *se-as-ce* :

assumes $se \ c \ l \ c'$

shows $\text{states } c' = \{\sigma'. \exists \sigma \in \text{states } c. ce \ \sigma \ l \ \sigma'\}$

using *assms*

by (*cases l*)

(*auto simp add: states-of-se-assume states-of-se-assign*)

lemma [*simp*] :

$ce\text{-star} \ \sigma \ [] \ \sigma' = (\sigma' = \sigma)$

by (*subst ce-star.simps simp*)

lemma *ce-star-Cons* :

$ce\text{-star} \ \sigma1 \ (l \# \ ls) \ \sigma2 = (\exists \sigma. ce \ \sigma1 \ l \ \sigma \wedge ce\text{-star} \ \sigma \ ls \ \sigma2)$

by (*subst (1) ce-star.simps blast*)

```

lemma se-star-as-ce-star :
  assumes se-star c ls c'
  shows  $states\ c' = \{\sigma'. \exists \sigma \in states\ c. ce-star\ \sigma\ ls\ \sigma'\}$ 
using assms
proof (induct ls arbitrary : c)
  case Nil thus ?case by simp
next
  case (Cons l ls c)

  then obtain  $c''$  where  $se\ c\ l\ c''$ 
    and  $se-star\ c''\ ls\ c'$ 
  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  $\sigma'' \in states\ c''$ 
    and  $ce-star\ \sigma''\ ls\ \sigma'$ 
  using Cons(1)  $\langle se-star\ c''\ ls\ c' \rangle$  by blast

  moreover
  then obtain  $\sigma$  where  $\sigma \in states\ c$ 
    and  $ce\ \sigma\ l\ \sigma''$ 
  using  $\langle se\ c\ l\ c'' \rangle$  se-as-ce by blast

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

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

  moreover
  then obtain  $\sigma''$  where  $ce\ \sigma\ l\ \sigma''$ 
    and  $ce-star\ \sigma''\ ls\ \sigma'$ 
  using ce-star-Cons by blast

  ultimately
  show ?case
  using Cons(1)  $\langle se-star\ c''\ ls\ c' \rangle$   $\langle se\ c\ l\ c'' \rangle$  by (auto simp add : se-as-ce)
qed

```

```

qed

end
theory LTS
imports Graph Labels SymExec
begin

```

8 Labelled Transition Systems

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

8.1 Basic definitions

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

```

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

```

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

```

abbreviation init ::
  ('vert,'var,'d,'x) lts-scheme  $\Rightarrow$  'vert
where
  init lts  $\equiv$  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  $\Rightarrow$  ('var,'d) label set
where
  labels lts  $\equiv$  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  $\Rightarrow$  ('vert edge  $\Rightarrow$  ('var,'d) label)  $\Rightarrow$  ('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 *well-formed labelled transition systems*.

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

abbreviation *graph* ::

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

where

graph lts \equiv *rgraph.truncate lts*

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

abbreviation *wf-lts* ::

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

where

wf-lts lts \equiv *wf-rgraph (graph lts)*

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

abbreviation *finite-lts* ::

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

where

finite-lts 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 *lts* from a configuration *c* if it is a sub-path of the underlying graph of *lts* and if it is feasible from the configuration *c*.

abbreviation *feasible-subpath* ::

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

where

feasible-subpath lts pc l1 as l2 \equiv *Graph.subpath lts l1 as l2*
 $\wedge \text{feasible pc (trace as (labelling lts))}$

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.

abbreviation *feasible-subpath-from* ::

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

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

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

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

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

end
theory *SubRel*
imports *Graph*
begin

9 Graphs equipped with a subsumption relation

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

9.1 Basic definitions and properties

9.1.1 Subsumptions and subsumption relations

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

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

A subsumption relation is a set of subsumptions.

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

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

abbreviation $\text{subsumee} ::$

$'v \text{ sub-}t \Rightarrow ('v \times \text{nat})$

where

$\text{subsumee } \text{sub} \equiv \text{fst } \text{sub}$

abbreviation $\text{subsumer} ::$

$'v \text{ sub-}t \Rightarrow ('v \times \text{nat})$

where

$\text{subsumer } \text{sub} \equiv \text{snd } \text{sub}$

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

abbreviation $\text{subsumees} ::$

$'v \text{ sub-rel-}t \Rightarrow ('v \times \text{nat}) \text{ set}$

where

$\text{subsumees } \text{subs} \equiv \text{subsumee } \text{' } \text{subs}$

abbreviation $\text{subsumers} ::$

$'v \text{ sub-rel-}t \Rightarrow ('v \times \text{nat}) \text{ set}$

where

$\text{subsumers } \text{subs} \equiv \text{subsumer } \text{' } \text{subs}$

The two following lemmas will prove useful in the following.

lemma $\text{subsumees-conv} :$

$\text{subsumees } \text{subs} = \{v. \exists v'. (v, v') \in \text{subs}\}$

by *force*

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

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

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

9.2 Well-formed subsumption relation of a graph

9.2.1 Well-formed subsumption relations

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

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

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

locale *wf-sub-rel* = *sub-rel* +
assumes *sub-imp-same-verts* :
 $sub \in subs \implies fst\ (subsumee\ sub) = fst\ (subsumer\ sub)$

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

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

begin

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

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

lemma (in *wf-sub-rel*) *subsumed-by-two-imp* :
assumes $(v, v1) \in subs$
assumes $(v, v2) \in subs$
shows $v1 = v2$
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 $(v, v') \in subs^+$
shows $(v, v') \in subs$
using *tranclD[OF assms] tranclD[of - v' subs]*
rtranclD[of - v' subs]
inter-empty
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* : $vertices\ subs \subseteq Graph.vertices\ g$

```

begin
  lemmas sub-rel-of = related-are-verts

```

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

```

  lemma trancl-sub-rel-of :
    sub-rel-of g (subs+)
  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
  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
  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 $v1$ to $v2$ if these two vertices are equal or if they belong to the transitive closure of the relation. A non-empty sequence is consistent if it is made of consistent sequences whose extremities are linked in the transitive closure of the subsumption relation.

```
fun ces :: ('v × nat) ⇒ ('v × nat) edge list ⇒ ('v × nat) ⇒ 'v sub-rel-t ⇒ bool
where
  ces v1 [] v2 subs = (v1 = v2 ∨ (v1,v2) ∈ subs+)
| ces v1 (e#es) v2 subs = ((v1 = src e ∨ (v1,src e) ∈ subs+) ∧ ces (tgt e) es v2
subs)
```

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

lemma

```
assumes Graph.ces v1 es v2
shows ces v1 es v2 subs
using assms by (induct es arbitrary : v1, auto)
```

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

lemma

```
assumes ces v1 es v2 {}
shows Graph.ces v1 es v2
using assms by (induct es arbitrary : v1, auto)
```

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

lemma *acas-imp-dcas* :

```
assumes (v1,v2) ∈ subs
assumes ces v2 es v subs
shows ces v1 es v subs
using assms by (cases es, simp-all) (intro disjI2, force)+
```

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

lemma *ces-Un* :

```
assumes ces v1 es v2 subs1
```

shows $ces\ v1\ es\ v2\ (subs1\ \cup\ subs2)$
using *assms* **by** (*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 *assms* **by** (*induct es arbitrary : v1, auto simp add : trancl-mono*)

Simplification lemmas for *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 (*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 (*intro iffI, goal-cases*)
case 1 thus ?*case*
by (*induct es1 arbitrary : v1*)
(*simp-all del : split-paired-Ex, blast*)
next
case 2 thus ?*case*
proof (*induct es1 arbitrary : v1*)
case (*Nil v1*)

then obtain *v* **where** $ces\ v1\ []\ v\ subs$
and $ces\ v\ es2\ v2\ subs$
by *blast*

thus ?*case*
unfolding *ces.simps*
proof (*elim disjE, goal-cases*)
case 1 thus ?*case* **by** *simp*
next
case 2 thus ?*case* **by** (*cases es2*) (*simp, intro disjI2, fastforce*)
qed
next
case *Cons* **thus** ?*case* **by** *auto*
qed
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 $v2$ is also a vertex of g .

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

assumes $ces\ v1\ es\ v2\ subs$

assumes $set\ es \subseteq edges\ g$

assumes $v1 \in Graph.vertices\ g$

shows $v2 \in Graph.vertices\ g$

using *assms trancl-sub-rel-of*

unfolding *sub-rel-of-def subsumers-conv vertices-def*

by (*induct es arbitrary : v1*) (*force, (simp del : split-paired-Ex, fast)*)

9.3.2 Sub-paths

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

definition *subpath* ::

$((v \times nat), 'x)\ rgraph\ scheme \Rightarrow (v \times nat) \Rightarrow (v \times nat)\ edge\ list \Rightarrow (v \times nat) \Rightarrow ((v \times nat) \times (v \times nat))\ set \Rightarrow bool$

where

$subpath\ g\ v1\ es\ v2\ subs \equiv sub\ rel\ of\ g\ subs$

$\wedge v1 \in Graph.vertices\ g$

$\wedge ces\ v1\ es\ v2\ subs$

$\wedge set\ es \subseteq edges\ g$

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

abbreviation *subpath-from* ::

$((v \times nat), 'x)\ rgraph\ scheme \Rightarrow (v \times nat) \Rightarrow (v \times nat)\ edge\ list \Rightarrow 'v\ sub\ rel\ t \Rightarrow bool$

where

$subpath\ from\ g\ v\ es\ subs \equiv \exists v'.\ subpath\ g\ v\ es\ v'\ subs$

Simplification lemmas for *SubRel.subpath*.

lemma *Nil-sp* :

$subpath\ g\ v1\ []\ v2\ subs \longleftrightarrow sub\ rel\ of\ g\ subs$

$\wedge v1 \in Graph.vertices\ g$

$\wedge (v1 = v2 \vee (v1, v2) \in subs^+)$

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* :
 $subpath\ g\ v1\ []\ v2\ subs \longleftrightarrow sub-rel-of\ g\ subs$
 $\wedge v1 \in Graph.vertices\ g$
 $\wedge (v1 = v2 \vee (v1, v2) \in subs)$
using *trancl-eq* **by** (*simp add : Nil-sp*)

Simplification lemma for the one-element sequence.

lemma *sp-one* :
shows $subpath\ g\ v1\ [e]\ v2\ subs \longleftrightarrow sub-rel-of\ g\ subs$
 $\wedge (v1 = src\ e \vee (v1, src\ e) \in subs^+)$
 $\wedge e \in edges\ g$
 $\wedge (tgt\ e = v2 \vee (tgt\ e, v2) \in subs^+)$
using *sub-rel-of.trancl-sub-rel-of[of g subs]*
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 $subpath\ g\ v1\ [e]\ v2\ subs \longleftrightarrow sub-rel-of\ g\ subs$
 $\wedge (v1 = src\ e \vee (v1, src\ e) \in subs)$
 $\wedge e \in edges\ g$
 $\wedge (tgt\ e = v2 \vee (tgt\ e, v2) \in subs)$
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 $subpath\ g\ v1\ (e\ \# \ es)\ v2\ subs \longleftrightarrow sub-rel-of\ g\ subs$
 $\wedge (v1 = src\ e \vee (v1, src\ e) \in subs^+)$
 $\wedge e \in edges\ g$
 $\wedge subpath\ g\ (tgt\ e)\ es\ v2\ subs$
using *sub-rel-of.trancl-sub-rel-of[of g subs]*
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* :
 $subpath\ g\ v1\ (e\ \# \ es)\ v2\ subs \longleftrightarrow sub-rel-of\ g\ subs$
 $\wedge (v1 = src\ e \vee (v1, src\ e) \in subs)$
 $\wedge e \in edges\ g$
 $\wedge subpath\ g\ (tgt\ e)\ es\ v2\ subs$
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 *sp-append-one* :
 $subpath\ g\ v1\ (es\ @\ [e])\ v2\ subs \longleftrightarrow subpath\ g\ v1\ es\ (src\ e)\ subs$
 $\wedge\ e \in edges\ g$
 $\wedge\ (tgt\ e = v2 \vee (tgt\ e, v2) \in subs^+)$
unfolding *subpath-def* **by** (*auto simp add : ces-append-one*)

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

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

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

lemma *sp-append* :
 $subpath\ g\ v1\ (es1\ @\ es2)\ v2\ subs \longleftrightarrow$
 $(\exists\ v.\ subpath\ g\ v1\ es1\ v\ subs \wedge subpath\ g\ v\ es2\ v2\ subs)$
proof (*intro iffI, goal-cases*)
case 1 thus *?case*
using *sub-rel-of.ces-imp-ends-vertices*
by (*simp add : subpath-def ces-append*) *blast*
next
case 2 thus *?case*
unfolding *subpath-def*
by (*simp only : ces-append*) *fastforce*
qed

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

lemma *fst-of-sp-is-vert* :
assumes *subpath g v1 es v2 subs*
shows $v1 \in Graph.vertices\ g$
using *assms* **by** (*simp add : subpath-def*)

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

lemma *lst-of-sp-is-vert* :
assumes *subpath g v1 es v2 subs*
shows $v2 \in Graph.vertices\ g$

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 $(v2, v3) \in \text{subs}$
shows *subpath g v1 es v3 subs*
proof (*cases es*)
case *Nil*

moreover
hence $v1 \in \text{Graph.vertices } g$
and $v1 = v2 \vee (v1, v2) \in \text{subs}^+$
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-Nil-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 $(v1, v2) \in \text{subs}$

```

assumes subpath g v1 es v subs
assumes out-edges g v1 = {}
shows  $es = [] \vee \text{subpath } g \ v2 \ es \ v \ subs$ 
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  $v1 \notin \text{subsumees } subs$ 
  assumes out-edges g v1 = {}
  assumes subpath g v1 es v2 subs
  shows  $es = []$ 
using assms transclD 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  $\text{tgt } e \notin \text{subsumees } subs$ 
  assumes out-edges g (tgt e) = {}
  assumes subpath g v1 es v2 subs
  assumes  $e \in \text{set } es$ 
  shows  $\exists es'. es = es' @ [e] \wedge e \notin \text{set } es'$ 
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  $e = e' \vee (e \neq e' \wedge e \in \text{set } es)$  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

```

```

next
  case 2 thus ?case using Cons(1)[of tgt e'] by force
qed
qed

```

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

lemma (in *sub-rel-of*) *sp-ends-in-tgt-imp-mem* :

```

assumes tgt e ∉ Graph.vertices g
assumes v ≠ tgt e
assumes subpath (add-edge g e) v es (tgt e) subs
shows e ∈ set es

```

proof –

```

have tgt e ∉ subsumers subs using assms(1) sub-rel-of by auto

```

```

hence (v,tgt e) ∉ subs+ using tranclD2 by force

```

```

hence es ≠ [] using assms(2,3) by (auto simp add : Nil-sp)

```

```

then obtain es' e' where es = es' @ [e'] by (simp add : neq-Nil-conv2) blast

```

moreover

```

hence e' ∈ edges (add-edge g e) using assms(3) by (auto simp add: subpath-def)

```

moreover

```

have tgt e' = tgt e
using tranclD2 assms(3) ⟨tgt e ∉ subsumers subs⟩ ⟨es = es' @ [e']⟩
by (force simp add : sp-append-one)

```

ultimately

```

show ?thesis using assms(1) unfolding vertices-def image-def by force

```

qed

end

theory *ArcExt*

imports *SubRel*

begin

10 Extending rooted graphs with edges

In this section, we formalize the operation of adding to a rooted graph an edge whose source is already a vertex of the given graph but not its

target. We call this operation an extension of the given graph by adding an edge. This corresponds to an abstraction of the act of adding an edge to the red part of a red-black graph as a result of symbolic execution of the corresponding transition in the LTS under analysis, where all details about symbolic execution would have been abstracted. We then state and prove a number of facts describing the evolution of the set of paths of the given graph, first without considering subsumption links then in the case of rooted graph equipped with a subsumption relation.

