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

Romain Aissat and Frédéric Voisin and Burkhart Wolff<br>LRI, Univ Paris-Sud, CNRS, CentraleSupélec, Université Paris-Saclay, France aissat@lri.fr, wolff@lri.fr

September 13, 2023


#### Abstract

TRACER [1] is a tool for verifying safety properties of sequential C programs. TRACER attempts at building a finite symbolic execution graph which over-approximates the set of all concrete reachable states and the set of feasible paths.

We present an abstract framework for TRACER and similar CE-GAR-like systems $[2,3,4,5,6]$. The framework provides 1) a graphtransformation based method for reducing the feasible paths in controlflow graphs, 2) a model for symbolic execution, subsumption, predicate abstraction and invariant generation. In this framework we formally prove two key properties: correct construction of the symbolic states and preservation of feasible paths. The framework focuses on core operations, leaving to concrete prototypes to "fit in" heuristics for combining them.

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


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

## Contents

1 Introduction ..... 5
2 Rooted Graphs ..... 9
2.1 Basic Definitions and Properties ..... 9
2.1.1 Edges ..... 9
2.1.2 Rooted graphs ..... 9
2.1.3 Vertices ..... 9
2.1.4 Basic properties of rooted graphs ..... 10
2.1.5 Out-going edges ..... 10
2.2 Consistent Edge Sequences, Sub-paths and Paths ..... 11
2.2.1 Consistency of a sequence of edges ..... 11
2.2.2 Sub-paths and paths ..... 11
2.3 Adding Edges ..... 14
2.4 Trees ..... 15
3 Arithmetic Expressions ..... 15
3.1 Variables and their domain ..... 15
3.2 Program and symbolic states ..... 16
3.3 The aexp type-synonym ..... 16
3.4 Variables of an arithmetic expression ..... 17
3.5 Fresh variables ..... 18
4 Boolean Expressions ..... 18
4.1 Basic definitions ..... 19
4.1.1 The bexp type-synonym ..... 19
4.1.2 Satisfiability of an expression ..... 19
4.1.3 Entailment ..... 19
4.1.4 Conjunction ..... 19
4.2 Properties about the variables of an expression ..... 20
4.2.1 Variables of a conjunction ..... 20
4.2.2 Variables of an equality ..... 21
5 Labels ..... 22
6 Stores ..... 23
6.1 Basic definitions ..... 23
6.1.1 The store type-synonym ..... 23
6.1.2 Symbolic variables of a store ..... 23
6.1.3 Fresh symbolic variables ..... 24
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 subsomption 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 loosing part of information at a given point, abstractions might introduce infeasible paths into the result. A counterexample guided refinement is used to rule out such abstractions.
This method takes a CFG $G$ and a user given precondition and builds a new CFG $G^{\prime}$ that still over-approximates the set of feasible paths of $G$ but contains less infeasible paths. It proceeds basically as follows (see [8] for more details). First, it starts by building a classical symbolic execution tree (SET) of the program under analysis. As soon as a cyclic path is detected, the algorithm searches for a subsumption of the point at the end of the cycle by one of its ancestors. When doing this, the algorithm is allowed to abstract the ancestor in order to force the subsumption. When a subsumption is established, the current symbolic execution halts along that path and a subsumption link is added to the SET, turning it into a symbolic execution graph (SEG). When an occurrence of a final location of the original CFG is reached, we check if abstractions that might have been performed along the current path did not introduce certain infeasible paths in the new representation. If no refinement is needed, symbolic execution resumes at the next pending point. Otherwise, the analysis restarts at the point where the "faulty" abstraction occurred, but now this point is strengthened with a safeguard condition: future abstractions occurring at this point will have to entail the safeguard condition, preventing the faulty abstraction to occur again. These safeguard conditions could be user-provided but are typically the result of a weakest precondition calculus. When the analysis is over, the SEG is turned into a new CFG.
Our motivation is in random testing of imperative programs. There exist efficient algorithms that draw in a statistically uniform way long paths from very large graphs [9]. If the probability of drawing a feasible path from such a transformed CFG was high, this would lead to an efficient statistical structural white-box testing method. With testing in mind, a crucial property that our approach must have, besides being correct, is to preserve the set
of feasible paths of the original CFG. Our goal with this formalization is to establish correctness of the approach and the fact that it preserves the feasible paths of the original CFG, that is:

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

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

1. adding an arc to the SEG as the result of a symbolic execution step in the original CFG,
2. adding a subsumption link to the SEG,
3. abstracting a node of the SEG,
4. marking a node as unsatisfiable,
5. labelling a node with a safeguard condition.

Heuristics control, for example, the order in which these operators are applied, which of the possible abstractions is selected, etc. These heuristics cannot interfer with the correctness of the approach or the preservation of feasible paths since they simply combine the five kernel transformations. In the following, we model the different data structures that our method performs on and formalize our five operators but completely skip the heuristics aspects of the approach. Thus, our results extend to a large family of algorithms that add specific heuristics in their goal to over-approximate the set of feasible paths of a CFG.
Due to the nature of the problem, symbolic execution in presence of unbounded loops, such algorithms might not terminate. In practice, this is handled using some kind of timeout condition. When such condition triggers, the SEG is only a partial unfolding of the original CFG. Thus, the resulting CFG cannot contains all feasible paths of the original one. In this situation, the only way to preserve the set of feasible paths is to "connect" the SEG to the original CFG. The SEG is the currently known over-approximating set of prefixes of feasible paths and the original CFG represents the unknown part of the set of feasible paths.

In the following, we use an adequate data structure that we call a red-black graph. Its black part is the original CFG: it represents the unknown part of the set of feasible paths and is never modified during the analysis. The red part represents the SEG: its vertices are occurrences of the vertices of the black part. Then, we define the five operators that will modify the red part as described previously. We only consider red-black graphs built using these five operators, starting from a red-black graph whose red part is empty. Paths of such structures are called red-black paths. Such paths start in the red part and might end in the black part: they are made of a red feasible prefix and a black prefix on which nothing is known about feasibility. Finally, we prove that, given any red-black graph built using our five operators and modulo a renaming of vertices, the set of red-black paths is a subset of the set of black paths and that the set of feasible black paths is a subset of the set of red-black paths.
In the following, we proceed as follows (see Figure 1 for the detailed hierarchy). First, we formalize all the aspects related to symbolic execution, subsumption and abstraction (Aexp.thy, Bexp.thy, Store.thy, Conf.thy, Labels.thy, SymExec.thy). Then, we formalize graphs and their paths (Graph.thy). Using extensible records allows us to model Labeled Transition Systems from graphs (Lts.thy). Since we are interested in paths going through subsumption links, we also define these notions for graphs equipped with subsumption relations (SubRel.thy) and prove a number of theorems describing how the set of paths of such graphs evolve when an arc (ArcExt.thy) or a subsumption link (SubExt.thy) is added. Finally, we formalize the notion of red-black graphs and prove the two properties we are mainly interested in (RB.thy).


Figure 1: The hierarchy of theories.
theory Graph
imports Main
begin

## 2 Rooted Graphs

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

First, we will need the following simple lemma, which is not graph related, but that will prove useful when we will want to exhibit the last element of a non-empty sequence.
lemma neq-Nil-conv2 :
$x s \neq[]=\left(\exists x x s^{\prime} . x s=x s^{\prime} @[x]\right)$
by (induct xs rule : rev-induct, auto)

### 2.1 Basic Definitions and Properties

### 2.1.1 Edges

We model edges by a record 'vedge 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 ( $\left.{ }^{\prime} v,^{\prime} x\right)$ rgraph-scheme, i.e. an object that has at least all the components of a 'v rgraph. The second type parameter 'x stands for the hypothetical type parameters that such an object could have in addition of the type of vertices ' $v$. Using ( ${ }^{\prime} v,{ }^{\prime} x$ ) rgraph-scheme instead of 'v rgraph allows to reuse the following definition(s) for all type of objects that have at least the components of a rooted graph. For example, we will reuse the following definition to characterize the set of locations of a LTS (see LTS.thy).
definition vertices ::
( $v,{ }^{\prime} x$ ) rgraph-scheme $\Rightarrow$ ' $v$ set
where
vertices $g=\{$ root $g\} \cup$ src 'edges $g \cup$ tgt'edges $g$

### 2.1.4 Basic properties of rooted graphs

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

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

```
abbreviation wf-rgraph ::
    ('v,'x) rgraph-scheme }=>\mathrm{ bool
where
wf-rgraph g = root g\in src' edges g=(edges g\not={})
```

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 ::
    \(\left({ }^{\prime} v,{ }^{\prime} x\right)\) 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 $e s$ is consistent from vertex $v 1$ to another vertex $v 2$ if $v 1=v 2$ if it is empty, or, if it is not empty:

- $v 1$ is the source of its first element, and
- $v 2$ is the target of its last element, and
- the target of each of its elements is the source of its follower.
fun ces ::
$' v \Rightarrow{ }^{\prime} v$ edge list $\Rightarrow^{\prime} v \Rightarrow$ bool
where
ces $v 1[] v 2=(v 1=v 2)$
|ces v1 $(e \# e s) v 2=(s r c e=v 1 \wedge$ ces $($ tgt e) es v2 $)$


### 2.2.2 Sub-paths and paths

Let $g$ be a rooted graph, es a sequence of edges and $v 1$ and $v 2$ two vertices. $e s$ is a sub-path in $g$ from $v 1$ to $v 2$ if:

- it is consistent from $v 1$ to $v 2$,
- $v 1$ 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 $v 1$ is not one of its vertices.
definition subpath ::
( ${ }^{\prime} v,{ }^{\prime} x$ ) rgraph-scheme $\Rightarrow{ }^{\prime} v \Rightarrow{ }^{\prime} v$ edge list $\Rightarrow{ }^{\prime} v \Rightarrow$ bool
where
subpath $g$ v1 es $v 2 \equiv$ ces v1 es v2 $\wedge v 1 \in$ vertices $g \wedge$ set es $\subseteq$ edges $g$
Let $e s$ be a sub-path of $g$ leading from $v 1$ to $v 2 . v 1$ and $v 2$ are both vertices of $g$.
lemma fst-of-sp-is-vert :
assumes subpath g v1 es v2
shows $\quad v 1 \in$ vertices $g$
using assms by (simp add : subpath-def)

```
lemma lst-of-sp-is-vert :
    assumes subpath \(g\) v1 es v2
    shows \(v 2 \in\) vertices \(g\)
using assms by (induction es arbitrary : v1, auto simp add: subpath-def ver-
tices-def)
```

The empty sequence of edges is a sub-path from $v 1$ to $v 2$ 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 ::
$\left({ }^{\prime} v,{ }^{\prime} x\right)$ rgraph-scheme $\Rightarrow{ }^{\prime} v \Rightarrow{ }^{\prime} v$ edge list $\Rightarrow$ bool
where
subpath-from $g$ ves $\equiv \exists v^{\prime}$. subpath $g v$ es $v^{\prime}$
abbreviation subpaths-from ::
( $\left.v,{ }^{\prime} x\right)$ rgraph-scheme $\Rightarrow^{\prime} v \Rightarrow$ 'v edge list set
where
subpaths-from $g v \equiv\{$ es. subpath-from $g$ ves $\}$
A path is a sub-path starting at the root of the graph.
abbreviation path ::
('v,'x) rgraph-scheme $\Rightarrow$ 'v edge list $\Rightarrow^{\prime} v \Rightarrow$ bool where
path $g$ es $v \equiv$ subpath $g($ root $g)$ es $v$

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

The empty sequence is a path of any rooted graph.

```
lemma
    [] \in paths g
```

by (auto simp add : subpath-def vertices-def)
Some useful simplification lemmas for subpath.
lemma sp-one :
subpath $g$ v1 $[e] v 2=($ src $e=v 1 \wedge e \in$ edges $g \wedge$ tgt $e=v 2)$
by (auto simp add : subpath-def vertices-def)
lemma sp-Cons :
subpath $g$ v1 $(e \# e s) v 2=($ src $e=v 1 \wedge e \in$ edges $g \wedge$ subpath $g(t g t e)$ es v2 $)$
by (auto simp add : subpath-def vertices-def)
lemma sp-append-one :
subpath gv1 (es@[e]) v2 = (subpath gv1 es (srce) $\wedge e \in$ edges $g \wedge$ tgt $e=v 2)$
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 $g$ ves2 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 $g$ ves v2
shows $v 1=v 2$
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 $e \in$ set es
shows $\exists e s^{\prime} . e s=e s^{\prime} @[e] \wedge e \notin$ set $e s^{\prime}$
using assms(2,3)
proof (induction es arbitrary : v1)
case Nil thus?case by simp
next
case (Cons é es)
hence $e=e^{\prime} \vee\left(e \neq e^{\prime} \wedge e \in\right.$ 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 $\operatorname{assms}(1) \operatorname{Cons}(1)[$ 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 ::
$\left({ }^{\prime} v,{ }^{\prime} x\right)$ rgraph-scheme $\Rightarrow{ }^{\prime} v$ edge $\Rightarrow\left({ }^{\prime} v,{ }^{\prime} x\right)$ rgraph-scheme
where
add-edge $g e \equiv$ rgraph.edges-update $(\lambda$ edges. edges $\cup\{e\}) g$
Let $e s$ 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 $e s$.
lemma sp-ends-in-tgt-imp-mem :
assumes tgt $e \notin$ vertices $g$
assumes $v \neq t g t e$
assumes subpath (add-edge ge)ves (tgt e)
shows $e \in$ set es
proof -
have es $\neq[]$ using assms(2,3) by (auto simp add : subpath-def)
then obtain $e^{\prime} e s^{\prime}$ where $e s=e s^{\prime} @[e\rceil$ 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 }=>\mathrm{ bool
where
    is-tree g\equiv\foralll\inGraph.vertices g. \exists! p. Graph.path g pl
```

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 ' $v$ symvar $=' v \times$ nat
lemma
$\neg$ finite (UNIV::'v symvar set)
by (simp add : finite-prod)
The previous lemma has no name and thus cannot be referenced in the following. Indeed, it is of no use for proving the properties we are interested in. In the following, we will give other unnamed lemmas when we think they might help the reader to understand the ideas behind our modeling choices.

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

### 3.2 Program and symbolic states

A state is a total function giving values in $D$ to variables. The latter are represented by elements of type ' $v$. Unlike in the ' $v$ 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 $\left({ }^{\prime} v,{ }^{\prime} d\right)$ state $=' v \Rightarrow{ }^{\prime} d$

### 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 ( $\left.{ }^{\prime} v,{ }^{\prime} d\right)$ aexp $=\left({ }^{\prime} v,{ }^{\prime} d\right)$ state $\Rightarrow{ }^{\prime} d$
In order to represent expressions over program variables as well as symbolic variables, the type synonym $a e x p$ is parameterized by the type of variables. Arithmetic and boolean expressions over program variables are used to express program statements. Arithmetic and boolean expressions over symbolic variables are used to represent the constraints occurring in path predicates during symbolic execution.

### 3.4 Variables of an arithmetic expression

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