10.1 Definition and Basic properties

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

abbreviation *extends* ::

$(v, x) \text{ rgraph-scheme} \Rightarrow v \text{ edge} \Rightarrow (v, x) \text{ rgraph-scheme} \Rightarrow \text{bool}$

where

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

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

lemma *extends-tgt-out-edges* :

assumes *extends* $g \ e \ g'$

shows $\text{out-edges } g' (\text{tgt } e) = \{\}$

using *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 sub-rel-of*)

assumes *extends* $g \ e \ g'$

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

using *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 *sp-in-extends* :

assumes *extends* $g \ e \ g'$

assumes *Graph.subpath* $g \ v1 \ \text{es} \ v2$

shows *Graph.subpath* $g' \ v1 \ \text{es} \ v2$

using *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'

unfolding *is-tree-def* *Ball-def*

proof (*intro allI impI*)

fix v

have $\text{root } g' = \text{root } g$ **using** *assms(2)* **by** *simp*

assume $v \in \text{Graph.vertices } g'$

hence $v \in \text{Graph.vertices } g \vee v = \text{tgt } e$

using *assms(2)* **by** (*auto simp add : vertices-def*)

thus $\exists ! es. \text{path } g' es v$

proof (*elim disjE, goal-cases*)

case *1*

then obtain es

where $\text{Graph.path } g es v$

and $\forall es'. \text{Graph.path } g es' v \longrightarrow es' = es$

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

hence $\text{Graph.path } g' es v$

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

by (*subst <root g' = root g>*)

moreover

have $\forall es'. \text{Graph.path } g' es' v \longrightarrow es' = es$

proof (*intro allI impI*)

fix es'

assume $\text{Graph.path } g' es' v$

thus $es' = es$

proof (*case-tac e \in set es', goal-cases*)

case *1*

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

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

thus *?thesis*
using *assms(2)*
 $\text{Graph.lst-of-sp-is-vert}[OF \langle \text{Graph.path } g \text{ } es \text{ } v \rangle]$
by *simp*

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

ultimately
show *?thesis* **by** *auto*

next
case 2

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

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

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

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

```

assume Graph.path g' es' (tgt e)

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

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

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

then obtain es''
where  $es' = es'' @ [e]$ 
and  $e \notin \text{set } es''$ 
by blast

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

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

hence  $es'' = es$ 
using  $\langle \forall as'. \text{Graph.path } g as' (\text{src } e) \longrightarrow as' = es \rangle$ 
by simp

thus  $es' = es @ [e]$  using  $\langle es' = es'' @ [e] \rangle$  by simp
qed

ultimately
show ?thesis using 2 by auto
qed
qed

```

10.3 Properties of sub-paths in an extension

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

```

lemma sp-in-extends-w-sub :
  assumes extends g a g'
  assumes subpath g v1 es v2 subs
  shows subpath g' v1 es v2 subs
using 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 sub-rel-of) sp-from-old-verts-imp-sp-in-old :
  assumes extends g e g'
  assumes v1 ∈ Graph.vertices g
  assumes v2 ∈ Graph.vertices g
  assumes subpath g' v1 es v2 subs
  shows subpath g v1 es v2 subs
proof –
  have e ∉ set es
  proof (intro notI)
    assume e ∈ set es

    have v2 = tgt e
    proof –
      have tgt e ∉ subsumees subs using sub-rel-of assms(1) by fast

    moreover
    have out-edges g' (tgt e) = {} using assms(1) by (rule extends-tgt-out-edges)

    ultimately
    have  $\exists es'. es = es' @ [e] \wedge e \notin set es'$ 
    using assms(4)  $\langle e \in set es \rangle$ 
    by (intro sp-through-de-decomp)

    then obtain es' where es = es' @ [e] e ∉ set es' by blast

    hence tgt e = v2 ∨ (tgt e, v2) ∈ subs+
    using assms(4) by (simp add : sp-append-one)

    thus ?thesis using  $\langle tgt e \notin subsumees subs \rangle$  tranclD[of tgt e v2 subs] by force
  qed

thus False using assms(1,3) by simp

```


qed

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[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 $v \neq \text{tgt } e$
shows $\exists es'. es = es' @ [e] \wedge e \notin \text{set } es' \wedge \text{subpath } g \ v \ es' \ (\text{src } e) \ \text{subs}$
proof –
obtain es' **where** $es = es' @ [e]$ **and** $e \notin \text{set } es'$
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 $v \in \text{Graph.vertices } g$
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)  $\langle es = es' @ [e] \rangle$  by (simp add : sp-append-one)

ultimately
show ?thesis
using assms(1) sub-rel-of  $\langle e \notin set\ es' \rangle$ 
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 subsumption relations

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

11.1 Definition

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

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

- the subsumee must have no out-edges.

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

abbreviation *extends* ::

$((v \times nat), x) \text{ rgraph-scheme} \Rightarrow v \text{ sub-rel-t} \Rightarrow v \text{ sub-t} \Rightarrow v \text{ sub-rel-t} \Rightarrow \text{bool}$

where

$\text{extends } g \text{ subs } sub \text{ subs}' \equiv ($
 $\quad \text{subsumee } sub \neq \text{subsumer } sub$
 $\wedge \text{fst } (\text{subsumee } sub) = \text{fst } (\text{subsumer } sub)$
 $\wedge \text{subsumee } sub \in \text{Graph.vertices } g$
 $\wedge \text{subsumee } sub \notin \text{subsumers } subs$
 $\wedge \text{subsumee } sub \notin \text{subsumees } subs$
 $\wedge \text{subsumer } sub \in \text{Graph.vertices } g$
 $\wedge \text{subsumer } sub \notin \text{subsumees } subs$
 $\wedge \text{out-edges } g (\text{subsumee } sub) = \{\}$
 $\wedge \text{subs}' = \text{subs} \cup \{sub\})$

11.2 Properties of extensions

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

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

lemma (in *sub-rel-of*)

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

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

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* $g \text{ subs } sub \text{ subs}'$

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 ∈ subsumees subs'

hence v = subsumee sub ∨ v ∈ subsumees subs using assms by auto

thus ∃! v'. (v,v') ∈ subs'
proof (elim disjE, goal-cases)
  case 1 show ?thesis
    unfolding Ex1-def
    proof (rule-tac ?x=subsumer sub in exI, intro conjI)
      show (v, subsumer sub) ∈ subs' using 1 assms by simp
    next
      have v ∉ subsumees subs using assms 1 by auto

      thus ∀ v'. (v, v') ∈ subs' → v' = subsumer sub
      using assms by auto force
    qed
  next
    case 2

    then obtain v' where (v,v') ∈ subs by auto

    hence v ≠ subsumee sub
    using assms unfolding subsumees-conv by (force simp del : split-paired-All split-paired-Ex)

    show ?thesis
    using assms
      ⟨v ≠ subsumee sub⟩
      ⟨(v,v') ∈ subs⟩ subsumed-by-one
    unfolding subsumees-conv 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 wf-sub-rel assms by auto
qed

```

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

```

lemma (in wf-sub-rel-of) extends-imp-wf-sub-rel-of :
  assumes extends g subs sub subs'
  shows wf-sub-rel-of g subs'
using sub-rel-of assms

```

wf-sub-rel.extends-imp-wf-sub-rel[*OF wf-sub-rel assms*]
by (*simp add : wf-sub-rel-of-def sub-rel-of-def*)

11.3 Properties of sub-paths in an extension

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

lemma *sp-in-extends* :
assumes *extends g subs sub subs'*
assumes *subpath g v1 es v2 subs*
shows *subpath g v1 es v2 subs'*
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 subsumee of the new subsumption, then the sub-paths starting at any other vertex.

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

lemma (**in** *wf-sub-rel-of*) *sp-in-extends-imp1* :
assumes *extends g subs (v1,v2) subs'*
assumes *subpath g v1 es v subs'*
shows $es = [] \vee \text{subpath } g \ v2 \ es \ v \ subs'$
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 subsumee are either:

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

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

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

```
fun uses-sub ::
  ('v × nat) ⇒ ('v × nat) edge list ⇒ ('v × nat) ⇒ (('v × nat) × ('v × nat)) ⇒
  bool
where
  uses-sub v1 [] v2 sub = (v1 ≠ v2 ∧ sub = (v1,v2))
| uses-sub v1 (e#es) v2 sub = (v1 ≠ src e ∧ sub = (v1,src e) ∨ uses-sub (tgt e)
  es v2 sub)
```

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

```
lemma
  assumes uses-sub v1 es v2 sub
  assumes ces v1 es v2 subs
  shows sub ∈ subs+
using assms by (induction es arbitrary : v1) fastforce+
```

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

```
lemma (in wf-sub-rel)
  assumes uses-sub v1 es v2 sub
  assumes ces v1 es v2 subs
  shows sub ∈ subs
using assms trancl-eq by (induction 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 extends g subs (v,v') subs'
  assumes subpath g v1 es v2 subs
  shows ¬ uses-sub v1 es v2 (v,v')
proof (intro notI)
  assume uses-sub v1 es v2 (v,v')
```

```
moreover
have ces v1 es v2 subs using assms(2) by (simp add : subpath-def)
```

```
ultimately
have (v,v') ∈ subs+ by (induction 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 $v1$ to $v2$ after the extension. Then, the empty sequence is a sub-path leading from $v1$ to $v2$ in the graph before the extension if and only if $(v1, v2)$ is not the new subsumption.

```

lemma (in wf-sub-rel-of) sp-Nil-in-extends-imp :
  assumes extends g subs (v,v') subs'
  assumes subpath g v1 [] v2 subs'
  shows subpath g v1 [] v2 subs  $\longleftrightarrow$  (v1  $\neq$  v  $\vee$  v2  $\neq$  v')
proof (intro iffI, goal-cases)
  case 1 thus ?case
  using assms(1)
    extends-and-sp-imp-not-using-sub[OF assms(1), of v1 [] v2]
  by auto
next
  case 2

```

```

  have v1 = v2  $\vee$  (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  $\vee$  (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 :

```

```

assumes extends g subs (v,v') subs'
assumes subpath g v1 es v2 subs'
assumes  $\neg$  uses-sub v1 es v2 (v,v')
shows subpath g v1 es v2 subs
using sub-rel-of assms extends-imp-wf-sub-rel-of
by (induction es arbitrary : v1)
    (auto simp add : sp-Nil-in-extends-imp wf-sub-rel-of.sp-Cons sp-Cons)

```

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

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

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

```

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

shows subpath g v1 es v2 subs  $\vee$  ( $\exists$  es1 es2. es = es1 @ es2
     $\wedge$  es1  $\neq$  []
     $\wedge$  subpath g v1 es1 v subs
     $\wedge$  subpath g v es2 v2 subs')

    (is ?P es v1)

```

```

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  $\vee$  (v1, src edge)  $\in$  subs'
and edge  $\in$  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  $\neg$  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'⟩

```

```

      <uses-sub (tgt edge) es v2 (v,v')>
      extends-and-sp-imp-not-using-sub[OF assms(1)]
    by fast
  next
    case (2 es1 es2) thus ?thesis
    using <es = es1 @ es2>
      <subpath g v1 [edge] (tgt edge) subs>
      <subpath g v es2 v2 subs'>
    by (intro disjI2, rule-tac ?x=edge # es1 in exI) (auto simp add : sp-Cons)
  qed
qed
qed
next
  case 2 thus ?thesis
  using assms(1,2) by (simp add : sp-in-extends-not-using-sub)
qed

end
theory RB
imports LTS ArcExt SubExt
begin

```

12 Red-Black Graphs

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

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

12.1 Basic Definitions

12.1.1 The type of Red-Black Graphs

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

- *red* is the red graph, called *red part*, which represents the unfolding of the black part. Its vertices are indexed black vertices,
- *black* is the original LTS, the *black part*,

- *subs* is the subsumption relation over the vertices of *red*,
- *init-conf* is the initial configuration,
- *confs* is a function associating configurations to the vertices of *red*,
- *marked* is a function associating truth values to the vertices of *red*. We use it to represent the fact that a particular configuration (associated to a red location) is known to be unsatisfiable,
- *strengthenings* is a function associating boolean expressions over program variables to vertices of the red graph. Those boolean expressions can be seen as invariants that the configuration associated to the “strengthened” red vertex has to model.

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

record (*'vert, 'var, 'd*) *pre-RedBlack* =
red :: (*'vert × nat*) *rgraph*
black :: (*'vert, 'var, 'd*) *lts*
subs :: *'vert sub-rel-t*
init-conf :: (*'var, 'd*) *conf*
confs :: (*'vert × nat*) ⇒ (*'var, 'd*) *conf*
marked :: (*'vert × nat*) ⇒ *bool*
strengthenings :: (*'vert × nat*) ⇒ (*'var, 'd*) *bexp*

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

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

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

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

We extend this idea to sequences of edges.