```
definition vars ::
    ('v,'d) aexp \(\Rightarrow\) 'v set
where
    vars \(e=\{v . \exists \sigma\) val. \(e(\sigma(v:=v a l)) \neq e \sigma\}\)
lemma vars-example-1 :
    fixes \(e::(\) ('v,integer) aexp
    assumes \(e=(\lambda \sigma . \sigma x-\sigma y)\)
    assumes \(x \neq y\)
    shows vars \(e=\{x, y\}\)
unfolding set-eq-iff
proof (intro allI iffI)
    fix \(v\) assume \(v \in\) vars \(e\)
    then obtain \(\sigma\) val
    where \(e(\sigma(v:=v a l)) \neq e \sigma\)
    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 vars e
    using assms
    by (auto simp add : vars-def)
        (rule-tac ?x=\lambda v.0 in exI, rule-tac ? }x=1\mathrm{ in exI, simp)+
qed
lemma vars-example-2 :
    fixes e::('v,integer) aexp
    assumes e=(\lambda\sigma.\sigmax-\sigma y)
    assumes }x=
    shows vars e={}
using assms by (auto simp add : vars-def)
```


### 3.5 Fresh variables

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

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

```
abbreviation fresh ::
    'v = ('v,'d) aexp = bool
where
    fresh ve\equivv\not\in 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 ( \(\left.{ }^{\prime} v, ' d\right)\) bexp \(=\left({ }^{\prime} v,{ }^{\prime} d\right)\) state \(\Rightarrow\) bool
definition vars ::
    ('v,'d) bexp \(\Rightarrow{ }^{\prime} v\) set
where
    vars \(e=\{v . \exists \sigma\) val. \(e(\sigma(v:=\) val \()) \neq e \sigma\}\)
abbreviation fresh ::
    \({ }^{\prime} v \Rightarrow\left({ }^{\prime} v,{ }^{\prime} d\right)\) bexp \(\Rightarrow\) bool
where
    fresh \(v e \equiv v \notin\) vars \(e\)
```


### 4.1.2 Satisfiability of an expression

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

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


### 4.1.3 Entailment

A boolean expression $\varphi$ entails another boolean expression $\psi$ if all states making $\varphi$ true also make $\psi$ true.
definition entails :: ( ${ }^{\prime} v,{ }^{\prime} d$ ) bexp $\Rightarrow\left(' v,{ }^{\prime} d\right)$ bexp $\Rightarrow$ bool (infixl $\left.\models{ }_{B} 55\right)$
where

$$
\varphi \not \models_{B} \psi \equiv(\forall \sigma . \varphi \sigma \longrightarrow \psi \sigma)
$$

### 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 $\sigma$ if, for all elements $e$ of $E$, $e$ associates true to $\sigma$.

```
definition conjunct ::
    (' \(\left.v,{ }^{\prime} d\right)\) bexp set \(\Rightarrow\left({ }^{\prime} v, ' d\right)\) bexp
where
    conjunct \(E \equiv(\lambda \sigma . \forall e \in E . e \sigma)\)
```


### 4.2 Properties about the variables of an expression

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

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

### 4.2.1 Variables of a conjunction

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

```
lemma vars-of-conj :
    vars \((\lambda \sigma\). e1 \(\sigma \wedge e 2 \sigma) \subseteq\) vars e1 \(\cup\) vars \(e 2\)
(is vars ?e \(\subseteq\) vars e1 \(\cup\) vars e2)
unfolding subset-iff
proof (intro allI impI)
    fix \(v\) assume \(v \in\) vars ?e
```

then obtain $\sigma$ val
where ? $e(\sigma(v:=v a l)) \neq$ ?e $\sigma$
unfolding vars-def by blast
hence $e 1(\sigma(v:=v a l)) \neq e 1 \sigma \vee e 2(\sigma(v:=v a l)) \neq e 2 \sigma$ by auto
thus $v \in$ vars e1 $\cup$ vars e2 unfolding vars-def by blast qed

```
lemma finite-conj :
    assumes finite E
    assumes }\foralle\inE\mathrm{ . finite (vars e)
    shows finite (vars (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 :
```



```
(is vars ?e \subseteqAexp.vars e1 \cup Aexp.vars e2)
unfolding subset-iff
proof (intro allI impI)
    fix v}\mathrm{ assume v}\in\mathrm{ vars ?e
    then obtain \sigma val where ?e }(\sigma(v:=val))\not=?e,
    unfolding vars-def by blast
    hence e1 (\sigma (v:=val))\not=e1 \sigma\vee e2 (\sigma (v:=val)) \not=e2 \sigma
    by auto
    thus v\in Aexp.vars e1 \cup 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.\mathrm{ e1 }\sigma=e=2 \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 $\left({ }^{\prime} v,{ }^{\prime} d\right)$ label $=$ Skip $\mid$ Assume ( $\left.{ }^{\prime} v,{ }^{\prime} d\right) \operatorname{bexp} \mid \operatorname{Assign}{ }^{\prime} v\left({ }^{\prime} v,{ }^{\prime} d\right) \operatorname{aexp}$
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)
        \(\mid\) Assign \(-e \Rightarrow\) finite (Aexp.vars e)
    |- \(\quad \Rightarrow\) True
```

abbreviation finite-labels ::
('v,'d) label list $\Rightarrow$ bool
where
finite-labels $l s \equiv(\forall l \in$ set $l s$. finite-label $l)$
end
theory Store
imports Aexp Bexp
begin

## 6 Stores

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

### 6.1 Basic definitions

### 6.1.1 The store type-synonym

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

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

We define stores as total functions from program variables to indexes.
type-synonym 'a store $=$ ' $a \Rightarrow$ nat

### 6.1.2 Symbolic variables of a store

The symbolic variable associated to a program variable $v$ by a store $s$ is the couple ( $v, s v$ ).
definition symvar ::
' $a \Rightarrow$ 'a store $\Rightarrow{ }^{\prime}$ a symvar
where
symvar $v s \equiv(v, s v)$
The function associating symbolic variables to program variables obtained from $s$ is injective.

## lemma

$\operatorname{inj}(\lambda$ v. symvar $v s)$
by (auto simp add : inj-on-def symvar-def)
The sets of symbolic variables of a store is the image set of the function symvar.

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


### 6.1.3 Fresh symbolic variables

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

```
definition fresh-symvar ::
    'v symvar }=>\mp@subsup{}{}{\prime}v\mathrm{ store }=>\mathrm{ bool
where
    fresh-symvar sv s = (sv & symvars s)
```


### 6.2 Consistency

We say that a program state $\sigma$ and a symbolic state $\sigma_{\text {sym }}$ are consistent with respect to a store $s$ if, for each variable $v$, the value associated by $\sigma$ to $v$ is equal to the value associated by $\sigma_{\text {sym }}$ to the symbolic variable associated to $v$ by $s$.

```
definition consistent ::
    \(\left({ }^{\prime} v,{ }^{\prime} d\right)\) state \(\Rightarrow\left({ }^{\prime} v\right.\) symvar, \(\left.{ }^{\prime} d\right)\) state \(\Rightarrow{ }^{\prime} v\) store \(\Rightarrow\) bool
where
    consistent \(\sigma \sigma_{\text {sym }} s \equiv\left(\forall v . \sigma_{\text {sym }}(\right.\) symvar \(\left.v s)=\sigma v\right)\)
```

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

## lemma

$\exists \sigma \sigma_{\text {sym }}$. consistent $\sigma \sigma_{\text {sym }} s$
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 $\sigma \sigma_{\text {sym }} s=\left(\forall\right.$ sv $\in$ symvars s. $\left.\sigma_{\text {sym }} s v=\sigma(f s t s v)\right)$
by (auto simp add : consistent-def symvars-def symvar-def)
lemma consistent-eq2 :
consistent $\sigma \sigma_{\text {sym }}$ store $=\left(\sigma=\left(\lambda v . \sigma_{\text {sym }}(\right.\right.$ symvar $v$ store $\left.\left.)\right)\right)$
by (auto simp add : consistent-def)

```
lemma consistentI1 :
    consistent \(\sigma(\lambda\) sv. \(\sigma(f s t s v))\) store
using consistent-eq1 by fast
lemma consistentI2 :
    consistent \(\left(\lambda v . \sigma_{\text {sym }}(\right.\) symvar \(v\) store \(\left.)\right) \sigma_{\text {sym }}\) 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 eto $s$ 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 ::
    (' \(\left.v,{ }^{\prime} d\right)\) aexp \(\Rightarrow\) 'v store \(\Rightarrow(' v\) symvar,'d) aexp
where
    adapt-aexp e \(s=\left(\lambda \sigma_{\text {sym }} . e\left(\lambda v \cdot \sigma_{s y m}(\right.\right.\) symvar \(\left.\left.v s)\right)\right)\)
```

Given an arithmetic expression $e$, a program state $\sigma$ and a symbolic state $\sigma_{\text {sym }}$ coherent with a store $s$, the value associated to $\sigma_{\text {sym }}$ by the adaptation of $e$ to $s$ is the same than the value associated by $e$ to $\sigma$. 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 \(\sigma \sigma_{\text {sym }} s\)
    shows (adapt-aexp e s) \(\sigma_{\text {sym }}=e \sigma\)
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))\not=e\sigma
    using assms unfolding Aexp.vars-def by blast
    moreover
    have (\lambdava. ((\lambdasv.\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[[f \sigma s]
        consistentIQ[of (\lambdasv.\sigma (fst sv))(?sv:= val) s]
    by (rule-tac ?x=\lambdasv.\sigma (fst sv) in exI, rule-tac ?x=val in exI)
        (simp add : adapt-aexp-is-subst)
qed
```

On the other hand, if $s v$ 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 \(s\left(\right.\) is Aexp.vars ? \(e^{\prime} \subseteq\) symvars \(\left.s\right)\)
unfolding subset-iff
proof (intro allI impI)
    fix \(s v\)
    assume sv \(\in\) Aexp.vars ? \(e^{\prime}\)
    then obtain \(\sigma_{\text {sym }}\) val
    where ? \(e^{\prime}\left(\sigma_{\text {sym }}(s v:=\right.\) val \(\left.)\right) \neq ? e^{\prime} \sigma_{\text {sym }}\)
    by (simp add : Aexp.vars-def, blast)
    hence \(\left(\lambda x .\left(\sigma_{\text {sym }}(s v:=\right.\right.\) val \(\left.)\right)(\) symvar \(\left.x s)\right) \neq\left(\lambda x . \sigma_{\text {sym }}(\operatorname{symvar} x s)\right)\)
    proof (intro notI)
    \(\operatorname{assume}\left(\lambda x .\left(\sigma_{\text {sym }}(s v:=\right.\right.\) val \(\left.)\right)(\) symvar \(\left.x s)\right)=\left(\lambda x . \sigma_{\text {sym }}(\operatorname{symvar} x s)\right)\)
    hence ? \(e^{\prime}\left(\sigma_{s y m}(s v:=v a l)\right)=? e^{\prime} \sigma_{s y m}\)
    by (simp add : adapt-aexp-def)
```

```
    thus False
    using «? 'e}(\mp@subsup{\sigma}{sym}{}(sv:=val))\not=? ?\mp@subsup{e}{}{\prime}\mp@subsup{\sigma}{sym}{}
    by (elim notE)
    qed
    then obtain v
    where ( }\mp@subsup{\sigma}{\mathrm{ sym }}{}(sv:=val))(\mathrm{ symvar v s)}\not=\mp@subsup{\sigma}{\mathrm{ sym }}{(symvar v s)
    by blast
    hence sv = symvar vs by (case-tac sv = symvar vs, simp-all)
    thus sv \in symvars s by (simp add : symvars-def)
qed
lemma symvar-var-imp-var :
    assumes sv \in Aexp.vars (adapt-aexp e s) (is sv \in Aexp.vars ?e')
    shows fst sv \in Aexp.vars e
proof -
    obtain v}\mathrm{ where sv = (v,sv)
    using assms(1) symvars-of-adapt-aexp
    unfolding symvars-def symvar-def
    by blast
    obtain }\mp@subsup{\sigma}{\mathrm{ sym }}{}\mathrm{ val where ? e' ( }\mp@subsup{\sigma}{\mathrm{ sym }}{}(sv:=val))\not=?\mp@subsup{e}{}{\prime}\mp@subsup{\sigma}{\mathrm{ sym}}{
    using assms unfolding Aexp.vars-def by blast
    moreover
    have}(\lambdav.(\mp@subsup{\sigma}{\mathrm{ sym }}{}(sv:=val))(symvar v s))=(\lambdav.\mp@subsup{\sigma}{\mathrm{ sym }}{}(\mathrm{ symvar v s)) (v:=
val)
    using <sv = (v,s v)〉 by (auto simp add : symvar-def)
    ultimately
    show ?thesis
    using <sv = (v,s v)>
```




```
    unfolding Aexp.vars-def
    by (simp add : adapt-aexp-is-subst) blast
qed
```

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

```
lemma adapt-aexp-vars :
    Aexp.vars (adapt-aexp e s)=(\lambda v. symvar v s)'Aexp.vars e
unfolding set-eq-iff image-def mem-Collect-eq Bex-def
proof (intro allI iffI, goal-cases)
    case (1 sv)
    moreover
    have sv = symvar (fst sv)s
    using 1 symvars-of-adapt-aexp
    by (force simp add: symvar-def symvars-def)
    ultimately
    show ?case using symvar-var-imp-var by blast
next
    case (2 sv) thus ?case using var-imp-symvar-var by fast
qed
```

The fact that the adaptation of an arithmetic expression to a store preserves finiteness of the set of variables trivially follows the previous lemma.
lemma finite-vars-imp-finite-adapt-a :
assumes finite (Aexp.vars e)
shows finite (Aexp.vars (adapt-aexp e s))
unfolding adapt-aexp-vars using assms by auto

### 6.4 Adaptation of a boolean expression to a store

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

```
definition adapt-bexp ::
    \(\left({ }^{\prime} v,{ }^{\prime} d\right)\) bexp \(\Rightarrow{ }^{\prime} v\) store \(\Rightarrow\left({ }^{\prime} v\right.\) symvar, \(\left.{ }^{\prime} d\right)\) bexp
where
    adapt-bexp e \(s=(\lambda \sigma . e(\lambda x . \sigma(\operatorname{symvar} x s)))\)
lemma adapt-bexp-is-subst :
    assumes consistent \(\sigma \sigma_{\text {sym }} s\)
    shows (adapt-bexp e s) \(\sigma_{\text {sym }}=e \sigma\)
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 \(\sigma\) val where \(A: e(\sigma(v:=v a l)) \neq e \sigma\)
```

using assms unfolding Bexp.vars-def by blast

## moreover

have $(\lambda v a .((\lambda s v . \sigma(f s t s v))(? s v:=v a l))(s y m v a r ~ v a ~ s))=(\sigma(v:=v a l))$
by (auto simp add : symvar-def)
ultimately
show ?thesis
unfolding Bexp.vars-def mem-Collect-eq
using consistentI1[of $\sigma s$ ]
consistentI2[of $(\lambda s v . \sigma(f s t ~ s v))(? s v:=v a l) s]$
by (rule-tac ? $x=\lambda s v . \sigma(f s t ~ s v)$ in exI, rule-tac ? $x=v a l$ 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 ? $e^{\prime} \subseteq$ ?SV)
proof
fix $s v$
assume $s v \in$ Bexp.vars ? $e^{\prime}$
then obtain $\sigma_{\text {sym }}$ val
where ? $e^{\prime}\left(\sigma_{\text {sym }}(s v:=\right.$ val $\left.)\right) \neq ? e^{\prime} \sigma_{\text {sym }}$
by (simp add : Bexp.vars-def, blast)
hence $\left(\lambda x .\left(\sigma_{\text {sym }}(s v:=\right.\right.$ val $\left.)\right)($ symvar $\left.x s)\right) \neq\left(\lambda x . \sigma_{\text {sym }}(\operatorname{symvar} x s)\right)$
by (auto simp add : adapt-bexp-def)
hence $\exists v .\left(\sigma_{\text {sym }}(s v:=v a l)\right)($ symvar $v s) \neq \sigma_{\text {sym }}($ symvar $v s)$ by force
then obtain $v$
where $\left(\sigma_{\text {sym }}(s v:=\right.$ val $\left.)\right)($ symvar $v s) \neq \sigma_{\text {sym }}($ symvar $v s)$
by blast
hence $s v=s y m v a r v s$ by $($ case-tac $s v=s y m v a r ~ v s$, simp-all $)$
thus $s v \in$ symvars $s$ by (simp add : symvars-def)
qed
lemma symvar-var-imp-var2 :
assumes $s v \in$ 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,sv)
    using assms symvars-of-adapt-bexp
    unfolding symvars-def symvar-def
    by blast
    obtain }\mp@subsup{\sigma}{sym}{}\mathrm{ val where ? e}\mp@subsup{e}{}{\prime}(\mp@subsup{\sigma}{sym}{}(sv:=val))\not=?\mp@subsup{e}{}{\prime}\mp@subsup{\sigma}{\mathrm{ sym}}{
    using assms unfolding vars-def by blast
    moreover
    have }(\lambdav.(\mp@subsup{\sigma}{\mathrm{ sym }}{}(sv:=val))(symvar v s))=(\lambdav.\mp@subsup{\sigma}{\mathrm{ sym }}{}(\mathrm{ symvar v s)) (v:=
val)
    using<sv = (v,s v)\rangle by (auto simp add : symvar-def)
    ultimately
    show ?thesis
    using <sv = (v,sv)>
```



```
        consistentI2[of \sigma क्sym (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)'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}\mathrm{ where vevars e sv=symvar vs 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 $\left({ }^{\prime} v,{ }^{\prime} d\right) \operatorname{conf}=$

```
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 \(\Rightarrow\left({ }^{\prime} v,{ }^{\prime} d\right)\) conf \(\Rightarrow\) bool
where
    fresh-symvar sv \(c=(s v \notin\) symvars \(c)\)
```


### 7.1.4 Satisfiability

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

```
abbreviation sat ::
    ( \({ }^{\prime} v,{ }^{\prime} d\) ) conf \(\Rightarrow\) bool
where
    sat \(c \equiv\) Bexp.sat \((\operatorname{conjunct}(\) 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 ::
    \(\left({ }^{\prime} v,{ }^{\prime} d\right)\) conf \(\Rightarrow\left({ }^{\prime} v,{ }^{\prime} d\right)\) state set
where
    states \(c=\left\{\sigma . \exists \sigma_{\text {sym }}\right.\). consistent \(\sigma \sigma_{\text {sym }}(\) store \(c) \wedge\) conjunct (pred \(\left.\left.c\right) \sigma_{\text {sym }}\right\}\)
```

A configuration is satisfiable if and only if its set of states is not empty.
lemma sat-eq :
sat $c=($ states $c \neq\{ \})$
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 ::
$\left({ }^{\prime} v,{ }^{\prime} d\right)$ conf $\Rightarrow\left({ }^{\prime} v,{ }^{\prime} d\right)$ conf $\Rightarrow$ bool (infixl $\left.\sqsubseteq 55\right)$
where
$c_{2} \sqsubseteq c_{1} \equiv\left(\right.$ states $c_{2} \subseteq$ states $\left.c_{1}\right)$
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 states $c 1=$ states $c 2$
using assms by (simp add : subsums-def)
A satisfiable configuration can only be subsumed by satisfiable configurations.
lemma sat-sub-by-sat :
assumes sat $c_{2}$
and $\quad c_{2} \sqsubseteq c_{1}$
shows sat $c_{1}$
using assms sat-eq[of $\left.c_{1}\right]$ sat-eq[of $\left.c_{2}\right]$
by (simp add: subsums-def) fast
On the other hand, an unsatisfiable configuration can only subsume unsatisfiable configurations.

```
lemma unsat-subs-unsat :
    assumes ᄀ sat c1
    assumes c2 \sqsubseteqc1
    shows \neg sat c2
using assms sat-eq[of c1] sat-eq[of c2]
by (simp add: subsums-def)
```


### 7.1.7 Semantics of a configuration

The semantics of a configuration $c$ is a boolean expression $e$ over program states associating true to a program state if it is a state of $c$. In practice, given two configurations $c_{1}$ and $c_{2}$, it is not possible to enumerate their sets of states to establish the inclusion in order to detect a subsumption. We detect the subsumption of the former by the latter by asking a constraint solver if sem $c_{1}$ entails sem $c_{2}$. The following theorem shows that the way we detect subsumption in practice is correct.

```
definition sem ::
    \(\left({ }^{\prime} v,{ }^{\prime} d\right)\) conf \(\Rightarrow\left({ }^{\prime} v,{ }^{\prime} d\right)\) bexp
where
    sem \(c=(\lambda \sigma . \sigma \in\) states \(c)\)
```


## theorem

$$
c_{2} \sqsubseteq c_{1} \longleftrightarrow \operatorname{sem}^{2} c_{2}=_{B} \operatorname{sem} c_{1}
$$

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 ::
    \(\left({ }^{\prime} v, ' d\right) \operatorname{conf} \Rightarrow\left({ }^{\prime} v, ' d\right)\) conf \(\Rightarrow\) bool
where
    abstract c \(c_{a} \equiv c \sqsubseteq c_{a}\)
```


### 7.1.9 Entailment

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

```
abbreviation entails ::
    ('v,'d) conf }=>(\mp@subsup{}{}{\prime}v,'d) bexp => bool (infixl \models=c 55)
where
    c\vDash\mp@subsup{\models}{c}{}\varphi\equiv\operatorname{sem}c|=\mp@subsup{|}{B}{}\varphi
lemma
    sem c \models}\mp@subsup{B}{B}{}e\longleftrightarrow(\forall\sigma\in\mathrm{ 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 no specify how this fresh variable is chosen (see the two assumptions in the third case). We could have model se (and se-star) by a function producing the new configuration, instead of using inductive predicates. However this would require to provide the two said assumptions in each lemma involving $s e$, 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 $s e$ and $s e \_$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 ::
    \(\left({ }^{\prime} v,{ }^{\prime} d\right)\) conf \(\Rightarrow\left({ }^{\prime} v, ' d\right)\) label \(\Rightarrow\left({ }^{\prime} v, ' d\right)\) conf \(\Rightarrow\) bool
where
    se c Skip c
\(\mid\) se \(c(\) Assume e) \(\\) store \(=\) store \(c\), pred \(=\) pred \(c \cup\{\) adapt-bexp e (store \(c)\} \mid\)
\(\mid f s t s v=v \quad \Longrightarrow\)
    fresh-symvar sv \(c \Longrightarrow\)
    se \(c(\) Assign \(v e) \cap\) store \(=(\) store \(c)(v:=\) snd sv \()\),
                        pred \(=\) pred \(c \cup\{(\lambda \sigma . \sigma s v=(\) adapt-aexp \(e(\) store \(c)) \sigma)\} D\)
```

In the same spirit, we extend symbolic execution to sequence of labels.
inductive se-star :: ('v,'d) conf $\Rightarrow(' v, ' d)$ label list $\Rightarrow\left({ }^{\prime} v, ' d\right)$ conf $\Rightarrow$ bool where se-star c [] c
$\mid$ se c1 l c2 $\Longrightarrow$ se-star c2 ls c3 $\Longrightarrow$ se-star c1 $(l \# l s) c 3$

### 7.2.2 Basic properties of se

If symbolic execution yields a satisfiable configuration, then it has been performed from a satisfiable configuration.
lemma se-sat-imp-sat :
assumes se clc $c^{\prime}$
assumes sat $c^{\prime}$
shows sat c
using assms by cases (auto simp add : sat-def conjunct-def)
If symbolic execution is performed from an unsatisfiable configuration, then it will yield an unsatisfiable configuration.

```
lemma unsat-imp-se-unsat :
    assumes se clc \(c^{\prime}\)
    assumes \(\neg\) sat \(c\)
    shows \(\neg\) sat \(c^{\prime}\)
using assms by cases (simp add : sat-def conjunct-def)+
```

Given two configurations $c$ and $c^{\prime}$ and a label $l$ such that se $c l c^{\prime}$, the three following lemmas express $c^{\prime}$ as a function of $c$.
lemma [simp]:

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

lemma se-Assume-eq :
se $c\left(\right.$ Assume e) $c^{\prime}=\left(c^{\prime}=\\right.$ store $=$ store $c$, pred $=$ pred $c \cup\{$ adapt-bexp e
(store $c)\}$ ))
by (simp add : se.simps)
lemma se-Assign-eq :
se $c($ Assign $v e) c^{\prime}=$
( $\exists$ sv. fresh-symvar sv $c$
$\wedge f s t s v=v$
$\wedge c^{\prime}=($ store $=($ store $c)(v:=$ snd sv $)$,
pred $=\operatorname{insert}(\lambda \sigma . \sigma$ sv $=$ adapt-aexpe $($ store $c) \sigma)($ pred $c) D)$
by (simp only : se.simps, blast)

Given two configurations $c$ and $c^{\prime}$ and a label $l$ such that se $c l c^{\prime}$, the two following lemmas express the path predicate of $c^{\prime}$ 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) \(c^{\prime}\)
    shows conjunct (pred \(c^{\prime}\) ) \(=\)
            \((\lambda \sigma\). conjunct \((\) pred \(c) \sigma \wedge\) adapt-bexp \(e(\) store \(c) \sigma)\)
using assms se-Assume-eq[of cee \(\left.\begin{array}{c} \\ \end{array}\right]\)
by (auto simp add : conjunct-def)
```

lemma path-pred-of-se-Assign :
assumes se $c(A s s i g n v e) c^{\prime}$
shows $\exists$ sv. conjunct $\left(\right.$ pred $\left.c^{\prime}\right)=$
$(\lambda \sigma$. conjunct $($ pred $c) \sigma \wedge \sigma s v=$ adapt-aexp $e($ store $c) \sigma)$
using assms se-Assign-eq[of cure $\left.\begin{array}{c}\text { l }\end{array}\right]$
by (fastforce simp add : conjunct-def)

Let $c$ and $c^{\prime}$ be two configurations such that $c^{\prime}$ is obtained from $c$ by symbolic execution of a label of the form Assume e. The states of $c^{\prime}$ 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\mathrm{ states c.e e }\sigma
using assms se-Assume-eq[of c e c c]
```

by（auto simp add ：adapt－bexp－is－subst states－def conjunct－def）
Let $c$ and $c^{\prime}$ be two configurations such that $c^{\prime}$ 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 $c^{\prime}$ as a function of the set of states of $c$ ．Since the proof requires a number of details，we split into two sub lemmas．