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

12.1.2 Well-formed and finite red-black graphs

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

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

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

```
locale wf-pre-RedBlack = pre-RedBlack +  
  assumes red-wf : wf-rgraph (red prb)  
  assumes black-wf : wf-lts (black prb)  
  assumes consistent-roots : fst (root (red prb)) = root (black prb)  
  assumes ui-re-are-be :  $e \in \text{edges}(\text{red } prb) \implies \text{ui-edge } e \in \text{edges}(\text{black } prb)$   
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 :  $\forall e \in \text{pred}(\text{init-conf } prb). \text{finite}(Bexp.vars$   
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* ::

$$\begin{aligned} & ('vert, 'var, 'd) \text{ pre-RedBlack} \\ & \Rightarrow ('vert \times nat) \text{ edge} \\ & \Rightarrow ('var, 'd) \text{ conf} \\ & \Rightarrow ('vert, 'var, 'd) \text{ pre-RedBlack} \Rightarrow bool \end{aligned}$$

where

$$\begin{aligned} \text{se-extends } prb \ re \ c \ prb' & \equiv \\ & \text{ui-edge } re \in \text{edges } (black \ prb) \\ & \wedge \text{ArcExt.extends } (red \ prb) \ re \ (red \ prb') \\ & \wedge \text{src } re \notin \text{subsumees } (subs \ prb) \\ & \wedge \text{se } (confs \ prb \ (src \ re)) \ (\text{labelling } (black \ prb) \ (\text{ui-edge } re)) \ c \\ & \wedge \text{prb}' = \{\!| \text{red} \quad = \text{red } prb', \\ & \quad \text{black} \quad = \text{black } prb, \\ & \quad \text{subs} \quad = \text{subs } prb, \\ & \quad \text{init-conf} = \text{init-conf } prb, \\ & \quad \text{confs} \quad = (\text{confs } prb) \ (\text{tgt } re := c), \\ & \quad \text{marked} \quad = (\text{marked } prb) (\text{tgt } re := \text{marked } prb \ (src \ re)), \end{aligned}$$

$strengthenings = strengthenings\ prb\]$

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

abbreviation *se-extends2* ::

$('vert, 'var, 'd)\ pre-RedBlack \Rightarrow ('vert, 'var, 'd)\ pre-RedBlack \Rightarrow bool$

where

$se-extends2\ prb\ prb' \equiv$
 $\exists\ re \in edges\ (red\ prb').$
 $ui-edge\ re \in edges\ (black\ prb)$
 $\wedge\ ArcExt.extends\ (red\ prb)\ re\ (red\ prb')$
 $\wedge\ src\ re \notin subsumees\ (subs\ prb)$
 $\wedge\ se\ (confs\ prb\ (src\ re))\ (labelling\ (black\ prb)\ (ui-edge\ re))\ (confs\ prb'\ (tgt\ re))$
 $\wedge\ black\ prb' = black\ prb$
 $\wedge\ subs\ prb' = subs\ prb$
 $\wedge\ init-conf\ prb' = init-conf\ prb$
 $\wedge\ confs\ prb' = (confs\ prb)\ (tgt\ re := confs\ prb'\ (tgt\ re))$
 $\wedge\ marked\ prb' = (marked\ prb)(tgt\ re := marked\ prb\ (src\ re))$
 $\wedge\ strengthenings\ prb' = strengthenings\ prb$

12.2.2 Extension by marking

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

abbreviations *mark-extends* ::

$('vert, 'var, 'd)\ pre-RedBlack \Rightarrow ('vert \times nat) \Rightarrow ('vert, 'var, 'd)\ pre-RedBlack \Rightarrow bool$

where

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

$$\begin{aligned} \text{strengthenings} &= \text{strengthenings } prb, \\ \dots &= \text{more } prb \end{aligned}$$

12.2.3 Extension by subsumption

The abstract operation of introducing a subsumption link.

abbreviation *subsum-extends* ::

$$('vert, 'var, 'd) \text{ pre-RedBlack} \Rightarrow 'vert \text{ sub-}t \Rightarrow ('vert, 'var, 'd) \text{ pre-RedBlack} \Rightarrow \text{bool}$$

where

$$\begin{aligned} \text{subsum-extends } prb \text{ sub } prb' &\equiv \\ &SubExt.\text{extends } (red \text{ } prb) \text{ (subs } prb) \text{ sub (subs } prb') \\ &\wedge \neg \text{ marked } prb \text{ (subsumer } sub) \\ &\wedge \neg \text{ marked } prb \text{ (subsumee } sub) \\ &\wedge \text{ confs } prb \text{ (subsumee } sub) \sqsubseteq \text{ confs } prb \text{ (subsumer } sub) \\ &\wedge prb' = (\{ red \quad = red \text{ } prb, \\ &\quad \quad \quad black \quad = black \text{ } prb, \\ &\quad \quad \quad subs \quad = insert \text{ } sub \text{ (subs } prb), \\ &\quad \quad \quad \text{init-conf} = \text{init-conf } prb, \\ &\quad \quad \quad \text{confs} \quad = \text{confs } prb, \\ &\quad \quad \quad \text{marked} \quad = \text{marked } prb, \\ &\quad \quad \quad \text{strengthenings} = \text{strengthenings } prb, \\ &\quad \quad \quad \dots \quad = \text{more } prb \end{aligned}$$

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* ::

$$\begin{aligned} &('vert, 'var, 'd) \text{ pre-RedBlack} \\ &\Rightarrow ('vert \times nat) \\ &\Rightarrow ('var, 'd) \text{ conf} \\ &\Rightarrow ('vert, 'var, 'd) \text{ pre-RedBlack} \\ &\Rightarrow \text{bool} \end{aligned}$$

where

$$\begin{aligned} \text{abstract-extends } prb \text{ } rv \text{ } c_a \text{ } prb' &\equiv \\ &rv \in \text{red-vertices } prb \\ &\wedge \neg \text{ marked } prb \text{ } rv \\ &\wedge \text{ out-edges } (red \text{ } prb) \text{ } rv = \{\} \\ &\wedge rv \notin \text{subsumees } (subs \text{ } prb) \\ &\wedge \text{ abstract } (confs \text{ } prb \text{ } rv) \text{ } c_a \\ &\wedge c_a \models_c (\text{strengthenings } prb \text{ } rv) \end{aligned}$$

$$\begin{aligned}
& \wedge \text{finite } (\text{pred } c_a) \\
& \wedge (\forall e \in \text{pred } c_a. \text{finite } (\text{vars } e)) \\
\wedge \text{prb}' = & \langle \! \langle \text{red} = \text{red } \text{prb}, \\
& \text{black} = \text{black } \text{prb}, \\
& \text{subs} = \text{subs } \text{prb}, \\
& \text{init-conf} = \text{init-conf } \text{prb}, \\
& \text{confs} = (\text{confs } \text{prb})(rv := c_a), \\
& \text{marked} = \text{marked } \text{prb}, \\
& \text{strengthenings} = \text{strengthenings } \text{prb}, \\
& \dots = \text{more } \text{prb} \! \rangle \rangle
\end{aligned}$$

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* ::

$$\begin{aligned}
& ('vert, 'var, 'd) \text{pre-RedBlack} \\
& \Rightarrow ('vert \times \text{nat}) \\
& \Rightarrow ('var, 'd) \text{bexp} \\
& \Rightarrow ('vert, 'var, 'd) \text{pre-RedBlack} \\
& \Rightarrow \text{bool}
\end{aligned}$$

where

$$\begin{aligned}
& \text{strengthen-extends } \text{prb } rv \ e \ \text{prb}' \equiv \\
& \quad rv \in \text{red-vertices } \text{prb} \\
& \quad \wedge rv \notin \text{subsumees } (\text{subs } \text{prb}) \\
& \quad \wedge \text{confs } \text{prb } rv \models_c e \\
\wedge \text{prb}' = & \langle \! \langle \text{red} = \text{red } \text{prb}, \\
& \text{black} = \text{black } \text{prb}, \\
& \text{subs} = \text{subs } \text{prb}, \\
& \text{init-conf} = \text{init-conf } \text{prb}, \\
& \text{confs} = \text{confs } \text{prb}, \\
& \text{marked} = \text{marked } \text{prb}, \\
& \text{strengthenings} = (\text{strengthenings } \text{prb})(rv := (\lambda \sigma. (\text{strengthenings } \text{prb} \\
rv) \sigma \wedge e \sigma)), \\
& \dots = \text{more } \text{prb} \! \rangle \rangle
\end{aligned}$$

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* ::

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

where

base :

$\text{fst } (\text{root } (\text{red } \text{prb})) = \text{init } (\text{black } \text{prb}) \quad \Longrightarrow$
 $\text{edges } (\text{red } \text{prb}) = \{\}$ \Longrightarrow
 $\text{subs } \text{prb} = \{\}$ \Longrightarrow
 $(\text{confs } \text{prb}) (\text{root } (\text{red } \text{prb})) = \text{init-conf } \text{prb} \Longrightarrow$
 $\text{marked } \text{prb} = (\lambda \text{ rv. False}) \quad \Longrightarrow$
 $\text{strengthenings } \text{prb} = (\lambda \text{ rv. } (\lambda \sigma. \text{True})) \quad \Longrightarrow \text{RedBlack } \text{prb}$

| *se-step* :

$\text{RedBlack } \text{prb} \quad \Longrightarrow$
 $\text{se-extends } \text{prb } \text{re } \text{p}' \text{prb}' \quad \Longrightarrow \text{RedBlack } \text{prb}'$

| *mark-step* :

$\text{RedBlack } \text{prb} \quad \Longrightarrow$
 $\text{mark-extends } \text{prb } \text{rv } \text{prb}' \quad \Longrightarrow \text{RedBlack } \text{prb}'$

| *subsum-step* :

$\text{RedBlack } \text{prb} \quad \Longrightarrow$
 $\text{subsum-extends } \text{prb } \text{sub } \text{prb}' \quad \Longrightarrow \text{RedBlack } \text{prb}'$

| *abstract-step* :

$\text{RedBlack } \text{prb} \quad \Longrightarrow$
 $\text{abstract-extends } \text{prb } \text{rv } \text{c}_a \text{prb}' \quad \Longrightarrow \text{RedBlack } \text{prb}'$

| *strengthen-step* :

$\text{RedBlack } \text{prb} \quad \Longrightarrow$
 $\text{strengthen-extends } \text{prb } \text{rv } \text{e } \text{prb}' \quad \Longrightarrow \text{RedBlack } \text{prb}'$

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 $\text{RedBlack } \text{prb}$

shows $\text{loop-free } (\text{red } \text{prb})$

using *assms* **by** $(\text{induct } \text{prb}) \text{ auto}$

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

lemma

assumes $\text{RedBlack } \text{prb}$

assumes $\text{re} \in \text{edges } (\text{red } \text{prb})$

shows $tgt\ re \neq\ root\ (red\ prb)$
using *assms* **by** (*induct prb*) (*auto simp add : vertices-def*)

Red edges are specific versions of black edges.

lemma *ui-re-is-be* :
assumes *RedBlack prb*
assumes $re \in edges\ (red\ prb)$
shows $ui\text{-}edge\ re \in edges\ (black\ prb)$
using *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.

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

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

lemma *consistent-roots* :
assumes *RedBlack prb*
shows $fst\ (root\ (red\ prb)) = init\ (black\ prb)$
using *assms* **by** (*induct prb*) *auto*

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

lemma
assumes *RedBlack prb*
shows $is\text{-}tree\ (red\ prb)$
using *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.

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

thus *?thesis*
using *assms consistent-roots ui-re-is-be*
by (*auto simp add : wf-pre-RedBlack-def*)

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

```

qed

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 (subsumee 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)*

using *assms(1) Cons(1) 2 by simp*

ultimately

show *?case by simp*

qed

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 $\forall e \in \text{pred}(\text{confs } prb \ rv). \text{finite}(Bexp.vars \ e)$

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  $\text{finite-label } (\text{labelling } (\text{black } prb) (\text{ui-edge } re))$ 
  using  $\text{se-step}$  by  $(\text{auto simp add : finite-RedBlack-def})$ 

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

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

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

```

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

```

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

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

  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  $\exists c'. se (confs prb rv) l c'$ 
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 $\exists c'. \text{se-star } (\text{confs } prb \text{ } rv) \text{ } ls \text{ } c'$
using *finite-RedBlack* *assms*
finite-imp-ex-se-star-succ[*of confs prb rv ls*]
finite-pred[*OF assms(1), of rv*]
finite-pred-constr-symvars[*OF assms(1), of rv*]
unfolding *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** *finite-RedBlack*) *sp-imp-ex-se-star-succ* :
assumes *RedBlack prb*
assumes *subpath (red prb) rv1 res rv2 (subs prb)*
shows $\exists c. \text{se-star}$
 $(\text{confs } prb \text{ } rv1)$
 $(\text{trace } (ui\text{-es } res) \text{ } (\text{labelling } (black \text{ } prb)))$
 c
using *finite-RedBlack* *assms* *ex-se-star-succ*
by (*simp* *add* : *subpath-def* *finite-RedBlack-def*)

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

lemma (**in** *finite-RedBlack*)
assumes *RedBlack prb*
assumes *rv* \in *red-vertices prb*
shows *updatable (confs prb rv)*
using *finite-RedBlack* *assms*
finite-conj[*OF finite-pred*[*OF assms(1)*]
finite-pred-constr-symvars[*OF assms(1)*]]
finite-pred-imp-se-updatable
unfolding *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 *sub-subsumed* :
assumes *RedBlack prb*
assumes *sub* \in *subs prb*
shows *confs prb (subsumee sub) \sqsubseteq confs prb (subsumer sub)*
using *assms*
proof (*induct prb*)
case *base* **thus** ?*case* **by** *simp*
next
case (*se-step prb re c' prb'*)

moreover


```

hence  $sub \in subs\ prb$  by auto

hence  $subsumee\ sub \in red\text{-}vertices\ prb$ 
and  $subsumer\ sub \in red\text{-}vertices\ prb$ 
  using se-step(1) subs-sub-rel-of
  unfolding sub-rel-of-def by fast+

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

ultimately
show ?case by auto
next
  case mark-step thus ?case by simp
next
  case (subsum-step prb sub prb') thus ?case by auto
next
  case (abstract-step prb rv ca prb')

hence  $rv \neq subsumee\ sub$  by auto

show ?case
proof (case-tac rv = subsumer sub)
  assume  $rv = subsumer\ sub$ 

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

  ultimately
  show ?thesis
    using abstract-step
      subsums-trans[of confs prb (subsumee sub)
        confs prb (subsumer sub)
        confs prb' (subsumer sub)]
    by (simp add : subsums-refl)
  next
    assume  $rv \neq subsumer\ sub$  thus ?thesis using abstract-step  $\langle rv \neq subsumee\ sub \rangle$  by simp
  qed
next
  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 $\text{subpath } (\text{red } prb) \text{ } rv1 \ [] \ rv2 \ (\text{subs } prb) =$
 $(rv1 \in \text{red-vertices } prb \wedge (rv1 = rv2 \vee (rv1, rv2) \in (\text{subs } prb)))$

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 $\text{subpath } (\text{red } prb) \text{ } rv1 \ [re] \ rv2 \ (\text{subs } prb) =$
 $(\text{sub-rel-of } (\text{red } prb) \ (\text{subs } prb)$
 $\wedge (rv1 = \text{src } re \vee (rv1, \text{src } re) \in (\text{subs } prb))$
 $\wedge re \in \text{edges } (\text{red } prb) \wedge (\text{tgt } re = rv2 \vee (\text{tgt } re, rv2) \in (\text{subs } prb)))$

using *assms subs-wf-sub-rel-of wf-sub-rel-of.sp-one* **by** *fast*

lemma *rb-sp-Cons* :

assumes *RedBlack prb*

shows $\text{subpath } (\text{red } prb) \text{ } rv1 \ (re \ \# \ res) \ rv2 \ (\text{subs } prb) =$
 $(\text{sub-rel-of } (\text{red } prb) \ (\text{subs } prb)$
 $\wedge (rv1 = \text{src } re \vee (rv1, \text{src } re) \in \text{subs } prb)$
 $\wedge re \in \text{edges } (\text{red } prb)$
 $\wedge \text{subpath } (\text{red } prb) \ (\text{tgt } re) \ res \ rv2 \ (\text{subs } prb))$

using *assms subs-wf-sub-rel-of wf-sub-rel-of.sp-Cons* **by** *fast*

lemma *rb-sp-append-one* :

assumes *RedBlack prb*

shows $\text{subpath } (\text{red } prb) \text{ } rv1 \ (res \ @ \ [re]) \ rv2 \ (\text{subs } prb) =$
 $(\text{subpath } (\text{red } prb) \text{ } rv1 \ res \ (\text{src } re) \ (\text{subs } prb)$
 $\wedge re \in \text{edges } (\text{red } prb)$
 $\wedge (\text{tgt } re = rv2 \vee (\text{tgt } re, rv2) \in \text{subs } prb))$

using *assms subs-wf-sub-rel 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 $c \sqsubseteq (confs prb rv2)$

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 $rv1 \in red\text{-vertices } prb'$

and $rv2 \in red\text{-vertices } prb'$

using *fst-of-sp-is-vert[OF se-step(4)]*

lst-of-sp-is-vert[OF se-step(4)]

by *simp-all*

hence $rv1 \in red\text{-vertices } prb \wedge rv1 \neq tgt\ re \vee rv1 = tgt\ re$

and $rv2 \in red\text{-vertices } prb \wedge rv2 \neq tgt\ re \vee rv2 = tgt\ re$

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*

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

next

case 2

hence $\exists res'. res = res' @ [re]$
 $\wedge re \notin set\ res'$
 $\wedge subpath\ (red\ prb)\ rv1\ res'\ (src\ re)\ (subs\ prb)$
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 $c' \sqsubseteq (confs\ prb\ (src\ re))$ **using** *se-step* **by** *fastforce*

thus *?thesis*
using *se-step <rv1 ≠ 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  $se\text{-}step\ 3$ 
     $sub\text{-}rel\text{-}of.sp\text{-}from\text{-}tgt\text{-}in\text{-}extends\text{-}is\text{-}Nil$ 
     $[OF\ sub\text{-}sub\text{-}rel\text{-}of[OF\ se\text{-}step(1)],\ of\ re\ red\ prb'\ res\ rv2]$ 
     $rb\text{-}Nil\text{-}sp[OF\ RedBlack.se\text{-}step[OF\ se\text{-}step(1,3)],\ of\ rv1\ rv2]$ 
  by  $auto$ 

  hence  $rv1 \in subsume\ (subs\ prb)$  using  $se\text{-}step(3)$  by  $force$ 

  thus  $?thesis$ 
    using  $se\text{-}step\ \langle rv1 = tgt\ re \rangle\ sub\text{-}sub\text{-}rel\text{-}of[OF\ se\text{-}step(1)]$ 
    by  $(auto\ simp\ add : sub\text{-}rel\text{-}of\text{-}def)$ 
qed

ultimately
show  $?thesis$  by  $simp$ 
next

  case  $4$ 

  moreover
  hence  $res = []$ 
    using  $se\text{-}step$ 
       $sub\text{-}rel\text{-}of.sp\text{-}from\text{-}tgt\text{-}in\text{-}extends\text{-}is\text{-}Nil$ 
       $[OF\ sub\text{-}sub\text{-}rel\text{-}of[OF\ se\text{-}step(1)],\ of\ re\ red\ prb'\ res\ rv2]$ 
    by  $auto$ 

  ultimately
  show  $?thesis$  using  $se\text{-}step$  by  $(simp\ add : subsums\text{-}refl)$ 
qed

next

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

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

  have  $RB' : RedBlack\ prb'$  by  $(rule\ RedBlack.subsum\text{-}step[OF\ subsum\text{-}step(1,3)])$ 