First，we show that if $\sigma^{\prime}$ is a state of $c^{\prime}$ ，then it has been obtain from an adequate update of a state $\sigma$ of $c$ ．

```
lemma states-of-se-assign1 :
    assumes se c (Assign ve) c'
    assumes }\mp@subsup{\sigma}{}{\prime}\in\mathrm{ states c'
    shows }\exists\sigma\in\mathrm{ states c. }\mp@subsup{\sigma}{}{\prime}=(\sigma(v:=e\sigma)
proof -
    obtain }\mp@subsup{\sigma}{sym}{
    where 1: consistent }\mp@subsup{\sigma}{}{\prime}\mp@subsup{\sigma}{\mathrm{ sym }}{}\mathrm{ (store c')
    and 2 : conjunct (pred c') }\mp@subsup{\sigma}{\mathrm{ sym }}{
    using assms(2) unfolding states-def by blast
    then obtain }
    where 3 : consistent \sigma \sigma sym (store c)
    using consistentI2 by blast
    moreover
    have conjunct (pred c) \sigma sym
    using assms(1) 2 by (auto simp add : se-Assign-eq conjunct-def)
    ultimately
    have }\sigma\in\mathrm{ states c by (simp add : states-def) blast
    moreover
    have }\mp@subsup{\sigma}{}{\prime}=\sigma(v:=e\sigma
    proof -
        have }\mp@subsup{\sigma}{}{\prime}v=e
        proof -
        have }\mp@subsup{\sigma}{}{\prime}v=\mp@subsup{\sigma}{\mathrm{ sym }}{}(\mathrm{ symvar v (store c}\mp@subsup{c}{}{\prime})
        using 1 by (simp add : consistent-def)
        moreover
        have }\mp@subsup{\sigma}{\mathrm{ sym }}{}(\mathrm{ symvar v (store c}\mp@subsup{c}{}{\prime}))=(\mathrm{ adapt-aexp e (store c)) 的sym
        using assms(1) 2 se-Assign-eq[of c vel']
        by (force simp add : symvar-def conjunct-def)
        moreover
        have (adapt-aexp e (store c)) 的和}=e \sigma
```

```
        using 3 by (rule adapt-aexp-is-subst)
        ultimately
        show ?thesis by simp
    qed
    moreover
    have }\forallx.x\not=v\longrightarrow\mp@subsup{\sigma}{}{\prime}x=\sigma
    proof (intro allI impI)
        fix }
        assume }x\not=
        moreover
        hence }\mp@subsup{\sigma}{}{\prime}x=\mp@subsup{\sigma}{\mathrm{ sym }}{(symvar x (store c))
        using assms(1) 1 unfolding consistent-def symvar-def
        by (drule-tac ? }x=x\mathrm{ in spec) (auto simp add:se-Assign-eq)
        moreover
        have }\mp@subsup{\sigma}{\mathrm{ sym }}{}(\mathrm{ symvar x (store c)) = 利
        using 3 by (auto simp add : consistent-def)
        ultimately
        show }\mp@subsup{\sigma}{}{\prime}x=\sigmax\mathrm{ 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 \(\sigma\) of \(c\) from which \(\sigma^{\prime}\) is obtained by an adequate update, then \(\sigma^{\prime}\) is a state of \(c^{\prime}\).
lemma states-of-se-assign2 :
    assumes se c(Assign ve) c'
    assumes }\exists\sigma\in\mathrm{ states c. }\mp@subsup{\sigma}{}{\prime}=\sigma(v:=e\sigma
    shows }\mp@subsup{\sigma}{}{\prime}\in\mathrm{ states c'
proof -
    obtain }
    where \sigma\in states c
    and }\quad\mp@subsup{\sigma}{}{\prime}=\sigma(v:=e\sigma
    using assms(2) by blast
```

```
then obtain 的ym
where 1:consistent \sigma \sigma sym (store c)
and 2:conjunct (pred c) }\mp@subsup{\sigma}{sym}{
unfolding states-def by blast
obtain sv
where 3:fresh-symvar sv c
and 4:fst sv=v
and 5:c'=( store = (store c)(v:= snd sv),
    pred =insert (\lambda\sigma.\sigma sv =adapt-aexp e (store c) \sigma)(pred c) D
using assms(1) se-Assign-eq[of cceeec` by blast
define }\mp@subsup{\sigma}{sym}{\prime}\mp@subsup{}{}{\prime}\mathrm{ where }\mp@subsup{\sigma}{sym}{\prime}\mp@subsup{}{}{\prime}=\mp@subsup{\sigma}{sym}{(sv}:=e \sigma
have consistent }\mp@subsup{\sigma}{}{\prime}\mp@subsup{\sigma}{\mathrm{ sym }}{\prime
using «\sigma'=\sigma (v:=e\sigma)> 145
by (auto simp add : symvar-def consistent-def \sigmasym}\mp@subsup{}{}{\prime}-def
moreover
have conjunct (pred c') \sigma क्sym}\mp@subsup{}{}{\prime
proof -
```



```
    using 2 3 by (simp add :fresh-symvar-def symvars-def Bexp.vars-def \sigmasym}\mp@subsup{}{\mathrm{ '-def)}}{\mathrm{ )}
    moreover
    have }\mp@subsup{\sigma}{\mathrm{ sym}}{\prime}'sv=(\mathrm{ adapt-aexp e (store c)) }\mp@subsup{\sigma}{\mathrm{ sym }}{}\mp@subsup{}{}{\prime
    proof -
    have Aexp.fresh sv (adapt-aexp e (store c))
    using 3 symvars-of-adapt-aexp[of e store c]
    by (auto simp add:fresh-symvar-def symvars-def)
        thus ?thesis
        using adapt-aexp-is-subst[OF 1, of e]
        by (simp add: Aexp.vars-def \sigmasym}\mp@subsup{}{}{\prime}-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 $c^{\prime}$ 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)|\sigma.\sigma\in states c
using assms states-of-se-assign1 states-of-se-assign2 by fast
```


### 7.2.3 Monotonicity of $s e$

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

```
theorem se-mono-for-sub:
    assumes se c1lc1'
    assumes se c2 l c2'
    assumes c2 \sqsubseteqc1
    shows c\mp@subsup{2}{}{\prime}\sqsubseteqc1'
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 \(c 1=\) states \(c 2\)
    assumes se c1 lc1'
    assumes se c2 lc2'
    shows states \(c 2^{\prime}=\) states \(c 1^{\prime}\)
using assms(1)
        se-mono-for-sub \([O F \operatorname{assms}(2,3)]\)
        se-mono-for-sub[OF \(\operatorname{assms}(3,2)]\)
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 lc1
assumes se clc2
shows states c1 = states c2
using assms se-mono-for-states-eq by fast

### 7.2.4 Basic properties of se

$\qquad$ star

Some simplification lemmas for se-star.
lemma $[$ simp $]$ :
se-star $c[] c^{\prime}=\left(c^{\prime}=c\right)$
by (subst se-star.simps) auto
lemma se-star-Cons :
se-star c1 $(l \# l s) c 2=(\exists$ c. se c1 l c $\wedge$ 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 $=(\exists$ c. se-star c1 ls1 $c \wedge$ 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 $=(\exists$ c. se-star c1 ls c $\wedge$ 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 $c^{\prime}$
assumes $\neg$ sat $c$
shows $\neg$ sat $c^{\prime}$
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 $c l s c^{\prime}$
assumes sat $c^{\prime}$
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 \sqsubseteqc1
    shows c2'}\sqsubseteqc\mp@subsup{1}{}{\prime
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 $c \mathcal{Q}^{\prime}=$ states $c 1^{\prime}$
using assms(1)
se-star-mono-for-sub[OF $\operatorname{assms}(2,3)]$
se-star-mono-for-sub $[O F \operatorname{assms}(3,2)]$
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 $c 2$
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 ::

```
    \(\left({ }^{\prime} v,{ }^{\prime} d\right)\) conf \(\Rightarrow\) bool
where
    updatable \(c \equiv \forall v . \exists\) sv. fst \(s v=v \wedge\) 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}\mp@subsup{c}{}{\prime}.\mathrm{ se cl 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 $S V$ and a program variable $v$ such that there exist symbolic variables in $S V$ 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 $S V$.

```
lemma finite-symvars-imp-ex-greatest-symvar :
    fixes \(S V\) :: 'a symvar set
    assumes finite \(S V\)
    assumes \(\exists s v \in S V . f s t s v=v\)
    shows \(\exists s v \in\{s v \in S V . f s t s v=v\}\).
            \(\forall s v^{\prime} \in\{s v \in S V . f s t s v=v\}\). snd \(s v^{\prime} \leq s n d s v\)
proof -
    have finite (snd' \(\{s v \in S V\). fst \(s v=v\}\) )
    and \(s n d\) ' \(\{s v \in S V\). fst \(s v=v\} \neq\{ \}\)
    using assms by auto
    moreover
    have \(\forall\) (E::nat set). 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 snd' '{sv\inSV.fst sv=v}
and }\forallm\insnd'{sv\inSV.fst sv=v}.m\leq
by blast
    moreover
    then obtain sv
    where sv\in{sv\inSV.fst sv=v} and 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 \existssv.fst sv = v^ fresh-symvar sv c
    proof (case-tac \existssv\in?V.fst sv = v, goal-cases)
        case 1
    then obtain max-sv
    where max-sv\in?V
    and fst max-sv = v
    and max :}\foralls\mp@subsup{v}{}{\prime}\in{sv\in?V.fst sv=v}.snd sv'\leq snd max-s
    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 snd max-sv \leq store c v, goal-cases)
        case 1 thus?case by (rule-tac ? }x=(v,Suc (store c v)) in exI) aut
    next
        case 2 thus ?case by (rule-tac ? }x=(v,Suc (snd max-sv)) in exI) aut
```

```
        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 }\foralle\in\mathrm{ pred c. finite (Bexp.vars e)
    shows \exists}\mp@subsup{c}{}{\prime}\mathrm{ . 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 subexpression 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 cl c'
    assumes }\foralle\in\mathrm{ pred c. finite (Bexp.vars e)
```

```
    shows }\foralle\in\mathrm{ pred c'. finite (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 ve)
    then obtain sv
    where fresh-symvar sv c
    and fst sv =v
    and }\mp@subsup{c}{}{\prime}=(\mathrm{ store }=(\mathrm{ store c)(v:= snd sv),
    pred = insert (\lambda\sigma.\sigma sv =adapt-aexp e (store c) \sigma) (pred c))
    using assms(2) se-Assign-eq[of c veec` by blast
    moreover
    have finite (Bexp.vars (\lambda\sigma.\sigma sv =adapt-aexp e(store c) \sigma))
    proof -
    have finite (Aexp.vars ( }\lambda\sigma.\sigma sv)
    by (auto simp add: Aexp.vars-def)
    moreover
    have finite (Aexp.vars (adapt-aexp 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 cl 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 }\foralle\in\mathrm{ pred c. finite (Bexp.vars e)
    assumes finite-labels ls
    shows \exists c'. se-star c 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 cl c1 using finite-imp-ex-se-succ by blast
    hence finite (pred c1)
    and }\foralle\in\mathrm{ pred c1. finite (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 <se c l c1` using se-star-Cons by blast
qed
```


### 7.3 Feasibility of a sequence of labels

A sequence of labels $l s$ is said to be feasible from a configuration $c$ if there exists a satisfiable configuration $c^{\prime}$ obtained by symbolic execution of $l s$ from c.
definition feasible :: ('v,'d) conf $\Rightarrow\left({ }^{\prime} v, ' d\right)$ label list $\Rightarrow$ bool where
feasible $c l s \equiv\left(\exists c^{\prime}\right.$. se-star cls $c^{\prime} \wedge$ sat $\left.c^{\prime}\right)$
A simplification lemma for the case where $l s$ is not empty.
lemma feasible-Cons :
feasible $c(l \# l s)=\left(\exists c^{\prime}\right.$.se c $l c^{\prime} \wedge$ sat $c^{\prime} \wedge$ feasible $\left.c^{\prime} l s\right)$
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 $c 1$ and $c 2$ such that $c 1$ subsums $c 2$, then any feasible sequence of labels from $c 2$ is also feasible from $c 1$. This is a crucial point in order to prove that our approach preserves the set of feasible paths of the original LTS. This proof requires a number of assumptions about the finiteness of the sequence of labels, of the path predicates of the two configurations and of their states of variables. Those assumptions are needed in order to show that there exist successors of both configurations by symbolic execution of the sequence of labels.

```
lemma subsums-imp-feasible :
    assumes finite-labels ls
    assumes finite (pred c1)
    assumes finite (pred c2)
    assumes \(\forall e \in\) pred c1. finite (Bexp.vars e)
    assumes \(\forall e \in\) pred c2. finite (Bexp.vars e)
    assumes \(c 2 \sqsubseteq c 1\)
    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 \(c 2^{\prime}\) where se \(c 2 l c 2^{\prime}\)
            and sat \(c 2^{\prime}\)
            and feasible c2' \(l s\)
    using feasible-Cons by blast
    obtain \(c 1^{\prime}\) where se c1 lc1'
    using finite-conj[OF \(\operatorname{Cons}(3,5)]\)
        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〈sec2 l c2'>Cons(4)])
    moreover
    have }\foralle\inpred c1'. finite (Bexp.vars e)
    by (rule se-preserves-finiteness1[OF〈finite-label l\rangle\langlese c1 l c1'〉Cons(5)])
    moreover
    have }\foralle\inpred c2'. finite (Bexp.vars e)
    by (rule se-preserves-finiteness1[OF<finite-label l〉\langlese c2 l c2'〉Cons(6)])
    moreover
    have }c\mp@subsup{2}{}{\prime}\sqsubseteqc\mp@subsup{1}{}{\prime
    by (rule se-mono-for-sub[OF <se c1 l c1'〉\langlese 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 $c e$ ，an inductive predicate describing concrete execution．Unlike symbolic execu－ tion，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 ::
    \(\left({ }^{\prime} v,{ }^{\prime} d\right)\) state \(\Rightarrow\left({ }^{\prime} v,{ }^{\prime} d\right)\) label \(\Rightarrow\left({ }^{\prime} v,{ }^{\prime} d\right)\) state \(\Rightarrow\) bool
where
    ce \(\sigma\) Skip \(\sigma\)
| e \(\sigma \Longrightarrow c e \sigma\) (Assume e) \(\sigma\)
| ce \(\sigma(\) Assign \(v e)(\sigma(v:=e \sigma))\)
inductive ce-star :: ('v,'d) state \(\Rightarrow(' v, ' d)\) label list \(\Rightarrow(' v, ' d)\) state \(\Rightarrow\) bool where
    ce-star \(c[] c\)
\(\mid\) ce c1 l c2 \(\Longrightarrow\) ce-star c2 ls c3 \(\Longrightarrow\) ce-star c1 \((l \# l s) c 3\)
lemma \([\) simp \(]\) :
    ce \(\sigma\) Skip \(\sigma^{\prime}=\left(\sigma^{\prime}=\sigma\right)\)
by (auto simp add : ce.simps)
lemma \([\) simp \(]\) :
    ce \(\sigma\) (Assume e) \(\sigma^{\prime}=\left(\sigma^{\prime}=\sigma \wedge e \sigma\right)\)
by (auto simp add : ce.simps)
lemma \([\) simp \(]\) :
    ce \(\sigma\left(\right.\) Assign ve) \(\sigma^{\prime}=\left(\sigma^{\prime}=\sigma(v:=e \sigma)\right)\)
by (auto simp add : ce.simps)
lemma se-as-ce :
    assumes se \(c l c^{\prime}\)
    shows states \(c^{\prime}=\left\{\sigma^{\prime} . \exists \sigma \in\right.\) states \(c\). ce \(\left.\sigma l \sigma^{\prime}\right\}\)
using assms
by (cases l)
    (auto simp add: states-of-se-assume states-of-se-assign)
lemma \([\) simp \(]\) :
    ce-star \(\sigma[] \sigma^{\prime}=\left(\sigma^{\prime}=\sigma\right)\)
by (subst ce-star.simps) simp
lemma ce-star-Cons :
    ce-star \(\sigma 1(l \# l s) \sigma 2=(\exists \sigma\). ce \(\sigma 1 l \sigma \wedge c e-s t a r ~ \sigma l s ~ \sigma 2) ~\)
by (subst (1) ce-star.simps) blast
```

```
lemma se-star-as-ce-star :
    assumes se-star c ls c'
    shows states c'}={\mp@subsup{\sigma}{}{\prime}.\exists\sigma\in\mathrm{ states c.ce-star }\sigmals\mp@subsup{\sigma}{}{\prime}
using assms
proof (induct ls arbitrary : c)
    case Nil thus ?case by simp
next
    case (Cons l ls c)
    then obtain c"|}\mathrm{ where se cl c'|
            and se-star c'lls 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 }\mp@subsup{\sigma}{}{\prime\prime}\mathrm{ where }\mp@subsup{\sigma}{}{\prime\prime}\in\mathrm{ states c "
            and ce-star \sigma" ls \sigma'
    using Cons(1)\langlese-star c }\mp@subsup{c}{}{\prime\prime}ls\mp@subsup{c}{}{\prime}〉 by blas
    moreover
    then obtain \sigma where \sigma\in states c
        and ce \sigmal \sigma'
    using <se c l c'>}\mp@subsup{}{}{\prime\prime}\mathrm{ 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 }\mp@subsup{\sigma}{}{\prime\prime}\mathrm{ where ce }\sigmal\mp@subsup{\sigma}{}{\prime\prime
                and ce-star \sigma" ls \sigma'
    using ce-star-Cons by blast
    ultimately
    show ?case
    using Cons(1)\langlese-star c"ls c'\rangle\langlese c l c'> 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 controlflow 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 }=>\mathrm{ ('var,'d) label
```

We call initial location the root of the underlying graph.

```
abbreviation init ::
('vert,'var,'d,'x) lts-scheme }=>\mathrm{ '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 $\Rightarrow\left({ }^{\prime} v a r,{ }^{\prime} d\right)$ 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 }=>\mathrm{ ('vert edge }=>('var,'d) label) => ('var,'d) label list
```


## where

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

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

## abbreviation graph ::

('vert,'var,'d,'x) lts-scheme $\Rightarrow$ 'vert 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) lts-scheme => bool
where
    wf-lts lts \equivwf-rgraph (graph lts)
```

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

```
abbreviation finite-lts ::
    ('vert,'var,'d,'x) lts-scheme => bool
where
    finite-lts lts \equiv\foralll\in range (labelling lts). finite-label l
```


### 8.2 Feasible sub-paths and paths

A sequence of edges is a feasible sub-path of an LTS lts from a configuration $c$ if it is a sub-path of the underlying graph of $l t s$ and if it is feasible from the configuration $c$.
abbreviation feasible-subpath ::

```
('vert,'var,' \(d\), ,'x) lts-scheme \(\Rightarrow\left({ }^{\prime}\right.\) var,'d) conf \(\Rightarrow{ }^{\prime}\) 'vert \(\Rightarrow{ }^{\prime}\) 'vert edge list \(\Rightarrow\) 'vert
\(\Rightarrow\) bool
where
    feasible-subpath lts pc l1 as l2 \(\equiv\) Graph.subpath lts l1 as l2
    \(\wedge\) feasible pc (trace as (labelling lts))
```

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

$$
\left({ }^{\prime} v e r t, \text { 'var,'d,'x) lts-scheme } \Rightarrow(' v a r, ' d) \text { conf } \Rightarrow{ }^{\prime} \text { vert } \Rightarrow{ }^{\prime} \text { vert edge list } \Rightarrow\right. \text { bool }
$$ where

feasible-subpath-from lts pc las $\equiv \exists l^{\prime}$. feasible-subpath lts pc las l'
abbreviation feasible-subpaths-from ::
('vert,'var,'d,'x) lts-scheme $\Rightarrow\left({ }^{\prime} v a r,{ }^{\prime} d\right)$ conf $\Rightarrow{ }^{\prime}$ 'vert $\Rightarrow{ }^{\prime}$ 'vert edge list set where
feasible-subpaths-from lts pc $l \equiv\{$ ts. feasible-subpath-from lts pc lts $\}$
As earlier, feasible paths are defined as feasible sub-paths starting at the initial location of the LTS.

```
abbreviation feasible-path ::
    ('vert,'var,'d,'x) lts-scheme }=>('var,'d) conf => 'vert edge list = ' 'vert = bool
where
    feasible-path lts pc as l \equiv feasible-subpath lts pc (init lts) as l
```

abbreviation feasible-paths ::
('vert,'var, $\left.{ }^{\prime} d, ' x\right)$ lts-scheme $\Rightarrow\left(' v a r,{ }^{\prime} d\right)$ conf $\Rightarrow{ }^{\prime}$ vert edge list set
where
feasible-paths lts $p c \equiv\{a s . \exists$ l. feasible-path lts pc as $l\}$
end
theory SubRel
imports Graph
begin

## 9 Graphs equipped with a subsumption relation

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

### 9.1 Basic definitions and properties

### 9.1.1 Subsumptions and subsumption relations

Subsumptions take place between occurrences of the vertices of a graph. We represent such occurrences by indexed versions of vertices. A subsumption is defined as pair of indexed vertices.
type-synonym 'v sub-t $=\left(\left({ }^{\prime} v \times n a t\right) \times\left({ }^{\prime} v \times n a t\right)\right)$
A subsumption relation is a set of subsumptions.
type-synonym 'v sub-rel-t $=$ 'v sub-t set
We consider the left member to be subsumed by the right one. The left member of a subsumption is called its subsumee, the right member its subsumer.
abbreviation subsumee ::
$' v$ sub- $t \Rightarrow\left({ }^{\prime} v \times n a t\right)$
where
subsumee sub $\equiv f$ st sub
abbreviation subsumer ::
$' v$ sub- $t \Rightarrow(' v \times n a t)$
where
subsumer sub $\equiv$ snd sub
We will need to talk about the sets of subsumees and subsumers of a subsumption relation.

```
abbreviation subsumees ::
    'v sub-rel- \(t \Rightarrow(' v \times n a t)\) set
where
    subsumees subs \(\equiv\) subsumee'subs
```

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

The two following lemmas will prove useful in the following.

```
lemma subsumees-conv:
    subsumees subs ={v.\exists v'. (v,\mp@subsup{v}{}{\prime})\in\mathrm{ subs }}
```

by force

## lemma subsumers-conv:

subsumers subs $=\left\{v^{\prime} . \exists v .\left(v, v^{\prime}\right) \in\right.$ 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 $\Rightarrow\left({ }^{\prime} v \times n a t\right)$ 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 \(w f\)-sub-rel \(=\) sub-rel +
    assumes sub-imp-same-verts :
        sub \(\in\) subs \(\Longrightarrow f s t(\) subsumee sub \()=f s t(\) subsumer sub \()\)
    assumes subsumed-by-one :
        \(\forall v \in\) subsumees subs. \(\exists!v^{\prime} .\left(v, v^{\prime}\right) \in\) 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 \((v, v 1) \in\) subs
    assumes \((v, v 2) \in s u b s\)
    shows \(\quad v 1=v_{2}\)
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 \(\left(v, v^{\prime}\right) \in\) subs \(^{+}\)
    shows \(\left(v, v^{\prime}\right) \in\) subs
using tranclD[OF assms] tranclD[of - \(v^{\prime}\) subs]
        rtranclD[of - \(\left.v^{\prime} s u b s\right]\)
        inter-empty
    by force
    lemma trancl-eq :
        subs \(^{+}=\)subs
    using in-trancl-imp r-into-trancl \([0 f-\)-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}\mathrm{ 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 $v 1$ to $v 2$ if these two vertices are equal or if they belong to the transitive closure of the relation. A non-empty sequence is consistent if it is made of consistent sequences whose extremities are linked in the transitive closure of the subsumption relation.
fun ces $::(' v \times n a t) \Rightarrow(' v \times n a t)$ edge list $\Rightarrow\left({ }^{\prime} v \times n a t\right) \Rightarrow{ }^{\prime} v$ sub-rel- $t \Rightarrow$ bool where
ces $v 1$ [] v2 subs $=\left(v 1=v 2 \vee(v 1, v 2) \in\right.$ subs $\left.^{+}\right)$
$\mid$ ces v1 $(e \# e s)$ v2 subs $=\left(\left(v 1=s r c e \vee(v 1, s r c e) \in s u b s^{+}\right) \wedge\right.$ ces $(t g t e)$ es v2 subs)

A consistent sequence from $v 1$ to $v 2$ without a subsumption relation is consistent between these two vertices in presence of any relation.

```
lemma
    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 $\left(v 1, v_{2}\right)$ be an element of a subsumption relation, and es a sequence of edges consistent wrt. this relation from vertex $v 2$. Then es is also consistent from 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 $(v 1, v 2) \in \operatorname{subs}$
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 subs $1 \subseteq$ 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 ves2 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 \(v 1[] v\) subs
            and ces ves2 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 $v 1$ to $v 2$ wrt. a sub-relation subs of a graph $g$. Suppose elements of this sequence are edges of $g$. If $v 1$ is
a vertex of $g$ then $v 2$ is also a vertex of $g$.
lemma (in sub-rel-of) ces-imp-ends-vertices :
assumes ces v1 es v2 subs
assumes set es $\subseteq$ edges $g$
assumes $v 1 \in$ Graph.vertices $g$
shows $v 2 \in G r a p h . v e r t i c e s ~ g ~$
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 $v 1$ to $v 2$, two vertices of a graph $g$ equipped with a subsumption relation subs, is a sequence of edges consistent wrt. subs from $v 1$ to $v 2$ whose elements are edges of $g$. Moreover, we must assume that subs is a sub-relation of $g$, otherwise es could "exit" $g$ through subsumption links.

```
definition subpath ::
\(\Rightarrow\left(\left({ }^{\prime} v \times n a t\right) \times\left({ }^{\prime} v \times n a t\right)\right)\) set \(\Rightarrow\) bool
where
    subpath g v1 es v2 subs \(\equiv\) sub-rel-of \(g\) subs
    \(\wedge v 1 \in\) Graph.vertices \(g\)
    \(\wedge\) ces v1 es v2 subs
    \(\wedge\) set es \(\subseteq\) edges \(g\)
```

    \(\left(\left({ }^{\prime} v \times n a t\right),{ }^{\prime} x\right)\) rgraph-scheme \(\Rightarrow\left({ }^{\prime} v \times n a t\right) \Rightarrow\left({ }^{\prime} v \times n a t\right)\) edge list \(\Rightarrow\left({ }^{\prime} v \times n a t\right)\)
    Once again, in some cases, we will not be interested in the ending vertex of a sub-path.
abbreviation subpath-from ::
$((' v \times n a t), ' x)$ rgraph-scheme $\Rightarrow\left({ }^{\prime} v \times n a t\right) \Rightarrow\left({ }^{\prime} v \times n a t\right)$ edge list $\Rightarrow{ }^{\prime} v$ sub-rel-t
$\Rightarrow$ bool
where
subpath-from $g$ ves subs $\equiv \exists v^{\prime}$. subpath $g v$ es $v^{\prime}$ subs
Simplification lemmas for SubRel.subpath.
lemma Nil-sp :
subpath $g$ v1[] v2 subs $\longleftrightarrow$ sub-rel-of $g$ subs
$\wedge v 1 \in$ Graph.vertices $g$
$\wedge\left(v 1=v 2 \vee(v 1, v 2) \in s u b s^{+}\right)$
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

$$
\begin{aligned}
& \wedge v 1 \in \text { Graph.vertices } g \\
& \wedge(v 1=v 2 \vee(v 1, v 2) \in \text { subs })
\end{aligned}
$$

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
$\wedge\left(v 1=\operatorname{src} e \vee(v 1\right.$, src $\left.e) \in \operatorname{subs}^{+}\right)$
$\wedge e \in$ edges $g$
$\wedge\left(\right.$ tgt $\left.e=v \mathcal{Z} \vee(t g t e, v \mathcal{Z}) \in \operatorname{subs}^{+}\right)$
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

$$
\begin{aligned}
& \wedge(v 1=\text { src } e \vee(v 1, \text { src } e) \in \text { subs }) \\
& \wedge e \in \text { edges } g \\
& \wedge(\text { tgt } e=v 2 \vee(\text { tgt e,v2 }) \in \text { subs })
\end{aligned}
$$

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 $\wedge\left(v 1=\operatorname{src} e \vee(v 1, s r c e) \in\right.$ subs $\left.^{+}\right)$ $\wedge e \in$ 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

$$
\begin{aligned}
& \wedge(v 1=\operatorname{src} e \vee(v 1, \text { src } e) \in \text { subs }) \\
& \wedge e \in \text { edges } g \\
& \wedge \text { subpath } g(\text { tgt e) es v2 subs }
\end{aligned}
$$

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 gv1 (es @ [e]) v2 subs $\longleftrightarrow$ subpath $g$ v1 es (src e) subs

$$
\begin{aligned}
& \wedge e \in \text { edges } g \\
& \wedge\left(\text { tgt } e=v 2 \vee(\text { tgt } e, v 2) \in \text { subs }^{+}\right)
\end{aligned}
$$

unfolding subpath-def by (auto simp add: ces-append-one)
Simpler version in the case of a well-formed subsumption relation.

$$
\begin{aligned}
& \text { lemma (in wf-sub-rel) sp-append-one : } \\
& \begin{aligned}
& \text { subpath } g v 1(e s @[e]) v 2 \text { subs } \longleftrightarrow \text { subpath } g \text { v1 es }(\text { src e) subs } \\
& \wedge e \in \text { edges } g \\
& \wedge(\text { tgt } e=v 2 \vee(\text { tgt } e, v 2) \in \text { subs })
\end{aligned}
\end{aligned}
$$

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 gv1 (es1 @ es2)v2 subs \longleftrightarrow
        (\exists v. subpath g v1 es1 v subs ^ 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 $v 1$. 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 $\quad v 1 \in$ 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 $\quad v 2 \in$ 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 subs
    shows subpath g v1 es v3 subs
proof (cases es)
    case Nil
    moreover
    hence v1 \in Graph.vertices g
    and v1 = v2 \vee (v1,v2) \in 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 \(e s^{\prime} e\) where \(e s=e s^{\prime} @[e]\) 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 $v 1$ whose set of out-edges is empty is either:

1. empty,
2. a sub-path starting at the subsumer $v 2$ of $v 1$.