```

```

show ?case
proof (case-tac rv1 = subsumee sub)

  assume rv1 = subsumee sub

  hence res = []  $\vee$  subpath (red prb') (subsumer sub) res rv2 (subs prb')
    using subsum-step(3,4)
    wf-sub-rel-of.sp-in-extends-imp1 [ OF subs-wf-sub-rel-of[OF sub-
sum-step(1)],
    of subsumee sub subsumer sub ]
    by simp

  thus ?thesis
proof (elim disjE)

    assume res = []

    hence rv1 = rv2  $\vee$  (rv1,rv2)  $\in$  (subs prb')
      using subsum-step rb-Nil-sp[OF RB'] by fast

    thus ?thesis
proof (elim disjE)

      assume rv1 = rv2
      thus ?thesis
        using subsum-step(5)  $\langle$ res = [] $\rangle$ 
        by (simp add : subsums-refl)
    next

      assume (rv1, rv2)  $\in$  (subs prb')
      thus ?thesis
        using subsum-step(5)  $\langle$ res = [] $\rangle$ 
        sub-subsumed[OF RB', of (rv1,rv2)]
        by simp
      qed

    next

    assume subpath (red prb') (subsumer sub) res rv2 (subs prb')

    thus ?thesis
    using subsum-step(5)
proof (induct res arbitrary : rv2 c rule : rev-induct, goal-cases)

```

case (1 *rv2 c*)

have *rv2 = subsumer sub*

proof –

have (*subsumer sub,rv2*) \notin *subs prb'*

proof (*intro notI*)

assume (*subsumer sub,rv2*) \in *subs prb'*

hence *subsumer sub* \in *subsumees (subs prb')* **by** *force*

moreover

have *subsumer sub* \in *subsumers (subs prb')*

using *subsum-step(3)* **by** *force*

ultimately

show *False*

using *subs-wf-sub-rel[OF RB']*

unfolding *wf-sub-rel-def*

by *auto*

qed

thus *?thesis* **using** *1(1) rb-Nil-sp[OF RB']* **by** *auto*

qed

thus *?case*

using *subsum-step(3) 1(2) <rv1 = subsumee sub>* **by** *simp*

next

case (2 *re res rv2 c*)

hence *A : subpath (red prb') (subsumer sub) res (src re) (subs prb')*

and *B : subpath (red prb') (src re) [re] (tgt re) (subs prb')*

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

obtain *c'*

where *C : se-star (confs prb' rv1) (trace (ui-es res) (labelling (black prb')))*

c'

and *D : 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 $subsum-step(6-8)$
 $\langle subpath\ (red\ prb')\ (src\ re)\ [re]\ (tgt\ re)\ (subs\ prb') \rangle$
 $RB'\ finite-RedBlack.ex-se-succ[of\ prb'\ src\ re]$
unfolding $finite-RedBlack-def$
by $(simp\ add : se-star-one\ fst-of-sp-is-vert)\ blast$

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

moreover
have $c'' \sqsubseteq confs\ prb' (tgt\ re)$
proof –
have $subpath\ (red\ prb)\ (src\ re)\ [re]\ (tgt\ re)\ (subs\ prb)$
proof –
have $src\ re \in red-vertices\ prb'$
and $tgt\ re \in red-vertices\ prb'$
and $re \in edges\ (red\ prb')$
using B **by** $(auto\ simp\ add : vertices-def\ sp-one)$

hence $src\ re \in red-vertices\ prb$
and $tgt\ re \in red-vertices\ prb$
and $re \in edges\ (red\ prb)$
using $subsum-step(3)$ **by** $auto$

thus $?thesis$
using $subs-sub-rel-of[OF\ subsum-step(1)]$
by $(simp\ add : sp-one)$
qed

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

moreover
have $confs\ prb' (tgt\ re) \sqsubseteq confs\ prb'\ rv2$
proof –
have $tgt\ re = rv2 \vee (tgt\ re,rv2) \in subs\ prb'$
using $2(2)\ rb-sp-append-one[OF\ RB']$ **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

```

```

and   res1 ≠ []
and   subpath (red prb) rv1 res1 (subsumee sub) (subs prb)
and   subpath (red prb') (subsumee sub) res2 rv2 (subs prb')

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

then have c ⊆ c2
proof -
  have c1 ⊆ confs prb' (subsumee sub)
    using subsum-step(2,3,6-8)
          ⟨subpath (red prb) rv1 res1 (subsumee sub) (subs prb)⟩
          ⟨se-star (confs prb' rv1) t-res1 c1⟩
    by (auto simp add : t-res1-def t-res2-def)

  thus ?thesis
    using ⟨se-star c1 t-res2 c⟩
          ⟨se-star (confs prb' (subsumee sub)) t-res2 c2⟩
          se-star-mono-for-sub
    by fast
qed

moreover

have c2 ⊆ confs prb' rv2
  using ⟨subpath (red prb') (subsumee sub) res2 rv2 (subs prb')⟩
        ⟨se-star (confs prb' (subsumee sub)) t-res2 c2⟩
  unfolding t-res2-def
  proof (induct res2 arbitrary : rv2 c2 rule : rev-induct, goal-cases)

case (1 rv2 c2)

  hence subsumee sub = rv2 ∨ (subsumee sub, rv2) ∈ subs prb'
  using rb-Nil-sp[OF RB'] by simp

thus ?case
proof (elim disjE)

```

```

assume subsumee sub = rv2
thus ?thesis
  using 1(2) by (simp add : subsums-refl)
next

  assume (subsumee sub, rv2)  $\in$  subs prb'
  thus ?thesis
    using 1(2)
      sub-subsumed[OF RB', of (subsumee sub, rv2)]
    by simp
  qed
next

case (2 re res2 rv2 c2)

have A : subpath (red prb') (subsumee sub) res2 (src re) (subs prb')
and B : subpath (red prb') (src re) [re] rv2 (subs prb')
  using 2(2) subs-wf-sub-rel[OF RB'] subs-wf-sub-rel-of[OF RB']
  by (simp-all only: wf-sub-rel.sp-append-one)
    (simp add : wf-sub-rel-of.sp-one wf-sub-rel-of-def)

obtain c3
where C : se-star (confs prb' (subsumee sub))
  (trace (ui-es res2) (labelling (black prb')))
  (c3)
and D : se c3 (labelling (black prb') (ui-edge re)) c2
  using 2(3) subsum-step(6-8) RB'
  (finite-RedBlack.ex-se-succ[of prb' src re])
  by (simp add : se-star-append-one) blast

obtain c4
where E : se (confs prb' (src re)) (labelling (black prb') (ui-edge re)) c4
  using subsum-step(6-8) RB' B
  (finite-RedBlack.ex-se-succ[of prb' src re])
  unfolding finite-RedBlack-def
  by (simp add : fst-of-sp-is-vert se-star-append) blast

have c2  $\sqsubseteq$  c4
proof –
  have c3  $\sqsubseteq$  confs prb' (src re) using 2(1) A C by fast

  thus ?thesis using D E se-mono-for-sub by fast

```

```

qed

moreover
have  $c_4 \sqsubseteq \text{confs } prb' (tgt \ re)$ 
proof -
  have  $\text{subpath } (red \ prb) (src \ re) [re] (tgt \ re) (subs \ prb)$ 
  proof -
    have  $src \ re \in red\text{-vertices } prb'$ 
    and  $tgt \ re \in red\text{-vertices } prb'$ 
    and  $re \in edges (red \ prb')$ 
    using  $B$  by (auto simp add : vertices-def sp-one)

    hence  $src \ re \in red\text{-vertices } prb$ 
    and  $tgt \ re \in red\text{-vertices } prb$ 
    and  $re \in edges (red \ prb)$ 
    using  $\text{subsum-step}(3)$  by auto

  thus ?thesis
    using  $\text{subs-sub-rel-of}[OF \ \text{subsum-step}(1)]$ 
    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  $tgt \ re = rv2 \vee (tgt \ re, \ rv2) \in (subs \ prb')$ 
  using  $\text{subsum-step } 2 \ \text{rb-sp-append-one}[OF \ RB', \ \text{of } \text{subsumee } sub \ res2$ 
re]
    by (auto simp add : vertices-def subpath-def)

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

```

```

    qed

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

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

next
case (abstract-step prb rv ca 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 (case-tac rv2 = rv)

  assume rv2 = rv

  hence confs prb rv2  $\sqsubseteq$  confs prb' rv2
    using abstract-step by (simp add : abstract-def)

  moreover
  have c  $\sqsubseteq$  confs prb rv2
    using abstract-step 2 by auto

  ultimately
  show ?thesis using subsums-trans by fast
next
  assume rv2  $\neq$  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 rv  $\in$  red-vertices prb
  assumes sat (confs prb rv)
  shows  $\neg$  marked prb rv
using assms
proof (induct prb arbitrary : rv)
  case base thus ?case by simp
next
  case (se-step prb re c prb')

  hence rv  $\in$  red-vertices prb  $\vee$  rv = tgt re by (auto simp add : vertices-def)

```

```

thus ?case
proof (elim disjE, goal-cases)
  case 1
  moreover
  hence  $rv \neq tgt\ re$  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 sat (confs prb (src re)) using se-step(3,5) se-sat-imp-sat by auto

  ultimately
  show ?thesis using se-step(2,3) by (elim conjE) auto
qed
next
  case (mark-step prb rv' prb')

  moreover
  hence  $rv \neq rv'$  and  $(rv,rv') \notin subs\ prb$ 
    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  $rv \in red\ vertices\ prb$ 
  assumes marked prb rv
  shows  $\neg sat\ (confs\ prb\ rv)$ 
using assms
proof (induct prb arbitrary : rv)
  case base thus ?case by simp

```

```

next
  case (se-step prb re c prb')

  hence  $rv \in \text{red-vertices } prb \vee rv = \text{tgt } re$  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  $\neg \text{sat } (\text{confs } prb \ rv)$  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  $\neg \text{sat } (\text{confs } prb \ (\text{src } re))$  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 (subsumee sub)
using assms
proof (induct prb)
  case base thus ?case by simp
next
  case (se-step prb re c prb')

  moreover
  hence subsumee 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(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 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 $rv1 = rv2 \vee (rv1, rv2) \in \text{subs } prb$ **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 subsumee-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 $re \in \text{edges } (red \text{ } prb)$ **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 subsumee-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* ::

$(\text{'vert}, \text{'var}, \text{'d}, \text{'x}) \text{ pre-RedBlack-scheme} \Rightarrow (\text{'vert} \times \text{nat}) \text{ set}$

where

$$\begin{aligned} \text{fringe } prb \equiv & \{rv \in \text{red-vertices } prb. \\ & rv \notin \text{subsumees } (subs \text{ } prb) \wedge \\ & \neg \text{marked } prb \text{ } rv \quad \wedge \\ & \text{ui-edge } \text{' } \text{out-edges } (red \text{ } prb) \text{ } rv \subset \text{out-edges } (black \text{ } prb) \text{ } (fst \text{ } rv)\} \end{aligned}$$

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 $\text{edges } (red \text{ } prb) = \{\}$

assumes $\text{subs } prb = \{\}$

assumes $\text{marked } prb = (\lambda rv. \text{False})$

assumes $\text{out-edges } (black \text{ } prb) \text{ } (fst \text{ } (root \text{ } (red \text{ } prb))) \neq \{\}$

shows $\text{fringe } prb = \{root \text{ } (red \text{ } prb)\}$

using *assms* **by** $(\text{auto simp add : fringe-def vertices-def})$

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 $\text{sub-rel-of } (red \text{ } prb) \text{ } (subs \text{ } prb)$

assumes $\text{se-extends } prb \text{ } re \text{ } c' \text{ } prb'$

assumes $\neg \text{marked } prb \text{ } (src \text{ } re)$

assumes $\text{ui-edge } \text{' } (\text{out-edges } (red \text{ } prb') \text{ } (src \text{ } re)) \subset \text{out-edges } (black \text{ } prb) \text{ } (fst \text{ } (src \text{ } re))$

assumes $\text{out-edges } (black \text{ } prb) \text{ } (fst \text{ } (tgt \text{ } re)) \neq \{\}$

shows $\text{fringe } prb' = \text{fringe } prb \cup \{tgt \text{ } re\}$

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

```

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

  moreover
  hence  $rv \in \text{red-vertices } prb \vee rv = \text{tgt } re$ 
  using  $\text{assms}(2)$  by (auto simp add : fringe-def vertices-def)

  ultimately
  show ?case using  $\text{assms}(2)$  by (auto simp add : fringe-def)
next
  case (2 rv)

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

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

  moreover
  have  $\text{wi-edge } \langle \text{out-edges } (\text{red } prb') \text{ } rv \rangle \subset \text{out-edges } (\text{black } prb') \text{ } (\text{fst } rv)$ 
  using 2
  proof (elim disjE)
    assume  $rv \in \text{fringe } prb$ 

    thus ?thesis
    proof (case-tac  $rv = \text{src } re$ )
      assume  $rv = \text{src } re$  thus ?thesis using  $\text{assms}(2,4)$  by auto
    next
      assume  $rv \neq \text{src } re$  thus ?thesis
      using  $\text{assms}(2)$   $\langle rv \in \text{fringe } prb \rangle$ 
      by (auto simp add : fringe-def)
    qed
  next
    assume  $rv = \text{tgt } re$  thus ?thesis
    using  $\text{assms}(2,5)$  extends-tgt-out-edges[of re red prb red prb'] by (elim conjE)
  auto
  qed