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 $(v 1, v 2) \in$ subs
assumes subpath g v1 es $v$ subs
assumes out-edges g v1 $=\{ \}$
shows es $=[] \vee$ 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 $v 1$, then es could be empty or it could also be a non-empty sub-path leading from $v 2$ to $v 1$. If $v$ is not $v 1$, it could be $v^{2}$ 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 \(\notin\) subsumees subs
    assumes out-edges \(g\) v1 \(=\{ \}\)
    assumes subpath \(g\) v1 es v2 subs
    shows es = []
using assms tranclD by (cases es) (auto simp add : sp-Cons, force)
```

Let $e$ be an edge whose target is not subsumed and has not out-going edges. A sub-path es containing $e$ ends by $e$ and this occurrence of $e$ is unique along es.
lemma sp-through-de-decomp :
assumes tgt $e \notin$ subsumees subs
assumes out-edges $g($ tgt e) $=\{ \}$
assumes subpath $g$ v1 es v2 subs
assumes $e \in$ set es
shows $\exists e s^{\prime} . e s=e s^{\prime} @[e] \wedge e \notin$ set $e s^{\prime}$
using $\operatorname{assms}(3,4)$
proof (induction es arbitrary : v1)
case (Nil v1) thus ?case by simp
next
case (Cons é es v1)
hence subpath $g\left(t g t e^{\prime}\right)$ es v2 subs
and $e=e^{\prime} \vee\left(e \neq e^{\prime} \wedge e \in\right.$ 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 $\operatorname{assms}(1,2)]$ 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 \not\inGraph.vertices g
    assumes v\not= tgt e
    assumes subpath (add-edge g e) v es (tgt e) subs
    shows e\in 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 }\not=[]\mathrm{ using assms(2,3) by (auto simp add : Nil-sp)
    then obtain es' e' where es =es'@ @ [ ] by (simp add : neq-Nil-conv2) blast
    moreover
    hence e}\mp@subsup{e}{}{\prime}\in\mathrm{ 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\rangle\langlees=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 ::
    \(\left({ }^{\prime} v,{ }^{\prime} x\right)\) rgraph-scheme \(\Rightarrow{ }^{\prime} v\) edge \(\Rightarrow\left({ }^{\prime} v,{ }^{\prime} x\right)\) rgraph-scheme \(\Rightarrow\) bool
where
    extends \(g\) e \(g^{\prime} \equiv\) src \(e \in\) Graph.vertices \(g\)
                \(\wedge\) tgt \(e \notin\) Graph.vertices \(g\)
                \(\wedge g^{\prime}=(\) 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 \(g^{\prime}\)
    shows out-edges \(g^{\prime}(\) tgt \(e)=\{ \}\)
using assms unfolding vertices-def image-def by force
```

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

```
lemma (in sub-rel-of)
```

    assumes extends \(g\) e \(g^{\prime}\)
    shows sub-rel-of \(g^{\prime}\) 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 $g^{\prime}$
assumes Graph.subpath g v1 es v2
shows Graph.subpath $g^{\prime}$ v1 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 root g' = root g using assms(2) by simp
    assume v G Graph.vertices g'
    hence v\inGraph.vertices g V v= tgt e
    using assms(2) by (auto simp add : vertices-def)
    thus }\exists\mathrm{ !es. path g' es v
    proof (elim disjE, goal-cases)
    case 1
    then obtain es
    where Graph.path g es v
    and }\foralle\mp@subsup{s}{}{\prime}\mathrm{ . Graph.path ges'v}\longrightarrowes'=e
    using assms(1) unfolding Ex1-def is-tree-def by blast
    hence Graph.path g' es v
    using assms(2) sp-in-extends[OF assms(2)]
    by (subst «root g' = root g`)
    moreover
    have }\foralle\mp@subsup{s}{}{\prime}\mathrm{ . Graph.path g}\mp@subsup{g}{}{\prime}e\mp@subsup{s}{}{\prime}v\longrightarrowe\mp@subsup{s}{}{\prime}=e
    proof (intro allI impI)
        fix es'
            assume Graph.path g' es' v
            thus es' = es
            proof (case-tac e E set es', goal-cases)
            case 1
```

```
    then obtain es"
    where es' = es" @ [e]
    and e\not\in set es"
    using <Graph.path g' es' v〉
        Graph.sp-through-de-decomp[OF extends-tgt-out-edges[OF assms(2)]]
    by blast
    hence v=tgt e
    using <Graph.path g' es'v〉
    by (simp add : Graph.sp-append-one)
        thus ?thesis
        using assms(2)
            Graph.lst-of-sp-is-vert[OF〈Graph.path g es v>]
        by simp
    next
    case 2 thus ?thesis
    using assms
            \forall es'. Graph.path ges'v\longrightarrowes' = es\rangle\langleGraph.path g' es'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 Graph.path g es (src e)
    and }\foralle\mp@subsup{s}{}{\prime}\mathrm{ . Graph.path ges'(srce) }\longrightarrowe\mp@subsup{s}{}{\prime}=e
    using assms(1,2) unfolding is-tree-def by blast
    hence Graph.path g' es (src e)
    using sp-in-extends[OF assms(2)]
    by (subst <root g' = root g>)
    hence Graph.path g' (es @ [e]) (tgt e)
    using assms(2) by (auto simp add : Graph.sp-append-one)
    moreover
    have }\foralle\mp@subsup{s}{}{\prime}\mathrm{ . Graph.path g' es'(tgt e) }\longrightarrowe\mp@subsup{s}{}{\prime}=es@ @ [e
    proof (intro allI impI)
        fix es'
```

```
assume Graph.path \(g^{\prime} e s^{\prime}(\) tgt e)
moreover
hence \(e \in\) set es \({ }^{\prime}\)
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 out-edges \(g^{\prime}(\) tgt e) \(=\{ \}\)
using assms
by (intro extends-tgt-out-edges)
ultimately
have \(\exists e s^{\prime \prime} . e s^{\prime}=e s^{\prime \prime} @[e] \wedge e \notin\) set \(e s^{\prime \prime}\)
by (elim Graph.sp-through-de-decomp)
then obtain \(e s^{\prime \prime}\)
where \(e s^{\prime}=e s^{\prime \prime} @[e]\)
and \(e \notin\) set es \({ }^{\prime \prime}\)
by blast
hence Graph.path \(g^{\prime}\) es \({ }^{\prime \prime}(\operatorname{src} e)\)
using 〈Graph.path \(g^{\prime}\) es \({ }^{\prime}\) (tgt e)〉
by (auto simp add: Graph.sp-append-one)
hence Graph.path \(g\) es \({ }^{\prime \prime}(\operatorname{src} e)\)
using \(\operatorname{assms}(2)\left\langle e \notin\right.\) set es \(\left.{ }^{\prime \prime}\right\rangle\)
by (auto simp add : Graph.subpath-def vertices-def)
hence \(e s^{\prime \prime}=e s\)
using 〈 \(\forall a s^{\prime}\). Graph.path \(\left.g a s^{\prime}(\operatorname{src} e) \longrightarrow a s^{\prime}=e s\right\rangle\)
by \(\operatorname{simp}\)
thus \(e s^{\prime}=e s @[e] \mathbf{u s i n g}\left\langle e s^{\prime}=e s^{\prime \prime} @[e]\right\rangle\) 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-subs :
    assumes extends g a g}\mp@subsup{g}{}{\prime
    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 subpaths 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 $g^{\prime}$
assumes $v 1 \in$ Graph.vertices $g$
assumes $v 2 \in$ Graph.vertices $g$
assumes subpath $g^{\prime}$ v1 es v2 subs
shows subpath $g$ v1 es v2 subs
proof -
have $e \notin$ set es
proof (intro notI)
assume $e \in$ set es
have $v 2=$ tgt $e$
proof -
have tgt $e \notin$ subsumees subs using sub-rel-of assms(1) by fast
moreover
have out-edges $g^{\prime}($ tgt e) $=\{ \} \mathbf{u s i n g}$ assms(1) by (rule extends-tgt-out-edges)
ultimately
have $\exists e s^{\prime} . e s=e s^{\prime} @[e] \wedge e \notin$ set $e s^{\prime}$
using assms(4) $\langle e \in$ set es $\rangle$
by (intro sp-through-de-decomp)
then obtain $e s^{\prime}$ where $e s=e s^{\prime} @[e] e \notin$ set es' by blast
hence tgt $e=v 2 \vee($ tgt $e, v 2) \in s u b s^{+}$
using assms(4) by (simp add: sp-append-one)
thus ? thesis using «tgt e $\notin$ subsumees subs〉 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 \(g^{\prime}\)
    assumes subpath \(g^{\prime}(t g t e)\) es \(v\) subs
    shows es = []
using sub-rel-of assms
    extends-tgt-out-edges
    sp-from-de-empty[of tgt e subs \(g^{\prime}\) 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 \(g^{\prime}\)
    assumes subpath \(g^{\prime} v\) es (tgt e) subs
    assumes \(v \neq t g t e\)
    shows \(\exists e s^{\prime}\). es \(=e s^{\prime} @[e] \wedge e \notin\) set \(e s^{\prime} \wedge\) subpath \(g v e s^{\prime}(\) src e) subs
proof -
    obtain \(e s^{\prime}\) where \(e s=e s^{\prime} @[e]\) and \(e \notin\) set \(e s^{\prime}\)
    using sub-rel-of \(\operatorname{assms}(1,2,3)\)
        extends-tgt-out-edges[OF assms(1)]
        sp-through-de-decomp[of e subs \(g^{\prime} v\) es tgt e]
        sp-ends-in-tgt-imp-mem[of e ves]
    by blast
    moreover
    have subpath \(g v e s^{\prime}(s r c e)\) subs
    proof -
    have \(v \in\) Graph.vertices \(g\)
    using assms \((1,3)\) fst-of-sp-is-vert[OF \(\operatorname{assms(2)}\) ]
    by (auto simp add : vertices-def)
    moreover
    have SubRel.subpath \(g^{\prime} v e s^{\prime}(s r c e)\) subs
```

```
    using assms(2)<es =es' @ [e]> by (simp add : sp-append-one)
    ultimately
    show ?thesis
    using assms(1) sub-rel-of <e \not\in set es'〉
    unfolding subpath-def by (auto simp add : sub-rel-of-def)
qed
ultimately
    show ?thesis by blast
qed
end
theory SubExt
imports SubRel
begin
```


## 11 Extending subsomption relations

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

### 11.1 Definition

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

- the two vertices involved are distinct,
- they are occurrences of the same vertex,
- they are both vertices of the graph,
- the subsumee must not already be a subsumer or a subsumee,
- the subsumer must not be a subsumee (but it can already be a subsumer),
- the subsumee must have no out-edges.

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

```
abbreviation extends ::
```



```
where
    extends g subs sub subs' }\equiv
        subsumee sub }\not=\mathrm{ subsumer sub
    ^fst (subsumee sub) = fst (subsumer sub)
    ^subsumee sub }\in\mathrm{ Graph.vertices g
    ^ subsumee sub # subsumers subs
    ^ subsumee sub # subsumees subs
    ^ subsumer sub }\in\mathrm{ Graph.vertices g
    ^ subsumer sub # subsumees subs
    \out-edges g (subsumee sub) = {}
    ^subs' = subs \cup{sub})
```


### 11.2 Properties of extensions

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

Extending the sub-relation of a graph yields a new sub-relation for this graph.
lemma (in sub-rel-of)
assumes extends $g$ subs sub subs'
shows sub-rel-of $g$ subs ${ }^{\prime}$
using assms sub-rel-of unfolding sub-rel-of-def by force
Extending a well-formed relation yields a well-formed relation.

```
lemma (in wf-sub-rel) extends-imp-wf-sub-rel :
    assumes extends g subs sub subs'
    shows wf-sub-rel subs'
unfolding wf-sub-rel-def
proof (intro conjI, goal-cases)
    case 1 show ?case using wf-sub-rel assms by auto
next
    case 2 show ?case
    unfolding Ball-def
    proof (intro allI impI)
```

```
    fix v
    assume v \in subsumees subs'
    hence v= subsumee sub }\veev\in\mathrm{ subsumees subs using assms by auto
    thus }\exists!\mp@subsup{v}{}{\prime}.(v,\mp@subsup{v}{}{\prime})\insubs
    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) \in subs' using 1 assms by simp
        next
            have v\not\in subsumees subs using assms 1 by auto
            thus }\forall\mp@subsup{v}{}{\prime}.(v,\mp@subsup{v}{}{\prime})\in\mathrm{ subs' }\longrightarrow\mp@subsup{v}{}{\prime}=\mathrm{ subsumer sub
            using assms by auto force
        qed
    next
    case 2
    then obtain }\mp@subsup{v}{}{\prime}\mathrm{ where (v,v})\in\mathrm{ subs by auto
    hence v}\not=\mathrm{ subsumee sub
    using assms unfolding subsumees-conv by (force simp del : split-paired-All
split-paired-Ex)
    show ?thesis
    using assms
         v}\not=\mathrm{ subsumee sub>
        <(v,v})\in\mathrm{ 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 \({ }^{\prime}\)
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 subpaths 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 \((v 1, v 2)\) subs'
    assumes subpath \(g\) v1 es \(v\) subs \({ }^{\prime}\)
    shows es \(=[] \vee\) subpath \(g\) v2 es \(v\) subs \(^{\prime}\)
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 \(\operatorname{simp}\)
```

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 $s u b$ is a member of subs.

```
fun uses-sub ::
    \((' v \times n a t) \Rightarrow\left({ }^{\prime} v \times n a t\right)\) edge list \(\Rightarrow\left({ }^{\prime} v \times n a t\right) \Rightarrow\left(\left({ }^{\prime} v \times n a t\right) \times\left({ }^{\prime} v \times n a t\right)\right) \Rightarrow\)
bool
where
    uses-sub v1 [] v2 sub \(=(v 1 \neq v 2 \wedge\) sub \(=(v 1, v 2))\)
\(\mid\) uses-sub v1 \((e \# e s)\) v2 sub \(=(v 1 \neq\) src \(e \wedge\) sub \(=(v 1\), src e \() \vee\) uses-sub (tgt e)
es v2 sub)
```

In order for a sequence es using the subsumption sub to be consistent wrt. to a subsumption relation subs, the subsumption sub must occur in the transitive closure of subs.
lemma
assumes uses-sub v1 es v2 sub
assumes ces v1 es v2 subs
shows sub $\in$ subs $^{+}$
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 }\in\mathrm{ 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 \(\left(v, v^{\prime}\right)\) subs \({ }^{\prime}\)
    assumes subpath \(g\) v1 es v2 subs
    shows \(\neg\) uses-sub v1 es v2 \(\left(v, v^{\prime}\right)\)
proof (intro notI)
    assume uses-sub v1 es v2 ( \(v, v^{\prime}\) )
    moreover
    have ces v1 es v2 subs using assms(2) by (simp add : subpath-def)
    ultimately
    have \(\left(v, v^{\prime}\right) \in \operatorname{subs}^{+}\)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 $v 1$ to $v 2$ after the extension. Then, the empty sequence is a sub-path leading from $v 1$ to $v 2$ in the graph before the extension if and only if $(v 1, v \mathscr{2})$ is not the new subsumption.

```
lemma (in wf-sub-rel-of) sp-Nil-in-extends-imp :
    assumes extends g subs (v,v') subs'
    assumes subpath g v1 [] v2 subs'
    shows subpath g v1[] v2 subs \longleftrightarrow(v1 \not=v\veev2 f= 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\inGraph.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 $\left(v, v^{\prime}\right)$ subs ${ }^{\prime}$
assumes subpath g v1 es v2 subs'
assumes $\neg$ uses-sub v1 es v2 ( $v, v^{\prime}$ )
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,\mp@subsup{v}{}{\prime})\mathrm{ subs'
    assumes subpath g v1 es v2 subs'
    assumes v1 \not=v
    shows subpath g v1 es v2 subs \vee (\exists es1 es2. es =es1@ es2
                                    ^es1 = []
                                    ^ subpath g v1 es1 v subs
                                    ^ 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\mp@subsup{\operatorname{subs}}{}{\prime
    and edge }\in\mathrm{ 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 }\times\mathrm{ nat) rgraph
    black :: ('vert,'var,'d)lts
    subs ::'vert sub-rel-t
    init-conf :: ('var,'d) conf
    confs :: ('vert }\times\mathrm{ nat) }=>('var,'d) conf
    marked :: ('vert }\times\mathrm{ nat) }=>\mathrm{ bool
    strengthenings :: ('vert }\times\mathrm{ nat) }=>('var,'d) bexp
```

We call red vertices the set of vertices of the red graph.
abbreviation red-vertices ::
('vert,'var, ${ }^{\prime}$ d,'x) pre-RedBlack-scheme $\Rightarrow\left({ }^{\prime} v e r t \times n a t\right)$ set
where
red-vertices lts $\equiv$ 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 $\times$ nat) edge $\Rightarrow$ 'vert edge
where
ui-edge $e \equiv 0$ src $=f s t($ src e $), \operatorname{tgt}=f s t($ tgt e) D
We extend this idea to sequences of edges.
abbreviation ui-es ::
('vert $\times$ nat) edge list $\Rightarrow$ 'vert edge list
where
ui-es es $\equiv$ 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 \(\quad: e \in\) edges (red prb) \(\Longrightarrow\) ui-edge \(e \in\) edges (black prb)
begin
    lemmas wf-pre-RedBlack \(=\) red-wf black-wf consistent-roots ui-re-are-be
end
```

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

- the path predicate of its initial configuration contains a finite number of constraints,
- each of these constraints contains a finite number of variables,
- its black part is finite (cf. definition of finite-lts.).
locale finite-RedBlack $=$ pre-RedBlack +
assumes finite-init-pred : finite (pred (init-conf prb))
assumes finite-init-pred-symvars : $\forall e \in$ pred (init-conf prb). finite (Bexp.vars
e)
assumes finite-lts : finite-lts (black prb)
begin
lemmas finite-RedBlack $=$ finite-init-pred finite-init-pred-symvars finite-lts
end


### 12.2 Extensions of Red-Black Graphs

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

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

### 12.2.1 Extension by symbolic execution

The core abstract operation of symbolic execution: take a black edge and turn it red, by symbolic execution of its label. In the following abbreviation, $r e$ 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 ::
    ('vert,'var, 'd) pre-RedBlack
    \(\Rightarrow\) ('vert \(\times\) nat) edge
    \(\Rightarrow\left(' v a r,{ }^{\prime} d\right)\) conf
    \(\Rightarrow\left(' v e r t,{ }^{\prime}\right.\) var, 'd) pre-RedBlack \(\Rightarrow\) bool
where
    se-extends prb re c prb \({ }^{\prime} \equiv\)
        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)) c
    \(\wedge p r b^{\prime}=0\) red \(=\) red \(p r b^{\prime}\),
            black \(=\) black prb,
            subs \(\quad=\) subs prb,
            init-conf \(=\) init-conf prb,
            confs \(=(\) confs prb) (tgt re \(:=c)\),
            marked \(=(\) marked prb) (tgt re \(:=\) marked prb (src re) \()\),
```

$$
\text { strengthenings }=\text { strengthenings prb D }
$$

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 \(\Rightarrow\left({ }^{\prime} v e r t\right.\), '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 \({ }^{\prime}=\left(\right.\) confs prb) \(\left(\right.\) tgt re \(:=\) confs prb' \({ }^{\prime}\) tgt re \()\) )
    \(\wedge\) marked \(p r b^{\prime}=(\) 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 $r v$ has been proved unsatisfiable.

```
abbreviation mark-extends ::
    ('vert,'var,'d) pre-RedBlack \(\Rightarrow\) ('vert \(\times\) nat \() \Rightarrow\left({ }^{\prime} v e r t, ' v a r, ' d\right)\) pre-RedBlack \(\Rightarrow\)
bool
where
    mark-extends prb rv prb' \(\equiv\)
        \(r v \in\) red-vertices prb
    \(\wedge\) out-edges (red prb) rv \(=\{ \}\)
    \(\wedge r v \notin\) subsumees (subs prb)
    \(\wedge r v \notin\) subsumers (subs prb)
    \(\wedge \neg\) sat (confs prb rv)
    \(\wedge p r b^{\prime}=0\) red \(=\) red \(p r b\),
            black = black prb,
            subs \(=\) subs prb,
            init-conf \(=\) init-conf prb,
            confs \(=\) confs prb,
            marked \(=\left(\lambda r v^{\prime}\right.\). if \(r v^{\prime}=r v\) then True else marked \(\left.p r b r v^{\prime}\right)\),
```

```
strengthenings = strengthenings prb,
... = more prb D
```


### 12.2.3 Extension by subsumption

The abstract operation of introducing a subsumption link.

```
abbreviation subsum-extends ::
    ('vert,'var,'d) pre-RedBlack = 'vert sub-t => ('vert,'var,'d) pre-RedBlack => bool
where
    subsum-extends prb sub prb' \equiv
        SubExt.extends (red prb) (subs prb) sub (subs prb)
    \neg marked prb (subsumer sub)
    \wedge marked prb (subsumee sub)
    confs prb (subsumee sub) \sqsubseteq confs prb (subsumer sub)
    \wedgerb}\mp@subsup{}{\prime}{\prime}=\ red = red prb
        black = black prb,
        subs = insert sub (subs prb),
        init-conf = init-conf prb,
    confs = confs prb,
    marked = marked prb,
    strengthenings = strengthenings prb,
    ... = more prb D
```


### 12.2.4 Extension by abstraction

This operation replaces the configuration of a red vertex $r v$ 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 $r v$ and must entail its safeguard condition, if any.

```
abbreviation abstract-extends ::
    ('vert,'var,'d) pre-RedBlack
    \(\Rightarrow\) ('vert \(\times\) nat)
    \(\Rightarrow\left(' v a r,{ }^{\prime} d\right)\) conf
    \(\Rightarrow\left(' v e r t, ' v a r,{ }^{\prime} d\right)\) pre-RedBlack
    \(\Rightarrow\) bool
where
    abstract-extends prb rv ca prb \(^{\prime} \equiv\)
        \(r v \in\) red-vertices prb
    \(\wedge \neg\) marked prb rv
    \(\wedge\) out-edges (red prb) rv \(=\{ \}\)
    \(\wedge r v \notin\) subsumees (subs prb)
    \(\wedge\) abstract (confs prb rv) \(c_{a}\)
    \(\wedge c_{a} \models_{c}\) (strengthenings prb rv)
```

```
^finite (pred ca)
\wedge (\foralle\in pred ca. finite (vars e))
\wedgerb' = \ red = red prb,
    black = black prb,
    subs = subs prb,
    init-conf = init-conf prb,
    confs}=(\mathrm{ confs prb)(rv:= ca})
    marked = marked prb,
    strengthenings = strengthenings prb,
    ... = more prb )
```


### 12.2.5 Extension by strengthening

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

```
abbreviation strengthen-extends ::
    ('vert,'var,'d) pre-RedBlack
    \(\Rightarrow\) ('vert \(\times\) nat \()\)
    \(\Rightarrow(' v a r, ' d)\) bexp
    \(\Rightarrow\) ('vert,'var,'d) pre-RedBlack
    \(\Rightarrow\) bool
where
    strengthen-extends prb rv e prb' \(\equiv\)
        \(r v \in\) red-vertices prb
    \(\wedge r v \notin\) subsumees (subs prb)
    \(\wedge\) confs prb rv \(\models_{c}\) e
    \(\wedge p r b^{\prime}=0\) red \(=\) red \(p r b\),
        black = black prb,
        subs \(=\) subs \(p r b\),
        init-conf \(=\) init-conf prb,
        confs \(=\) confs prb,
        marked \(=\) marked prb,
        strengthenings \(=(\) strengthenings prb \()(r v:=(\lambda \sigma\). (strengthenings prb
rv) \(\sigma \wedge e \sigma)\) ),
    \(\ldots\) = more prb \()\)
```


### 12.3 Building Red-Black Graphs using Extensions

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

```
    ('vert,'var,'d) pre-RedBlack \(\Rightarrow\) bool
where
    base :
        fst \((\) root \((\) red prb \())=\) init \((\) black prb \() \quad \Longrightarrow\)
        edges \((\) red prb \()=\{ \} \quad \Longrightarrow\)
        subs prb \(=\{ \} \quad \Longrightarrow\)
        \((\) confs prb) \((\) root \((\) red prb \())=\) init-conf prb \(\Longrightarrow\)
        marked prb \(=(\lambda r v\). False \() \quad \Longrightarrow\)
        strengthenings prb \(=(\lambda r v .(\lambda \sigma\). True \()) \Longrightarrow\) RedBlack prb
| se-step :
    RedBlack prb \(\quad \Longrightarrow\)
        se-extends prb re \(p^{\prime}\) prb \(^{\prime} \quad \Longrightarrow\) RedBlack prb \({ }^{\prime}\)
| mark-step :
    RedBlack prb
        mark-extends prb rv prb' \(\quad \Longrightarrow\) RedBlack prb'
| subsum-step :
    RedBlack prb
        subsum-extends prb sub prb' \(\quad \Longrightarrow\) RedBlack prb'
| abstract-step :
    RedBlack prb
        \(\Longrightarrow\) RedBlack prb'
    abstract-extends prb rv \(c_{a}\) prb \(^{\prime} \quad \Longrightarrow\) RedBlack prb'
| strengthen-step :
    RedBlack prb \(\quad \Longrightarrow\)
        strengthen-extends prb rv e prb' \(\quad \Longrightarrow\) RedBlack prb'
```


### 12.4 Properties of Red-Black-Graphs

### 12.4.1 Invariants of the Red-Black Graphs

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

## lemma

assumes RedBlack prb
shows loop-free (red prb)
using assms by (induct prb) auto
A red edge can not lead to the (red) root.
lemma
assumes RedBlack prb
assumes re $\in$ edges (red prb)
shows tgt re $\neq$ root (red prb)
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-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-edge'out-edges (red prb) rv\subseteqout-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 \(\quad\) 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-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-lts (black prb)
    shows wf-pre-RedBlack prb
proof -
    have wf-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 \(r v \in\) red-vertices prb
    shows fst \(r v \in\) 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 :
```

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

```
        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 $\in$ subs prb
shows $\quad$ fst (subsumee sub) $=f s t$ (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 $=\operatorname{src} r e \vee(r v 1$, src re $) \in(s u b s p r b)^{+}$
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 $\operatorname{assms(1)]\text {,ofrv1}}$
src re]
subs-to-same-BL[OF $\operatorname{assms}(1)$, of (rv1,src re)]
by auto
moreover
have Graph.ces (tgt (ui-edge re)) (ui-es res) (fst rv2)
using $\operatorname{assms}(1) \operatorname{Cons}(1) 2$ by $\operatorname{simp}$

## 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\in red-vertices prb
    shows }\foralle\in\mathrm{ pred (confs prb rv). finite (Bexp.vars e)
using assms
proof (induct prb arbitrary : rv)
    case base thus ?case by (simp add : vertices-def finite-RedBlack-def)
next
    case (se-step prb re c' prb')
    hence rv \in red-vertices prb \veerv = tgt re by (auto simp add : vertices-def)
    thus ?case
    proof (elim disjE)
        assume rv\in 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 = tgt re
    moreover
    have finite-label (labelling (black prb) (ui-edge re))
    using se-step by (auto simp add : finite-RedBlack-def)
    moreover
    have }\foralle\in\mathrm{ pred (confs prb (src re)). finite (Bexp.vars e)
        using se-step se-step(2)[of src re] unfolding finite-RedBlack-def
    by (elim conjE) auto
    moreover
    have se (confs prb (src re)) (labelling (black prb) (ui-edge re)) c'
        using se-step by auto
        ultimately
    show ?thesis using se-step se-preserves-finiteness1 by fastforce
    qed
next
    case mark-step thus ?case by (simp add : finite-RedBlack-def)
next
    case subsum-step thus ?case by (simp add : finite-RedBlack-def)
next
    case abstract-step thus ?case by (auto simp add: finite-RedBlack-def)
next
    case strengthen-step thus ?case by (simp add : finite-RedBlack-def)
qed
```