```

```

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

  moreover
  hence  $rv \neq$  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  $\neg$  marked prb (src re)
  assumes  $\neg$  (ui-edge ' (out-edges (red prb') (src re))  $\subset$  out-edges (black prb) (fst (src re)))
  assumes out-edges (black prb) (fst (tgt re))  $\neq$  {}
  shows fringe prb' = fringe prb - {src re}  $\cup$  {tgt re}
unfolding set-eq-iff Un-iff singleton-iff Diff-iff
proof (intro allI iffI, goal-cases)
  case (1 rv)

  hence  $rv \in$  red-vertices prb  $\vee$   $rv =$  tgt re
  and  $rv \neq$  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** $assms(2)$ **by** $(\text{auto simp add : fringe-def vertices-def})$

moreover

have $rv \notin \text{subsumees } (subs \ prb')$

using 2

proof $(\text{elim } disjE)$

assume $rv \in \text{fringe } prb \wedge rv \neq \text{src } re$

thus $?thesis$ **using** $assms(2)$ **by** $(\text{auto simp add : fringe-def})$

next

assume $rv = \text{tgt } re$ **thus** $?thesis$

using $assms(1,2)$ **unfolding** sub-rel-of-def **by** fastforce

qed

moreover

have $ui\text{-edge } (out\text{-edges } (red \ prb') \ rv) \subset out\text{-edges } (black \ prb') \ (fst \ rv)$

using 2

proof $(\text{elim } disjE)$

assume $rv \in \text{fringe } prb \wedge rv \neq \text{src } re$ **thus** $?thesis$

using $assms(2)$ **by** $(\text{auto simp add : fringe-def})$

next

assume $rv = \text{tgt } re$ **thus** $?thesis$

using $assms(2,5)$ $\text{extends-tgt-out-edges}[of \ re \ red \ prb \ red \ prb']$ **by** $(\text{elim } conjE)$

auto

qed

moreover

have $\neg \text{marked } prb' \ rv$

using 2

proof $(\text{elim } disjE, \text{goal-cases})$

case 1

moreover

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

ultimately

show $?thesis$

using $assms \ 1$ **by** $(\text{auto simp add : fringe-def})$

next

case 2 **thus** $?thesis$ **using** $assms$ **by** *auto*

qed

ultimately

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

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

```



```

    case 4 thus ?case by auto
  qed
qed

```

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))  $\subset$  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  $\in$  red-vertices prb'
  and rv  $\neq$  tgt re
  using assms(1) by (auto simp add : fringe-def vertices-def)

```

moreover

```

  have ui-edge ' (out-edges (red prb') rv)  $\subset$  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  $\neq$  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 \neg (*ui-edge* ' (*out-edges (red prb')* (*src re*)) \subset *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* \in *red-vertices prb* **and** *rv* \neq *src re*
using *1 assms* **by** (*auto simp add : fringe-def vertices-def*)

moreover
have \neg *marked prb rv*
proof (*intro notI*)
assume *marked prb rv*

have *marked prb' rv*
proof -
have *rv* \neq *tgt re* **using** *assms(1)* \langle *rv* \in *red-vertices prb* \rangle **by** *auto*

thus *?thesis* **using** *assms(1)* \langle *marked prb rv* \rangle **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* \in *red-vertices prb'* **using** *assms(1)* **by** (*auto simp add : fringe-def*)

vertices-def)

moreover

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

moreover

have $ui\text{-edge } (out\text{-edges } (red \text{ } prb') \text{ } rv) \subset out\text{-edges } (black \text{ } prb') \text{ } (fst \text{ } rv)$
using $2 \text{ assms}(1)$ **by** $(\text{auto simp add : fringe-def})$

moreover

have $\neg \text{marked } prb' \text{ } rv$

proof –

have $rv \neq tgt \text{ } re$ **using** $assms(1) \ 2$ **by** $(\text{auto simp add : fringe-def})$

thus $?thesis$ **using** $assms(1) \ 2$ **by** $(\text{auto simp add : fringe-def})$

qed

ultimately

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

qed

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

lemma *subsumE-fringe* :

assumes $subsum\text{-extends } prb \text{ } sub \text{ } prb'$

shows $fringe \text{ } prb' = fringe \text{ } prb - \{subsumee \text{ } sub\}$

using $assms$ **by** $(\text{auto simp add : fringe-def})$

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* ::

$(\text{'vert}, \text{'var}, \text{'d}, \text{'x}) \text{ pre-RedBlack-scheme} \Rightarrow (\text{'vert} \times \text{nat}) \Rightarrow \text{'vert edge list set}$

where

$$\begin{aligned}
& \text{RedBlack-subpaths-from } prb \text{ } rv \equiv \\
& \quad ui\text{-es } \{res. \exists rv'. \text{subpath } (red \text{ } prb) \text{ } rv \text{ } res \text{ } rv' \text{ } (subs \text{ } prb) \wedge \neg \text{marked } prb \text{ } rv'\} \\
& \cup \{ui\text{-es } res1 \text{ } @ \text{ } bes2 \\
& \quad | \text{ } res1 \text{ } bes2. \exists rv1. rv1 \in \text{fringe } prb \\
& \quad \quad \wedge \text{subpath } (red \text{ } prb) \text{ } rv \text{ } res1 \text{ } rv1 \text{ } (subs \text{ } prb) \\
& \quad \quad \wedge \neg (\exists res21 \text{ } bes22. bes2 = ui\text{-es } res21 \text{ } @ \text{ } bes22 \\
& \quad \quad \quad \wedge res21 \neq [] \\
& \quad \quad \quad \wedge \text{subpath-from } (red \text{ } prb) \text{ } rv1 \text{ } res21 \text{ } (subs \text{ } prb)) \\
& \quad \wedge \text{Graph.subpath-from } (black \text{ } prb) \text{ } (fst \text{ } rv1) \text{ } bes2\}
\end{aligned}$$

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* \Rightarrow *'vert edge list set*

where

RedBlack-paths *prb* \equiv *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* \in *RedBlack-subpaths-from* *prb* (*root* (*red* *prb*))

thus *bes* \in *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 ∈ 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 ?thesis using assms ⟨rv ∈ fringe prb⟩ by (simp add : fringe-def
vertices-def)
  qed
next
  fix bes
  assume bes ∈ Graph.paths (black prb)
  show bes ∈ 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 ?thesis
    unfolding RedBlack-subpaths-from-def Un-iff image-def Bex-def 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 ¬ marked prb (root (red prb)) using assms(5) by simp
    next
      show bes = ui-es []
      using ⟨bes ∈ Graph.paths (black prb)⟩
      ⟨out-edges (black prb) (init (black prb)) = {}⟩
      by (cases bes) (auto simp add : Graph.sp-Cons)
    qed
  next
  assume out-edges (black prb) (init (black prb)) ≠ {}
  show ?thesis
  unfolding RedBlack-subpaths-from-def Un-iff mem-Collect-eq
  proof (intro disjI2, rule-tac ?x=[] in exI, rule-tac ?x=bes in exI,
intro conjI, goal-cases)

```

```

    case 1 show ?case by simp
  next
    case 2 show ?case
      unfolding Bex-def
      proof (rule-tac ?x=root (red prb) in exI, intro conjI, goal-cases)

        show root (red prb) ∈ fringe prb
          using assms(1-3,5) ‹out-edges (black prb) (init (black
prb)) ≠ {}›
          fringe-of-empty-red1
          by fastforce
      next
        show subpath (red prb)(root (red prb))([])(root (red prb))(subs
prb)
          using subs-sub-rel-of[OF RedBlack.base[OF assms(1-6)]]
          by (simp add : subpath-def vertices-def sub-rel-of-def)
      next
        case 3 show ?case
          proof (intro notI, elim exE conjE)
            fix res21 bes22 rv
            assume bes = ui-es res21 @ bes22
            and res21 ≠ []
            and subpath (red prb) (root (red prb)) res21 rv (subs
prb)

            moreover
              hence res21 = [] using assms by (simp add :
subpath-def)

            ultimately show False by (elim notE)
          qed
      next
        case 4 show ?case
          using assms ‹bes ∈ Graph.paths (black prb)› by simp
      qed
    qed
  qed
qed

```

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* \subseteq *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 lose 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* \in *red-vertices prb*
shows *feasible-subpaths-from (black prb) (confs prb rv) (fst rv)*
 \subseteq *RedBlack-subpaths-from prb rv*
using *assms finite-RedBlack*
proof (*induct prb arbitrary : rv*)

case (*base prb rv*)

moreover

hence *rv = root (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 $out\text{-}edges\ (black\ prb)\ (fst\ (root\ (red\ prb))) = \{\} \vee$
 $ui\text{-}edge\ 'out\text{-}edges(red\ prb)(root\ (red\ prb)) \subset out\text{-}edges(black\ prb)(fst\ (root$
 $(red\ prb)))$
using *base by auto*

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

next

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

have $RB' : RedBlack\ prb'$ **by** $(rule\ RedBlack.se\text{-}step[OF\ se\text{-}step(1,3)])$

show *?case*
unfolding *subset-iff*
proof $(intro\ allI\ impI)$

fix *bes*

assume $bes \in feasible\text{-}subpaths\text{-}from\ (black\ prb')\ (confs\ prb'\ rv)\ (fst\ rv)$

have $rv \in red\text{-}vertices\ prb \vee rv = tgt\ re$
using $se\text{-}step(3,4)$ **by** $(auto\ simp\ add : vertices\text{-}def)$

thus $bes \in RedBlack\text{-}subpaths\text{-}from\ prb'\ rv$
proof $(elim\ disjE)$

assume $rv \in red\text{-}vertices\ prb$

moreover
hence $rv \neq tgt\ re$ **using** $se\text{-}step$ **by** *auto*

ultimately
have $bes \in RedBlack\text{-}subpaths\text{-}from\ prb\ rv$
using $se\text{-}step\ \langle bes \in feasible\text{-}subpaths\text{-}from\ (black\ prb')\ (confs\ prb'\ rv)$
 $(fst\ rv)\rangle$
by *fastforce*

thus *?thesis*
apply $(subst\ (asm)\ RedBlack\text{-}subpaths\text{-}from\text{-}def)$

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 \neg *marked prb rv'*

moreover

hence \neg *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-subs[of re red prb red prb' rv res rv' subs*

prb]

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 : \neg (\exists res21 bes22. bes2 = ui-es res21 @ bes22*
 \wedge *res21 \neq []*
 \wedge *subpath-from (red prb) rv1 res21 (subs prb))*

and *F : Graph.subpath (black prb) (fst rv1) bes2 bl*

hence *rv1 \neq 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=res1 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-w-subs)
        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=res1@[re] in exI)
          apply (intro conjI)
          apply (rule-tac ?x=tgt re in exI)
          proof (intro conjI)
            show subpath (red prb') rv (res1 @ [re]) (tgt re) (subs prb')

```

```

using se-step(3) ⟨rv1 = src re⟩ C
      sp-in-extends-w-subs[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  $\neg$  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  $\neg$  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

```

```

show subpath (red prb') rv (res1 @ [re]) (tgt re) (subs prb')
  using se-step(3) ⟨rv1 = src re⟩ C
    sp-in-extends-w-subs[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  $\text{fringe-def mem-Collect-eq}$ 
  proof ( $\text{intro conjI}$ )
    show  $rv1 \in \text{red-vertices } prb'$ 
  using  $\text{se-step}(3) B$  by ( $\text{auto simp add : fringe-def vertices-def}$ )
  next
    show  $rv1 \notin \text{subsumees } (\text{subs } prb')$ 
    using  $\text{se-step}(3) B$  by ( $\text{auto simp add : fringe-def}$ )
  next
    show  $\neg \text{marked } prb' rv1$ 
    using  $B \text{ se-step}(3) \langle rv1 \neq \text{tgt } re \rangle \langle rv1 = \text{src } re \rangle$ 
    by ( $\text{auto simp add : fringe-def}$ )
  next
    have  $be \notin \text{ui-edge 'out-edges (red } prb') rv1$ 
    proof ( $\text{intro notI}$ )
      assume  $be \in \text{ui-edge 'out-edges (red } prb') rv1$ 

      then obtain  $re'$  where  $be = \text{ui-edge } re'$ 
        and  $re' \in \text{out-edges (red } prb') rv1$ 

      by  $\text{blast}$ 

    show  $\text{False}$ 
    using  $E$ 
    apply ( $\text{elim notE}$ )
    apply ( $\text{rule-tac } ?x=[re'] \text{ in } exI$ )
    apply ( $\text{rule-tac } ?x=bes2' \text{ in } exI$ )
    proof ( $\text{intro conjI}$ )
      show  $bes2 = \text{ui-es } [re'] @ bes2'$ 
      using  $\langle bes2 = be \# bes2' \rangle \langle be = \text{ui-edge } re' \rangle$  by  $\text{simp}$ 
    next
      show  $[re'] \neq []$  by  $\text{simp}$ 
    next
      have  $re' \in \text{edges (red } prb)$ 
      using  $\text{se-step}(3) \langle rv1 = \text{src } re \rangle \langle re' \in \text{out-edges (red } prb') rv1 \rangle$ 
       $\langle be \neq \text{ui-edge } re \rangle \langle be = \text{ui-edge } re' \rangle$ 
      by ( $\text{auto simp add : vertices-def}$ )

      thus  $\text{subpath-from (red } prb) rv1 [re'] (\text{subs } prb)$ 
      using  $\langle re' \in \text{out-edges (red } prb') rv1 \rangle$ 
       $\text{subs-sub-rel-of}[OF \text{ se-step}(1)]$ 
      by ( $\text{rule-tac } ?x=\text{tgt } re' \text{ in } exI$ )
      ( $\text{simp add : rb-sp-one}[OF \text{ se-step}(1)]$ )
    qed

```

```

      qed

      moreover
      have be ∈ out-edges (black prb) (fst rv1)
      using F ⟨bes2 = be # bes2'⟩ by (simp add : Graph.sp-Cons)

      ultimately
      show ui-edge ' out-edges (red prb') rv1 ⊂ out-edges (black
prb') (fst rv1)
      using se-step(3) red-OA-subset-black-OA[OF RB', of rv1]
by auto

      qed
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'))
apply (intro notI)
apply (elim exE conjE)
proof -
  fix res21 bes22 rv3
  assume bes2 = ui-es res21 @ bes22
  and res21 ≠ []
  and subpath (red prb') rv1 res21 rv3 (subs prb')
  moreover
  then obtain re' res21' where res21 = re' # res21'
      and be = ui-edge re'
      using ⟨bes2 = be # bes2'⟩ unfolding neq-Nil-conv by (elim
exE) simp
  ultimately
  have re' ∈ edges (red prb') by (simp add : sp-Cons)
  moreover
  have re' ∉ edges (red prb)
  using E
  apply (intro notI)
  apply (elim notE)
  apply (rule-tac ?x=[re'] in exI)
  apply (rule-tac ?x=bes2' in exI)
  proof (intro conjI)
    show bes2 = ui-es [re'] @ bes2'
    using ⟨bes2 = be # bes2'⟩ ⟨be = ui-edge re'⟩ by simp
  next

```

```

    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 exI)
      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) ⟨be ≠ ui-edge re⟩ ⟨be = ui-edge re'⟩ by auto
  qed
next
  show Graph.subpath-from (black prb') (fst rv1) bes2
    using se-step(3) F by auto
  qed
qed
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 exI)
  apply (rule-tac ?x=bes2 in exI)
  apply (intro conjI, goal-cases)
  proof -
    show bes = ui-es res1 @ bes2 by (rule ⟨bes = ui-es res1 @ bes2⟩)
  next
    case 2 show ?case
      apply (rule-tac ?x=rv1 in exI)
      proof (intro conjI, goal-cases)
        show rv1 ∈ fringe prb'
          using se-step(3) B ⟨rv1 ≠ src re⟩ ⟨rv1 ≠ tgt re⟩
            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-subs)

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 subsume-es-conv

```



```

by fast
      qed
      thus ?thesis using se-step(3) ⟨re' ∈ edges (red prb)⟩ by
simp
      qed

      show False
      using E
      apply (elim notE)
      apply (rule-tac ?x=[re'] in exI)
      apply (rule-tac ?x=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) rv1 [re'] (subs prb)
      using se-step(1)
        ⟨rv1 = src re' ∨ (rv1, src re') ∈ subs prb⟩
        ⟨re' ∈ edges (red prb)⟩
        rb-sp-one subs-sub-rel-of
      by fast
      qed
    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 (intro disjI1)
apply (rule-tac ?x=[] in exI)
proof (intro conjI, rule-tac ?x=tgt re in exI, intro conjI)
  show subpath (red prb') rv [] (tgt re) (subs prb')
    using se-step(3) ⟨rv = tgt re⟩ rb-Nil-sp[OF RB'] by (auto simp add
: vertices-def)
  next
    have sat (confs prb' (tgt re))
      using ⟨bes ∈ feasible-subpaths-from (black prb') (confs prb' rv) (fst
rv)⟩
        ⟨rv = tgt re⟩ se-star-sat-imp-sat
      by (auto simp add : feasible-def)

    thus ¬ marked prb' (tgt re)
      using se-step(3) sat-not-marked[OF RB', of tgt re]
      by (auto simp add : vertices-def)
  next
    show bes = ui-es []
      using se-step(3) ⟨rv = tgt re⟩ ⟨out-edges (black prb) (fst (tgt re)) =
{}⟩
        ⟨bes ∈ feasible-subpaths-from (black prb') (confs prb' rv) (fst rv)⟩
      by (cases bes) (auto simp add : Graph.sp-Cons)
  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=[] in exI)
apply (rule-tac ?x=bes in exI)
proof (intro conjI, goal-cases)
  show bes = ui-es [] @ bes by simp
next
  case 2
  show ?case
  apply (rule-tac ?x=rv in exI)
  proof (intro conjI)
    have ¬ marked prb (src re)
    proof –
      have sat (confs prb' (tgt re))
        using ⟨bes ∈ feasible-subpaths-from (black prb') (confs prb' rv) (fst
rv)⟩
          ⟨rv = tgt re⟩ se-star-sat-imp-sat
        by (auto simp add : feasible-def)
    
```

```

hence sat (confs prb' (src re))
  using se-step se-sat-imp-sat by auto

moreover
have src re  $\neq$  tgt re using se-step by auto

ultimately
have sat (confs prb (src re))
  using se-step(3) by (auto simp add : vertices-def)

thus ?thesis
  using se-step sat-not-marked[OF se-step(1), of src re] by fast
qed

thus rv  $\in$  fringe prb'
  using se-step(3)  $\langle rv = tgt re \rangle$  out-edges (black prb) (fst (tgt re))  $\neq$ 
    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

show subpath (red prb') rv [] rv (subs prb')
  using se-step(3)  $\langle rv = tgt re \rangle$  subs-sub-rel-of[OF RB]
  by (auto simp add : subpath-def vertices-def)

next

show  $\neg (\exists res21 bes22. bes = ui-es res21 @ bes22$ 
   $\wedge res21 \neq []$ 
   $\wedge subpath-from (red prb') rv res21 (subs prb'))$ 

proof (intro notI, elim exE conjE)
  fix res1 bes22 rv'

  assume bes = ui-es res1 @ bes22
  and res1  $\neq []$ 
  and subpath (red prb') rv res1 rv' (subs prb')

  have out-edges (red prb') (tgt re)  $\neq \{\}$   $\vee$  tgt re  $\in$  subsumees (subs prb')
  proof –
    obtain re' res2 where res1 = re'#res2
      using  $\langle res1 \neq [] \rangle$  unfolding neq-Nil-conv by blast

```

hence $rv = \text{src } re' \vee (rv, \text{src } re') \in \text{subs } prb$
using $se\text{-step}(3) \langle \text{subpath } (red \text{ } prb') \text{ } rv \text{ } res1 \text{ } rv' \text{ } (subs \text{ } prb') \rangle$
 $rb\text{-sp}\text{-Cons}[OF \text{ } RB', \text{ of } rv \text{ } re' \text{ } res2 \text{ } rv']$
by *auto*

thus *?thesis*
proof (*elim disjE*)
assume $rv = \text{src } re'$

moreover
hence $re' \in \text{out-edges } (red \text{ } prb') \text{ } (tgt \text{ } re)$
using $\langle \text{subpath } (red \text{ } prb') \text{ } rv \text{ } res1 \text{ } rv' \text{ } (subs \text{ } prb') \rangle$
 $\langle res1 = re' \# res2 \rangle \langle rv = tgt \text{ } re \rangle$
by (*auto simp add : sp-Cons*)

ultimately
show *?thesis* **using** $se\text{-step}(3)$ **by** *auto*

next
assume $(rv, \text{src } re') \in \text{subs } prb$

hence $tgt \text{ } re \in \text{red-vertices } prb$
using $se\text{-step}(3) \langle rv = tgt \text{ } re \rangle \text{ subs-sub-rel-of}[OF \text{ } se\text{-step}(1)]$
unfolding *sub-rel-of-def* **by** *force*

thus *?thesis* **using** $se\text{-step}(3)$ **by** *auto*
qed
qed

thus *False*
proof (*elim disjE*)
assume $\text{out-edges } (red \text{ } prb') \text{ } (tgt \text{ } re) \neq \{\}$
thus *?thesis* **using** $se\text{-step}(3)$
by (*auto simp add : vertices-def image-def*)

next
assume $tgt \text{ } re \in \text{subsumees } (subs \text{ } prb')$

hence $tgt \text{ } re \in \text{red-vertices } prb$
using $se\text{-step}(3) \text{ subs-sub-rel-of}[OF \text{ } se\text{-step}(1)]$
unfolding *subsumees-conv sub-rel-of-def* **by** *fastforce*

thus *?thesis* **using** $se\text{-step}(3)$ **by** (*auto simp add : vertices-def*)
qed
qed
next

```

      show Graph.subpath-from (black prb') (fst rv) bes
        using se-step(3)
          ⟨bes ∈ feasible-subpaths-from (black prb') (confs prb' rv) (fst rv)⟩
        by simp
    qed
  qed
  qed
  qed
  qed
next

case (mark-step prb rv2 prb' 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 bes
    assume bes ∈ feasible-subpaths-from (black prb') (confs prb' rv1) (fst rv1)
    then obtain c where se-star (confs prb rv1) (trace bes (labelling (black prb)))
  c
      and sat c
      using mark-step(3) ⟨bes ∈ feasible-subpaths-from (black prb') (confs prb'
rv1) (fst rv1)⟩
      by (simp add : feasible-def) blast

    have bes ∈ RedBlack-subpaths-from prb rv1
      using mark-step(2)[of rv1] mark-step(3-7)
        ⟨bes ∈ feasible-subpaths-from (black prb') (confs prb' rv1) (fst rv1)⟩
      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)

      fix res rv3
      assume bes = ui-es res
      and subpath (red prb) rv1 res rv3 (subs prb)
      and ¬ 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=res in exI,intro conjI)

```

```

show  $\exists rv'. \text{subpath } (red \text{ } prb') \text{ } rv1 \text{ } res \text{ } rv' \text{ } (subs \text{ } prb') \wedge \neg \text{marked } prb' \text{ } rv'$ 
apply (rule-tac ?x=rv3 in exI)
proof (intro conjI)
  show  $\text{subpath } (red \text{ } prb') \text{ } rv1 \text{ } res \text{ } rv3 \text{ } (subs \text{ } prb')$ 
    using mark-step(3)  $\langle \text{subpath } (red \text{ } prb) \text{ } rv1 \text{ } res \text{ } rv3 \text{ } (subs \text{ } prb) \rangle$ 
    by auto
next

  show  $\neg \text{marked } prb' \text{ } rv3$ 
  proof -

    have sat (confs prb rv3)
      proof -
        have  $c \sqsubseteq \text{confs } prb \text{ } rv3$ 
          using mark-step(1)
             $\langle \text{subpath } (red \text{ } prb) \text{ } rv1 \text{ } res \text{ } rv3 \text{ } (subs \text{ } prb) \rangle$ 
             $\langle \text{bes} = \text{ui-es } res \rangle$ 
             $\langle \text{se-star } (confs \text{ } prb \text{ } rv1) \text{ } (\text{trace } \text{bes } (\text{labelling } (\text{black } prb))) \rangle$ 
        c)
           $\langle \text{finite-RedBlack } prb \rangle$ 
          finite-RedBlack.SE-rel
          by simp

        thus ?thesis
          using  $\langle \text{se-star } (confs \text{ } prb \text{ } rv1) \text{ } (\text{trace } \text{bes } (\text{labelling } (\text{black } prb))) \rangle$ 
        c)
           $\langle \text{sat } c \rangle$ 
          sat-sub-by-sat
          by fast
        qed

      thus ?thesis
        using mark-step(3)  $\langle \text{subpath } (red \text{ } prb) \text{ } rv1 \text{ } res \text{ } rv3 \text{ } (subs \text{ } prb) \rangle$ 
          lst-of-sp-is-vert[of red prb rv1 res rv3 subs prb]
          sat-not-marked[OF RedBlack.mark-step[OF mark-step(1,3)]]
        by auto
      qed
    qed

next

  show  $\text{bes} = \text{ui-es } res$  by (rule  $\langle \text{bes} = \text{ui-es } res \rangle$ )
  qed

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
  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)))
      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 ¬ (∃ res21 bes22. bes2 = ui-es res21 @ bes22
        ∧ res21 ≠ []
        ∧ 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 ≠ []
  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 ⟨bes2 = ui-es res21
@ bes22⟩)
  next
    show res21 ≠ [] by (rule ⟨res21 ≠ []⟩)
  next
    show subpath-from (red prb) rv3 res21 (subs prb)
      using mark-step(3)
      ⟨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
  qed
  qed
  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 $bes \in feasible\ subpaths\ from\ (black\ prb')\ (confs\ prb'\ rv)\ (fst\ rv)$

hence $bes \in RedBlack\ subpaths\ from\ prb\ rv$
using *subsum-step(2)[of rv] subsum-step(3-7)* **by** *auto*

thus $bes \in RedBlack\ subpaths\ from\ prb'\ rv$
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 $\neg marked\ prb\ rv'$
thus $bes \in RedBlack\ subpaths\ from\ prb'\ rv$
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 $B : rv' \in fringe\ prb$
and $C : subpath\ (red\ prb)\ rv\ res1\ rv'\ (subs\ prb)$

and $E : \neg (\exists res21\ bes22. bes2 = ui\ es\ res21\ @\ bes22$
 $\wedge res21 \neq []$
 $\wedge subpath\ from\ (red\ prb)\ rv'\ res21\ (subs\ prb))$

and $F : Graph.subpath\ (black\ prb)\ (fst\ rv')\ bes2\ bl$
show $bes \in RedBlack\ subpaths\ from\ prb'\ rv$
proof (*case-tac rv' = subsumee sub*)

```

assume  $rv' = \text{subsumee } sub$ 

show  $?thesis$ 
  using  $\langle bes \in \text{feasible-subpaths-from } (black \text{ } prb') (confs \text{ } prb' \text{ } rv) (fst \text{ } rv) \rangle$ 
A C F
  proof ( $\text{induct } bes2 \text{ arbitrary} : bes \text{ } bl \text{ rule} : \text{rev-induct}, \text{ goal-cases}$ )

    case ( $1 \text{ } bes \text{ } bl$ ) thus  $?case$ 
      using  $\text{subsum-step}(3) \text{ } B \text{ } sp\text{-in-extends}[of \text{ } sub \text{ } red \text{ } prb]$ 
      by ( $\text{simp } (no\text{-asm}) \text{ only} :$ 
         $\text{RedBlack-subpaths-from-def } Un\text{-iff } image\text{-def } Bex\text{-def}$ 
mem-Collect-eq,
         $\text{intro } disjI1, \text{ rule-tac } ?x=res1 \text{ in } exI, \text{ intro } conjI$ 
        ( $\text{rule-tac } ?x=rv' \text{ in } exI, \text{ auto } simp \text{ add} : \text{fringe-def}$ )

    next

      case ( $2 \text{ } be \text{ } bes2 \text{ } bes \text{ } bl$ )
      then obtain  $c1 \text{ } c2 \text{ } c3$ 
        where  $se\text{-star } (confs \text{ } prb' \text{ } rv) (\text{trace } (ui\text{-es } res1) (\text{labelling } (black \text{ } prb)))$ 
c1
          and  $se\text{-star } c1 (\text{trace } bes2 (\text{labelling } (black \text{ } prb))) c2$ 
          and  $se \text{ } c2 (\text{labelling } (black \text{ } prb) \text{ } be) c3$ 
          and  $sat \text{ } c3$ 
          using  $\text{subsum-step}(3)$ 
          by ( $\text{simp } add : \text{feasible-def } se\text{-star-append } se\text{-star-append-one } se\text{-star-one}$ )
blast

      have  $ui\text{-es } res1 @ bes2 \in \text{RedBlack-subpaths-from } prb' \text{ } rv$ 
      proof –
        have  $ui\text{-es } res1 @ bes2 \in \text{feasible-subpaths-from } (black \text{ } prb') (confs \text{ } prb' \text{ } rv) (fst \text{ } rv)$ 
      proof –

        have  $\text{Graph.subpath-from } (black \text{ } prb') (fst \text{ } rv) (ui\text{-es } res1 @ bes2)$ 
        using  $\text{subsum-step } 2(5) \text{ } red\text{-sp-imp-black-sp}[OF \text{ } subsum\text{-step}(1) \text{ } C]$ 
        by ( $\text{simp } add : \text{Graph.sp-append}$ ) blast

        moreover
        have  $\text{feasible } (confs \text{ } prb' \text{ } rv)$ 
          ( $\text{trace } (ui\text{-es } res1 @ bes2) (\text{labelling } (black \text{ } prb'))$ )
        proof –
          have  $se\text{-star } (confs \text{ } prb' \text{ } rv)$ 
            ( $\text{trace } (ui\text{-es } res1 @ bes2) (\text{labelling } (black \text{ } prb'))$ )
            c2

```

```

    using subsum-step
      ‹se-star (confs prb' rv) (trace (ui-es res1)
        (labelling (black prb))) (c1)›
      ‹se-star c1 (trace bes2 (labelling (black prb))) c2›
  by (simp add : se-star-append) blast

  moreover
  have sat c2
    using ‹se c2 (labelling (black prb) be) c3› ‹sat c3›
    by (simp add : se-sat-imp-sat)

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

  ultimately
  show ?thesis by simp
qed

  moreover
  have Graph.subpath-from (black prb) (fst rv') bes2
    using 2(5) by (auto simp add : Graph.sp-append-one)

  ultimately
  show ?thesis using 2(1,4) by (auto simp add : Graph.sp-append-one)
qed

  thus ?case
  apply (subst (asm) RedBlack-subpaths-from-def)
  unfolding Un-iff image-def Bex-def mem-Collect-eq
  proof (elim disjE exE conjE, goal-cases)

    case (1 res rv'')
    show ?thesis
    proof (case-tac be ∈ ui-edge ' out-edges (red prb') rv'')

      assume be ∈ ui-edge ' out-edges (red prb') rv''
      then obtain re where be = ui-edge re
        and re ∈ 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 disjI1)

```

```

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 1(2) ⟨re ∈ out-edges (red prb') rv'⟩
    by (simp add : sp-append-one)
next
show ¬ marked prb' (tgt re)
proof –
  have sat (confs prb' (tgt re))
proof –
  have subpath (red prb') rv (res@[re]) (tgt re) (subs prb')
    using 1(2) ⟨re ∈ out-edges (red prb') rv'⟩
    by (simp add : sp-append-one)

then obtain c
where 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 1(1)
  ⟨se-star (confs prb' rv) (trace (ui-es res1)
    (labelling (black prb))) (c1)⟩
  ⟨se-star c1 (trace bes2 (labelling (black prb))) c2⟩
  ⟨se c2 (labelling (black prb) be) c3⟩
  ⟨sat c3⟩ ⟨be = ui-edge re⟩
  se-star-succs-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)
  using subsum-step(3,5,6,7)
  ⟨subpath (red prb') rv (res@[re]) (tgt re) (subs
prb')⟩
  ⟨se-star (confs prb' rv)(trace (ui-es (res@[re]))

```

```

      (labelling (black prb)))(c)
      finite-RedBlack.SE-rel[of prb] RB'
    by (simp add : finite-RedBlack-def)

  ultimately
  show ?thesis by (simp add: sat-sub-by-sat)
qed

  thus ?thesis
    using ⟨re ∈ out-edges (red prb') rv'⟩
      sat-not-marked[OF RB', of tgt re]
    by (auto simp add : vertices-def)
  qed
next
  show bes = ui-es (res@[re]) using 1(1) 2(3) ⟨be = ui-edge
re⟩ by simp
  qed

next

  assume be ∉ ui-edge ' out-edges (red prb') rv''

  show ?thesis
  proof (case-tac rv'' ∈ subsumeas (subs prb'))

    assume rv'' ∈ subsumeas (subs prb')

    then obtain arv'' where (rv'',arv'') ∈ (subs prb') by auto

    hence subpath (red prb') rv res arv'' (subs prb')
      using ⟨subpath (red prb') rv res rv'' (subs prb')⟩
      by (simp add : sp-append-sub)

    show ?thesis
    proof (case-tac be ∈ ui-edge ' out-edges (red prb') arv'')

      assume be ∈ ui-edge ' out-edges (red prb') arv''

      then obtain re where re ∈ out-edges (red prb') arv''
        and be = ui-edge re
        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=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 1(1)
    ⟨se-star (confs prb' rv) (trace (ui-es res1)
      (labelling (black prb))) (c1)⟩
    ⟨se-star c1 (trace bes2 (labelling (black prb))) c2⟩
    ⟨se c2 (labelling (black prb) be) c3⟩ ⟨sat c3⟩
    ⟨be = ui-edge re⟩
    se-star-succs-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)

```

```

    using subsum-step(3,5,6,7) se RB'
      finite-RedBlack.SE-rel[of prb']
      ⟨subpath (red prb') rv (res@[re]) (tgt re) (subs
prb')⟩
    by (simp add : finite-RedBlack-def)

  ultimately
  show ?thesis by (simp add: sat-sub-by-sat)
qed

  thus ¬ marked prb' (tgt re)
  using ⟨re ∈ out-edges (red prb') arv''⟩
    sat-not-marked[OF RB', of tgt re]
  by (auto simp add : vertices-def)

next

  show bes = ui-es (res @ [re])
  using ⟨bes = ui-es res1 @ bes2 @ [be]⟩
    ⟨ui-es res1 @ bes2 = ui-es res⟩
    ⟨be = ui-edge re⟩
  by simp

qed

next

  assume A : be ∉ ui-edge ' out-edges (red prb') arv''

  have src be = fst arv''
  proof -
    have Graph.subpath (black prb') (fst rv) (ui-es res1 @ bes2)
(fst arv'')
  using ⟨ui-es res1 @ bes2 = ui-es res⟩
    ⟨subpath (red prb') rv res arv'' (subs prb')⟩
    red-sp-imp-black-sp[OF RB']
  by auto

  moreover
  have Graph.subpath (black prb') (fst rv) (ui-es res1 @ bes2)
(src be)
  using ⟨bes ∈ feasible-subpaths-from (black prb') (confs prb' rv)
(fst rv)⟩
    ⟨bes = ui-es res1 @ bes2 @ [be]⟩
  by (auto simp add : Graph.sp-append Graph.sp-append-one

```

Graph.sp-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]⟩
  ⟨ui-es res1 @ bes2 = ui-es res⟩
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)
  ⟨Graph.subpath (black prb) (fst rv') (bes2 @ [be]) bl⟩
```



```

    by (simp add : Graph.sp-append-one)

    thus ui-edge 'out-edges (red prb') arv'' ⊂ out-edges (black
prb')
                                     (fst arv'')
    using ⟨src be = fst arv''⟩ A red-OA-subset-black-OA[OF
RB', of arv'']
    by auto
qed

next

show subpath (red prb') rv res arv'' (subs prb')
by (rule ⟨subpath (red prb') rv res arv'' (subs prb')⟩)

next

show ¬ (∃ res21 bes22. [be] = ui-es res21 @ bes22
    ∧ res21 ≠ []
    ∧ subpath-from (red prb') arv'' res21 (subs
prb'))

proof (intro notI, elim exE conjE, goal-cases)
  case (1 res21 bes22 rv'')

  have be ∈ ui-edge 'out-edges (red prb') arv''
  proof -
    obtain re res21' where res21 = re # res21'
    using 1(2) unfolding neq-Nil-conv by blast

    have be = ui-edge re and re ∈ out-edges (red prb') arv''
    proof -
      show be = ui-edge re using 1(1) ⟨res21 = re # res21'⟩
    by simp

    next
    have re ∈ edges (red prb')
      using 1(3) ⟨res21 = re # res21'⟩ by (simp add :
sp-Cons)

    moreover
    have src re = arv''
    proof -
      have (arv'',src re) ∉ 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)

    thus ?thesis
      using 1(3) ⟨res21 = re # res21'⟩
      by (simp add : rb-sp-Cons[OF RB'])
  qed

  ultimately
  show re ∈ 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)
    ⟨Graph.subpath (black prb) (fst rv') (bes2 @ [be]) bl⟩
    ⟨(rv'', arv'') ∈ subs prb'⟩
    ⟨subpath (red prb') rv res arv'' (subs prb')⟩
    ⟨src be = fst arv''⟩
    RB' red-sp-imp-black-sp subs-to-same-BL
  by (simp add : Graph.sp-append-one Graph.sp-one)
  qed
  qed
  qed

next

  assume rv'' ∉ subsumeas (subs prb')

  show ?thesis
  proof (case-tac be ∈ ui-edge ' out-edges (red prb') rv'')

    assume be ∈ ui-edge ' out-edges (red prb') rv''

    then obtain re where be = ui-edge re
      and re ∈ 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 disjI1)
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')⟩
           ⟨re ∈ out-edges (red prb') rv''⟩
    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')⟩
               ⟨re ∈ out-edges (red prb') rv''⟩
        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 1(1)
    ⟨se-star (confs prb' rv) (trace (ui-es res1)
           (labelling (black prb)))(c1)⟩
    ⟨se-star c1 (trace bes2 (labelling (black prb)))(c2)⟩
    ⟨se c2 (labelling (black prb) be) c3⟩ ⟨sat c3⟩
    ⟨be = ui-edge re⟩
    se-star-succs-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  $\sqsubseteq$  confs prb' (tgt re)
      using subsum-step(3,5,6,7) se RB'
        finite-RedBlack.SE-rel[of prb']
         $\langle$ subpath (red prb') rv (res@[re]) (tgt re) (subs
prb') $\rangle$ 
        by (simp add : finite-RedBlack-def)

    ultimately
    show ?thesis by (simp add: sat-sub-by-sat)
qed

    thus ?thesis
      using  $\langle$ re  $\in$  out-edges (red prb') rv'' $\rangle$ 
        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') rv''

show ?thesis
  unfolding RedBlack-subpaths-from-def Un-iff Bex-def
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  $\langle$ 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 rv''
    proof -

```

```

be)      have Graph.subpath (black prb') (fst rv) (ui-es res) (src
        using ⟨bes ∈ feasible-subpaths-from (black prb')
              (confs prb' rv) (fst rv)⟩
        ⟨bes = ui-es res1 @ bes2 @ [be]⟩
        ⟨ui-es res1 @ bes2 = ui-es res⟩
        red-sp-imp-black-sp
prb')⟩]  [OF RB' ⟨subpath (red prb') rv res rv'' (subs
        by (subst (asm)(2) eq-commute)
            (auto simp add : Graph.sp-append Graph.sp-one)

        thus ?thesis
        using red-sp-imp-black-sp
            [OF RB' ⟨subpath (red prb') rv res rv'' (subs prb')⟩]
        by (rule sp-same-src-imp-same-tgt)
qed

show ?case
apply (rule-tac ?x=rv'' in exI)
proof (intro conjI)

        show rv'' ∈ fringe prb'
        unfolding fringe-def mem-Collect-eq
        proof (intro conjI)
          show rv'' ∈ red-vertices prb'
            using ⟨subpath (red prb') rv res rv'' (subs prb')⟩
            by (simp add : lst-of-sp-is-vert)
        next
          show rv'' ∉ subsumees (subs prb')
            by (rule ⟨rv'' ∉ subsumees (subs prb')⟩)
        next
          show ¬ marked prb' rv'' by (rule ⟨¬ marked prb' rv''⟩)
        next
          have be ∈ edges (black prb')
            using subsum-step(3)
            ⟨Graph.subpath (black prb) (fst rv') (bes2 @
[be]) bl⟩
            by (simp add : Graph.sp-append-one)

        thus ui-edge ' out-edges (red prb') rv'' ⊂
              out-edges (black prb') (fst rv'')
        using ⟨src be = fst rv''⟩ A
              red-OA-subset-black-OA[OF RB', of rv'']
        by auto

```

qed

next

show $\text{subpath } (red \text{ } prb') \text{ } rv \text{ } res \text{ } rv'' \text{ } (subs \text{ } prb')$
by (rule $\langle \text{subpath } (red \text{ } prb') \text{ } rv \text{ } res \text{ } rv'' \text{ } (subs \text{ } prb') \rangle$)

next

show $\neg (\exists res21 \text{ } bes22. [be] = \text{ui-es } res21 \text{ } @ \text{ } bes22$
 $\wedge res21 \neq []$
 $\wedge \text{SubRel.subpath-from } (red \text{ } prb') \text{ } (rv''$
 $\text{ } (res21) \text{ } (subs \text{ } prb'))$

proof (intro notI, elim exE conjE, goal-cases)
case (1 res21 bes22 rv''')

have $be \in \text{ui-edge } \langle \text{out-edges } (red \text{ } prb') \text{ } rv''$

proof –

obtain $re \text{ } res21'$ where $res21 = re \# res21'$

using 1(2) unfolding *neq-Nil-conv* by blast

have $be = \text{ui-edge } re$

and $re \in \text{out-edges } (red \text{ } prb') \text{ } rv''$

proof –

show $be = \text{ui-edge } re$ using 1(1) $\langle res21 = re \# res21' \rangle$

by *simp*

next

have $re \in \text{edges } (red \text{ } prb')$

using 1(3) $\langle res21 = re \# res21' \rangle$ by (*simp add* :

sp-Cons)

moreover

have $\text{src } re = rv''$

proof –

have $(rv'', \text{src } re) \notin \text{subs } prb'$

using $\langle rv'' \notin \text{subsumees } (subs \text{ } prb') \rangle$ by force

thus ?thesis

using 1(3) $\langle res21 = re \# res21' \rangle$

by (*simp add* : *rb-sp-Cons[OF RB']*)

qed

ultimately

show $re \in \text{out-edges } (red \text{ } prb') \text{ } rv''$ by *simp*

qed

```

    thus ?thesis by auto
  qed

  thus False using A by (elim notE)
  qed

next

  show Graph.subpath-from (black prb') (fst rv'') [be]
    using subsum-step(3)
    ⟨Graph.subpath (black prb) (fst rv') (bes2 @ [be])
      ⟨src be = fst rv''⟩
    by (rule-tac ?x=tgt be in exI)
      (simp add : Graph.sp-append-one Graph.sp-one)

  qed
  qed
  qed
  qed
  qed

next

  case (2 res1' bes2' rv'' bl')

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

    assume bes2' = []

    have Graph.subpath (black prb') (fst rv) (ui-es res1' @ [be]) bl
    proof –
      have Graph.subpath (black prb') (fst rv) (ui-es res1') (src be)
      proof –
        have Graph.subpath (black prb') (fst rv') bes2 (src be)
        using subsum-step(3)
        ⟨Graph.subpath (black prb) (fst rv') (bes2@[be]) bl⟩
        by (simp add : Graph.sp-append-one)

      moreover
      have subpath (red prb') rv res1 rv' (subs prb')
      using subsum-step(3) ⟨subpath (red prb) rv res1 rv' (subs
prb)⟩

```

```

    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'⟩ ⟨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) ⟨Graph.subpath (black prb) (fst rv)
(bes2@[be]) bl⟩
  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 ∈ 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 ∈ ui-edge 'out-edges (red prb') rv'')

  assume be ∈ ui-edge 'out-edges (red prb') rv''

  then obtain re where be = ui-edge re

```



```

      and re ∈ 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 disjI1)
  apply (rule-tac ?x=res1'@[re] in exI)
  apply (intro conjI)
  apply (rule-tac ?x=tgt re in exI)
  proof (intro conjI)
    show subpath (red prb') rv (res1' @ [re]) (tgt re) (subs prb')
      using ⟨subpath (red prb') rv res1' rv'' (subs prb')⟩
        ⟨re ∈ out-edges (red prb') rv''⟩
      by (simp add : sp-append-one)
  next
    show ¬ marked prb' (tgt re)
  proof -
    have sat (confs prb' (tgt re))
    proof -
      have subpath (red prb') rv (res1'@[re]) (tgt re) (subs
prb')

      using ⟨subpath (red prb') rv res1' rv'' (subs prb')⟩
        ⟨re ∈ out-edges (red prb') rv''⟩
      by (simp add : sp-append-one)

    then obtain c
      where se : se-star (confs prb' rv) (trace (ui-es
(res1'@[re]))

        (labelling (black prb))) (c)
      using subsum-step(3,5,6,7) RB'
        finite-RedBlack.sp-imp-ex-se-star-succ
          [of prb' rv res1'@[re] tgt re]
      unfolding finite-RedBlack-def
      by simp blast

    hence sat c
  proof -
    have bes = ui-es (res1'@[re])
      using ⟨bes = ui-es res1 @ bes2 @ [be]⟩
        ⟨be = ui-edge re⟩ ⟨bes2' = []⟩
        ⟨ui-es res1 @ bes2 = ui-es res1' @
bes2'⟩

      by simp

```

```

    thus ?thesis
    using subsum-step(3) se-star-succs-states[OF
se]
    ‹bes ∈ feasible-subpaths-from (black
prb')
    (confs prb' rv)
    (fst rv)›
    by (auto simp add : feasible-def sat-eq)
qed

moreover
have c ⊆ confs prb' (tgt re)
using subsum-step(3,5,6,7) se
    finite-RedBlack.SE-rel[of prb'] RB'
    ‹subpath (red prb') (rv) (res1'@[re])
    (tgt re) (subs prb')›
by (simp add : finite-RedBlack-def)

ultimately
show ?thesis by (simp add: sat-sub-by-sat)
qed

thus ?thesis
using ‹re ∈ out-edges (red prb') rv''›
    sat-not-marked[OF RB', of tgt re]
by (auto simp add : vertices-def)
qed
next
show bes = ui-es (res1' @ [re])
using ‹bes = ui-es res1 @ bes2 @ [be]›
    ‹ui-es res1 @ bes2 = ui-es res1' @ bes2'›
    ‹bes2' = []› ‹be = ui-edge re›
by simp
qed

next

assume A : be ∉ ui-edge ‹out-edges (red prb') rv''›
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=[be] in exI)
proof (intro conjI, goal-cases)

```

show $bes = ui-es\ res1' @ [be]$
using $\langle bes = ui-es\ res1 @ bes2 @ [be] \rangle$
 $\langle ui-es\ res1 @ bes2 = ui-es\ res1' @ bes2' \rangle$
 $\langle bes2' = [] \rangle$
by *simp*

next

case 2 show *?case*
apply (*rule-tac ?x=rv'' in exI*)
proof (*intro conjI*)

show $rv'' \in fringe\ prb'$ **by** (*rule* $\langle rv'' \in fringe\ prb' \rangle$)

next

show *subpath (red prb') rv res1' rv'' (subs prb')*
by (*rule* $\langle subpath\ (red\ prb')\ rv\ res1'\ rv''\ (subs\ prb') \rangle$)

next

show $\neg (\exists res21\ bes22. [be] = ui-es\ res21 @ bes22$
 $\wedge res21 \neq []$
 $\wedge subpath-from\ (red\ prb')\ (rv'')$
 $(res21)\ (subs\ prb'))$

proof (*intro notI, elim exE conjE, goal-cases*)
case (*1 res21 bes22 rv''*)

then obtain $re\ res21'$ **where** $be = ui-edge\ re$
and $res21 = re \# res21'$
unfolding *neq-Nil-conv* **by** *auto*

moreover

hence $re \in out-edges\ (red\ prb')\ rv''$

using *1(3) <rv'' ∈ fringe prb'> RB'*

unfolding *subsumees-conv* **by** (*force simp add :*

fringe-def

rb-sp-Cons)

ultimately

show *False* **using** *A* **by** *auto*

qed

```

next

  show Graph.subpath-from (black prb') (fst rv'') [be]
    using  $\langle \text{Graph.subpath } (\text{black } prb') (\text{fst } rv) (\text{ui-es}$ 
res1'@[be]) bl \rangle
     $\langle \text{fst } rv'' = \text{src } be \rangle$ 
    by (auto simp add : Graph.sp-append-one Graph.sp-one)

  qed
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  $\langle \text{bes} = \text{ui-es } res1 \ @ \ bes2 \ @ \ [be] \rangle$ 
     $\langle \text{ui-es } res1 \ @ \ bes2 = \text{ui-es } res1' \ @ \ bes2' \rangle$ 
    by simp

  next

  case 2 show ?case
    apply (rule-tac ?x=rv'' in exI)
    proof (intro conjI)

      show rv'' ∈ fringe prb' by (rule  $\langle \text{rv''} \in \text{fringe } prb' \rangle$ )

    next

    show subpath (red prb') rv res1' rv'' (subs prb')
    by (rule  $\langle \text{subpath } (\text{red } prb') \text{ rv } res1' \text{ rv'' } (\text{subs } prb') \rangle$ )

```

```

next

  show  $\neg (\exists res21\ bes22. bes2' @ [be] = ui-es\ res21 @$ 
bes22
       $\wedge res21 \neq []$ 
       $\wedge subpath-from\ (red\ prb')\ (rv'')$ 
       $(res21)\ (subs\ prb')$ )
proof (intro notI, elim exE conjE, goal-cases)
  case (1 res21 bes22 rv''')

  then obtain re res21' where res21 = re # res21'
    and be' = ui-edge re
  using <bes2' = be' # bes2''> unfolding neq-Nil-conv

by auto

  show False
bes22
  using < $\neg (\exists res21\ bes22. bes2' = ui-es\ res21 @$ 
       $\wedge res21 \neq []$ 
       $\wedge subpath-from\ (red\ prb')\ (rv'')$ 
       $(res21)\ (subs\ prb'))$ >
  apply (elim notE)
  apply (rule-tac ?x=[re] in exI)
  apply (rule-tac ?x=bes2'' in exI)
  proof (intro conjI)
    show bes2' = ui-es [re] @ bes2''
      using <bes2' @ [be] = ui-es res21 @ bes22>
      <bes2' = be' # bes2''>
      <be' = ui-edge re>
    by simp
  next
  show [re]  $\neq []$  by simp
  next
  show subpath-from (red prb') rv'' [re] (subs prb')
    using <subpath (red prb') rv'' res21 rv'''(subs
prb')>
    <res21 = re # res21'>
    by (fastforce simp add : sp-Cons Nil-sp
vertices-def)

qed

next

show Graph.subpath-from (black prb') (fst rv'') (bes2' @

```

[be])

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 ∈ feasible-subpaths-from (black prb')
(confs prb' rv)
(fst rv)⟩

⟨bes = ui-es res1 @ bes2 @ [be]⟩

by (auto simp add : *Graph.sp-append Graph.sp-one*)

thus ?thesis **using** ⟨ui-es res1 @ bes2 = ui-es

res1'@bes2'⟩

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])

bl⟩

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
    qed
  qed

next

  assume  $rv' \neq \text{subsumee } sub$ 

  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  $bes = \text{ui-es } res1 @ bes2$  by (rule <bes = ui-es res1 @ bes2>)
    next

      case 2 show ?case
        apply (rule-tac ?x=rv' in exI)
        proof (intro conjI)
          show  $rv' \in \text{fringe } prb'$ 
          using subsum-step(3) subsumE-fringe[OF subsum-step(3)] B <rv'  $\neq$ 
subsumee sub>
            by simp
        next
          show subpath (red prb') rv res1 rv' (subs prb')
          using subsum-step(3) C by (auto simp add : sp-in-extends)

        next
          show  $\neg (\exists res21 \ bes22. bes2 = \text{ui-es } res21 @ bes22$ 
             $\wedge res21 \neq []$ 
             $\wedge \text{subpath-from (red prb') } rv' res21 (subs prb'))$ 
          proof (intro notI, elim exE conjE)
            fix res21 bes22 rv''

            assume  $bes2 = \text{ui-es } res21 @ bes22$ 
            and  $res21 \neq []$ 
            and subpath (red prb') rv' res21 rv'' (subs prb')

            then obtain re res21' where  $res21 = re \# res21'$ 
            unfolding neq-Nil-conv by blast

            have subpath (red prb) rv' [re] (tgt re) (subs prb)
            proof -
              have  $\neg \text{uses-sub } rv' [re] (tgt re) sub$  using <rv'  $\neq$  subsumee sub>

```

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 ?x=[re] in exI)
apply (rule-tac ?x=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)
apply (rule-tac ?x=tgt re in exI)
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
qed

```



```

qed
qed

next

case (abstract-step prb rv2 ca prb' rv1)
have RB' : RedBlack prb' by (rule RedBlack.abstract-step[OF abstract-step(1,3)])
have finite-RedBlack prb using abstract-step by (auto simp add : finite-RedBlack-def)
show ?case
unfolding subset-iff
proof (intro allI impI)

  fix bes

  assume bes ∈ feasible-subpaths-from (black prb') (confs prb' rv1) (fst rv1)

  show bes ∈ RedBlack-subpaths-from prb' rv1
  proof (case-tac rv2 = rv1)

    assume rv2 = rv1

    show ?thesis
    proof (case-tac out-edges (black prb') (fst rv1) = {})
      assume out-edges (black prb') (fst rv1) = {}
      show ?thesis
      unfolding RedBlack-subpaths-from-def Un-iff image-def Bex-def
mem-Collect-eq
      apply (intro disjI1)
      apply (rule-tac ?x=[] in exI)
      apply (intro conjI)
      apply (rule-tac ?x=rv1 in exI)
      proof (intro conjI)
        show subpath (red prb') rv1 [] rv1 (subs prb')
        using abstract-step(4) rb-Nil-sp[OF RB'] by fast
      next
        show ¬ marked prb' rv1 using abstract-step(3) ⟨rv2 = rv1⟩
    by simp
      next
        show bes = ui-es []
      using ⟨bes ∈ feasible-subpaths-from (black prb') (confs prb' rv1)
(fst rv1)⟩
        ⟨out-edges (black prb') (fst rv1) = {}⟩
      by (cases bes) (auto simp add : Graph.sp-Cons)
    qed
  qed

```

```

next
  assume out-edges (black prb') (fst rv1) ≠ {}

  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=bes in exI)
  proof (intro conjI, goal-cases)

    show bes = ui-es [] @ bes by simp

next

  case 2 show ?case
  apply (rule-tac ?x=rv1 in exI)
  proof (intro conjI)

    show rv1 ∈ fringe prb'
    using abstract-step(1,3) ⟨rv2 = rv1⟩ ⟨out-edges (black prb') (fst
rv1) ≠ {}⟩
    by (auto simp add : fringe-def)

next

  show subpath (red prb') rv1 [] rv1 (subs prb')
  using abstract-step(3) ⟨rv2 = rv1⟩
  rb-Nil-sp[OF RedBlack.abstract-step[OF abstract-step(1,3)]]
  by auto

next

  show  $\neg (\exists res21 bes22. bes = ui-es res21 @ bes22$ 
   $\wedge res21 \neq []$ 
   $\wedge subpath-from (red prb') rv1 res21 (subs prb'))$ 
  proof (intro notI, elim exE conjE)
  fix res21 rv3

  assume res21 ≠ []
  and subpath (red prb') rv1 res21 rv3 (subs prb')

  moreover
  then obtain re res21' where res21 = re # 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)
(fst rv1)⟩
    by simp

    qed
    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 bes = ui-es res
and subpath (red prb) rv1 res rv3 (subs prb)
and ¬ marked prb rv3

thus ?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=res in exI, intro conjI)
(rule-tac ?x=rv3 in exI, simp-all)
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 abstract-step(3) B C E F unfolding fringe-def
by (rule-tac ?x=rv3 in exI) auto
qed
qed
qed
next

case (strengthen-step prb rv2 e prb' rv1)
show ?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)

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

```

qed

qed

qed

qed

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)))* \subseteq *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* \sqsubseteq *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* :

```

assumes RedBlack prb
shows feasible-paths (black prb) (init-conf prb)  $\subseteq$  RedBlack-paths prb
unfolding subset-iff mem-Collect-eq
proof (intro allI impI, elim exE conjE, goal-cases)
  case (1 es bl)

  hence es  $\in$  feasible-subpaths-from (black prb) (init-conf prb) (fst (root (red prb)))
    using consistent-roots[OF assms] by simp blast

  hence es  $\in$  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  $\forall e \in \text{pred (confs prb (root (red prb)))}$ . finite (Bexp.vars e)
            using finite-RedBlack finite-pred-constr-symvars[OF assms]
            by (fastforce simp add : finite-RedBlack-def vertices-def)

        moreover
          have  $\forall e \in \text{pred (init-conf prb)}$ . finite (Bexp.vars e)
            using assms by (intro finite-init-pred-symvars)

        moreover
          have init-conf prb  $\subseteq$  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 subsump-

tions. It would be possible to generate the core operations on red-black graphs by the Isabelle code-generator, by introducing un-interpreted function symbols for concrete heuristic functions mapped to implementations written by hand. This represents a substantial albeit rewarding effort that has not yet been undertaken.

References

- [1] J. Jaffar, V. Murali, J. A. Navas, and A. E. Santosa, “TRACER: A Symbolic Execution Tool for Verification,” in *Proceedings of CAV '12*, 2012, pp. 758–766.
- [2] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, “The Software Model Checker Blast,” *STTT*, vol. 9, no. 5-6, pp. 505–525, 2007.
- [3] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav, “SATABS: SAT-Based Predicate Abstraction for ANSI-C,” in *Proceedings of TACAS '05*, 2005, pp. 570–574.
- [4] F. Ivancic, Z. Yang, M. K. Ganai, A. Gupta, I. Shlyakhter, and P. Ashar, “F-Soft: Software Verification Platform,” in *Proceedings of CAV '05*, 2005, pp. 301–306.
- [5] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko, “Synthesizing Software Verifiers from Proof Rules,” in *Proceedings of PLDI '12*, 2012, pp. 405–416.
- [6] K. L. McMillan, *Proceedings of CAV '06*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, ch. Lazy Abstraction with Interpolants, pp. 123–136.
- [7] R. Aissat, F. Voisin, and B. Wolff, “Infeasible paths elimination by symbolic execution techniques: Proof of correctness and preservation of paths,” in *ITP'16*, ser. LNCS, vol. 9807, 2016.
- [8] R. Aissat, M.-C. Gaudel, F. Voisin, and B. Wolff, “Pruning infeasible paths via graph transformations and symbolic execution: a method and a tool,” L.R.I., Univ. Paris-Sud, Tech. Rep. 1588, 2016. [Online]. Available: <https://www.lri.fr/srubrique.php?news=33>
- [9] A. Denise, M.-C. Gaudel, S.-D. Gouraud, R. Lassaigne, J. Oudinet, and S. Peyronnet, “Coverage-biased Random Exploration of large Models

and Application to Testing,” *International Journal on Software Tools for Technology Transfer*, vol. 14, no. 1, pp. 73–93, 2011.

- [10] L. Noschinski, “A Graph Library for Isabelle,” *Mathematics in Computer Science*, vol. 9, no. 1, pp. 23–39, 2015. [Online]. Available: <https://doi.org/10.1007/s11786-014-0183-z>
- [11] D. v. Oheimb and T. Nipkow, “Hoare logic for nanojava: Auxiliary variables, side effects, and virtual methods revisited,” ser. LNCS. Springer-Verlag, 2002, pp. 89–105. [Online]. Available: <http://dl.acm.org/citation.cfm?id=647541.730154>
- [12] A. Lochbihler, “Java and the java memory model - A unified, machine-checked formalisation,” in *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, ser. LNCS. Springer-Verlag, 2012, pp. 497–517. [Online]. Available: https://doi.org/10.1007/978-3-642-28869-2_25
- [13] T. Nipkow, “Winskel is (almost) right: Towards a mechanized semantics,” *Formal Asp. Comput.*, vol. 10, no. 2, pp. 171–186, 1998. [Online]. Available: <https://doi.org/10.1007/s001650050009>
- [14] A. D. Brucker and B. Wolff, “An Extensible Encoding of Object-oriented Data Models in HOL with an Application to IMP++,” *Journal of Automated Reasoning (JAR)*, vol. Selected Papers of the AVOCS-VERIFY Workshop 2006, no. 3–4, pp. 219–249, 2008, serge Autexier, Heiko Mantel, Stephan Merz, and Tobias Nipkow (eds).