The path predicate of a configuration associated to a red location of a redblack graph contains a finite number of constraints.
lemma finite-pred:
assumes RedBlack prb
assumes finite-RedBlack prb
assumes $r v \in$ red-vertices prb
shows finite (pred (confs prb rv))
using assms
proof (induct prb arbitrary : rv)
case base thus ?case by (simp add : vertices-def finite-RedBlack-def)
next
case (se-step prb re $\left.c^{\prime} p r b^{\prime}\right)$
hence $r v \in$ red-vertices prb $\vee r v=$ tgt re by (auto simp add : vertices-def)
thus ?case

```
    proof (elim disjE, goal-cases)
        case 1 thus ?thesis
            using se-step(2)[of rv] se-step(3,4)
            by (auto simp add : finite-RedBlack-def)
    next
        case 2
        moreover
        hence src re \in 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 \(r v\) 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 \(r v\).
lemma (in finite-RedBlack) ex-se-succ :
assumes RedBlack prb
assumes \(r v \in\) red-vertices prb
shows \(\exists c^{\prime}\). se (confs prb rv) l \(c^{\prime}\)
using finite-RedBlack assms
finite-imp-ex-se-succ[of confs prb rv]
finite-pred[of prbrv]
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 \(r v \in\) red-vertices prb
assumes finite-labels ls
```

```
    shows \(\exists c^{\prime}\). se-star (confs prb rv) ls \(c^{\prime}\)
using finite-RedBlack assms
    finite-imp-ex-se-star-succ[of confs prb rv ls]
    finite-pred \([O F \operatorname{assms}(1)\), 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. se-star
(confs prb rv1)
(trace (ui-es res) (labelling (black 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 $r l$ is update-able.
lemma (in finite-RedBlack)
assumes RedBlack prb
assumes $r v \in$ red-vertices prb
shows updatable (confs prb rv)
using finite-RedBlack assms
finite-conj[OF finite-pred $[$ OF $\operatorname{assms}(1)]$
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 \(c^{\prime}\) prb')
    moreover
```

hence sub $\in$ subs prb by auto
hence subsumee sub $\in$ red-vertices prb
and subsumer sub $\in$ red-vertices prb
using se-step(1) subs-sub-rel-of
unfolding sub-rel-of-def by fast+
moreover
have tgt re $\notin$ red-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 $c_{a}$ prb')
hence $r v \neq$ subsumee sub by auto
show ?case
proof (case-tac rv $=$ subsumer sub)
assume $r v=$ 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 $r v \neq$ subsumer sub thus ?thesis using abstract-step $\langle r v \neq$ subsumee sub> 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 subpath (red prb) rv1 [] rv2 (subs prb) =
    (rv1 \in red-vertices prb ^(rv1 = rv2 \vee (rv1,rv2) }\in(\mathrm{ subs prb)))
using assms subs-wf-sub-rel subs-sub-rel-of wf-sub-rel.Nil-sp by fast
lemma rb-sp-one:
    assumes RedBlack prb
    shows subpath (red prb) rv1 [re] rv2 (subs prb) =
        ( sub-rel-of (red prb) (subs prb)
        \wedge(rv1 = src re \vee (rv1, src re) \in(subs prb))
        \wedgee\inedges (red prb) ^(tgt re = rv2 \vee (tgt re, rv2) ) (subs prb)))
using assms subs-wf-sub-rel-of wf-sub-rel-of.sp-one by fast
lemma rb-sp-Cons:
    assumes RedBlack prb
    shows subpath (red prb) rv1 (re # res) rv2 (subs prb)=
        ( sub-rel-of (red prb) (subs prb)
        \wedge(rv1 = src re \vee (rv1, src re) \in subs prb)
        \wedge re\inedges (red prb)
        \wedge subpath (red prb) (tgt re) res rv2 (subs prb))
using assms subs-wf-sub-rel-of wf-sub-rel-of.sp-Cons by fast
lemma rb-sp-append-one :
    assumes RedBlack prb
    shows subpath (red prb) rv1 (res @ [re]) rv2 (subs prb)=
        ( subpath (red prb) rv1 res (src re) (subs prb)
        ^re\inedges (red prb)
    \wedge (tgt re = rv2 \vee (tgt re, rv2 ) \in 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 $c^{\prime}$ prb' rv1 res rv2 c)
have rv1 $\in$ red-vertices prb ${ }^{\prime}$
and rv2 $\in$ red-vertices $\mathrm{prb}^{\prime}$
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-vertices prb $\wedge r v 1 \neq t g t r e \vee r v 1=$ tgt re
and $\quad$ rv2 $\in$ red-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]
                ^re & set res'
                ^ 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 # 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 }\mp@subsup{c}{}{\prime}\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-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' res rv2]
            rb-Nil-sp[OF RedBlack.se-step[OF se-step(1,3)], of rv1 rv2]
        by auto
        hence rv1 \in subsumees (subs prb) using se-step(3) by force
        thus ?thesis
        using se-step <rv1 = tgt re> subs-sub-rel-of[OF se-step(1)]
        by (auto simp add : sub-rel-of-def)
    qed
    ultimately
    show ?thesis by simp
next
    case 4
    moreover
    hence res = []
        using se-step
            sub-rel-of.sp-from-tgt-in-extends-is-Nil
            [OF subs-sub-rel-of[OF se-step(1)], of re red prb' res rv2]
        by auto
    ultimately
    show ?thesis using se-step by (simp add : subsums-refl)
qed
```

next
case (mark-step prb rv prb') thus ?case by simp
next
case (subsum-step prb sub prb' rv1 res rv2 c)
have $R B^{\prime}$ : RedBlack prb' by (rule RedBlack.subsum-step $[$ OF subsum-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 \(\operatorname{simp}\)
    thus ?thesis
    proof (elim disjE)
    assume res \(=[]\)
    hence \(r v 1=r v 2 \vee(r v 1, r v 2) \in\left(s u b s p r b^{\prime}\right)\)
            using subsum-step rb-Nil-sp[OF RB] by fast
    thus ?thesis
    proof (elim disjE)
        assume rv1 \(=r v 2\)
        thus ?thesis
            using subsum-step (5) 〈res = []〉
            by (simp add : subsums-refl)
    next
```

        assume (rv1, rv2) \(\in\left(s u b s p r b^{\prime}\right)\)
        thus ?thesis
            using subsum-step \((5)\langle\) res \(=[]\rangle\)
                sub-subsumed \(\left[O F R B^{\prime}\right.\), of (rv1,rv2)]
            by \(\operatorname{simp}\)
    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) & 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)) \(c^{\prime \prime}\)
using subsum-step ( \(6-8\) )
    〈subpath (red prb') (src re) [re] (tgt re) (subs prb') >
    \(R B^{\prime}\) 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 \(c \sqsubseteq c^{\prime \prime}\)
proof -
    have \(c^{\prime} \sqsubseteq\) confs \(\mathrm{prb}^{\prime}\) (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 \(c^{\prime \prime} \sqsubseteq\) confs prb' \({ }^{\prime}\) 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 \(r e \in\) 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 \(\quad r e \in\) 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) \in (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 \(\neq\) subsumee sub
    hence subpath (red prb) rv1 res rv2 (subs prb) \(\vee\)
            ( \(\exists\) res1 res2. res \(=\) res1 @ res2
            \(\wedge\) res \(1 \neq[]\)
            \(\wedge\) subpath (red prb) rv1 res1 (subsumee sub) (subs prb)
            \(\wedge\) 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}\sqsubseteqc
proof -
    have c1 \sqsubseteq 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 $c 2 \sqsubseteq$ confs prb' rv2
using <subpath (red prb') (subsumee sub) res2 rv2 (subs prb') >
〈se-star (confs prb' (subsumee sub)) t-res2 c2〉
unfolding $t$-res之-def
proof (induct res2 arbitrary : rv2 c2 rule : rev-induct, goal-cases)
case (1 rv2 c2)
hence subsumee sub $=$ rv2 $\vee($ subsumee sub, rv2 $) \in$ 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) R\mp@subsup{B}{}{\prime}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 \sqsubseteqc4
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 c4}\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) ) (subs prb')
            using subsum-step 2 rb-sp-append-one[OF RB', of subsumee sub res2
            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(\mathrm{ subs prb')
        thus ?thesis
            using sub-subsumed RB'
            by fastforce
    qed
```

$r e]$

```
            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 \vee (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 \sqsubseteqconfs prb' rv2
                using abstract-step by (simp add : abstract-def)
        moreover
        have c}\sqsubseteqconfs prb rv2
            using abstract-step 2 by auto
        ultimately
        show ?thesis using subsums-trans by fast
        next
        assume rv2 }=rv\mathrm{ 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 $r v \in$ 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 $r v \in$ red-vertices $p r b \vee r v=t g t r e$ by (auto simp add : vertices-def)

```
    thus ?case
    proof (elim disjE, goal-cases)
        case 1
        moreover
        hence rv f= 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\not=rv' and (rv,rv') \not\in 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' ca 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 \(r v \in\) 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 $r v \in$ red-vertices $p r b \vee r v=$ tgt re by (auto simp add : vertices-def)
thus ?case
proof (elim disjE, goal-cases)
case 1
moreover
hence $r v \neq$ tgt re using se-step (3) by auto
hence marked prb rv using se-step by auto
ultimately
have $\neg$ sat (confs prb rv) by (rule se-step(2))
thus ?thesis using se-step(3) <rv $\neq$ tgt re〉 by auto
next
case 2
moreover
hence marked prb (src re) using se-step $(3,5)$ by auto
ultimately
have $\neg$ sat (confs prb (src re)) using se-step(2,3) by auto
thus ?thesis using se-step(3) $\langle r v=$ tgt re〉 unsat-imp-se-unsat by (elim conjE)
auto
qed
next
case (mark-step prb rv' prb') thus ?case by (case-tac rv' $=r v$ ) auto
next
case subsum-step thus? ?case by simp
next
case (abstract-step $-r v^{\prime}$-) thus ? case by (case-tac rv $=r v$ ) 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 }\not=\mathrm{ 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 }\not=\mathrm{ 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 e edges (red prb)
    assumes }\neg\mathrm{ marked prb (tgt re)
    shows \neg 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
```

lemma
assumes RedBlack prb
assumes RedBlack prb
assumes subpath (red prb) rv1 res rv2 (subs prb)
assumes subpath (red prb) rv1 res rv2 (subs prb)
assumes \neg marked prb rv2
assumes \neg marked prb rv2
shows \neg marked prb rv1
shows \neg marked prb rv1
using assms
using assms
proof (induct res arbitrary : rv1)
proof (induct res arbitrary : rv1)
case Nil

```
    case Nil
```

```
    hence rv1 = rv2 \vee (rv1,rv2) \in 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\mathrm{ marked prb (tgt re) by simp
        moreover
    have re e edges (red prb) using Cons(3) rb-sp-Cons[OF Cons(2), of rv1 re res
rv2] by fast
        ultimately
        show ?thesis using tgt-not-marked-imp[OF Cons(2)] by fast
    next
        case 2 thus ?thesis using 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 ::
    ('vert, 'var, 'd, 'x) pre-RedBlack-scheme \(\Rightarrow\) ('vert \(\times\) nat) set
where
    fringe \(p r b \equiv\{r v \in\) red-vertices \(p r b\).
    \(r v \notin\) subsumees (subs prb) \(\wedge\)
    \(\neg\) marked prb rv ^
    ui-edge ' out-edges (red prb) rv \(\subset\) out-edges (black prb) (fst rv)\}
```


### 12.7.2 Fringe of an empty red-part

At the beginning of the analysis, i.e. when the red part is empty, the fringe consists of the red root.
lemma fringe-of-empty-red1:
assumes edges (red prb) $=\{ \}$
assumes subs prb $=\{ \}$
assumes marked prb $=(\lambda$ rv. False $)$
assumes out-edges (black prb) $($ fst $($ root $($ red prb) $)) \neq\{ \}$
shows fringe prb $=\{$ root (red prb) $\}$
using assms by (auto simp add: fringe-def vertices-def)

### 12.7.3 Evolution of the fringe after extension

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

```
lemma seE-fringe1 :
    assumes sub-rel-of (red prb) (subs prb)
    assumes se-extends prb re \(c^{\prime} p r b^{\prime}\)
    assumes \(\neg\) marked prb (src re)
    assumes ui-edge ' (out-edges (red prb') (src re)) \(\subset\) out-edges (black prb) (fst (src
re))
    assumes out-edges (black prb) \((\) fst \((\) tgt re \()) \neq\{ \}\)
    shows fringe prb' \(=\) fringe prb \(\cup\{\) tgt re \(\}\)
unfolding set-eq-iff Un-iff singleton-iff
```

```
proof (intro allI iffI, goal-cases)
    case (1 rv)
    moreover
    hence rv \in red-vertices prb \veerv= tgt re
    using assms(2) by (auto simp add : fringe-def vertices-def)
    ultimately
    show ?case using assms(2) by (auto simp add: fringe-def)
next
    case (2rv)
    hence rv\in red-vertices prb' using assms(2) by (auto simp add : fringe-def
vertices-def)
    moreover
    have rv \not\in subsumees (subs prb')
    using 2
    proof (elim disjE)
        assume rv f fringe prb thus ?thesis using assms(2) by (auto simp add :
fringe-def)
    next
        assume rv = tgt re thus ?thesis
        using assms(1,2) unfolding sub-rel-of-def by force
    qed
    moreover
    have ui-edge '(out-edges (red prb') rv) \subset out-edges (black prb') (fst rv)
    using 2
    proof (elim disjE)
        assume rv\in fringe prb
        thus ?thesis
        proof (case-tac rv = src re)
            assume rv = src re thus ?thesis using assms(2,4) by auto
        next
            assume rv \not= src re thus ?thesis
            using assms(2) <rv \in fringe prb>
            by (auto simp add : fringe-def)
        qed
    next
        assume rv = tgt re thus ?thesis
        using assms(2,5) extends-tgt-out-edges[of re red prb red prb] by (elim conjE)
auto
    qed
```

```
    moreover
    have \neg marked prb' rv
    using 2
    proof (elim disjE, goal-cases)
    case 1
    moreover
    hence rv f= 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:
```

lemma seE-fringe4:
assumes sub-rel-of (red prb) (subs prb)
assumes sub-rel-of (red prb) (subs prb)
assumes se-extends prb re c' prb'
assumes se-extends prb re c' prb'
assumes \neg marked prb (src re)
assumes \neg marked prb (src re)
assumes }\neg(ui-edge'(out-edges (red prb') (src re)) \subset out-edges (black prb) (fs
assumes }\neg(ui-edge'(out-edges (red prb') (src re)) \subset out-edges (black prb) (fs
(src re)))
(src re)))
assumes out-edges (black prb) (fst (tgt re)) = {}
assumes out-edges (black prb) (fst (tgt re)) = {}
shows fringe prb' = fringe prb - {src re} \cup{tgt re}
shows fringe prb' = fringe prb - {src re} \cup{tgt re}
unfolding set-eq-iff Un-iff singleton-iff Diff-iff
unfolding set-eq-iff Un-iff singleton-iff Diff-iff
proof (intro allI iffI, goal-cases)
proof (intro allI iffI, goal-cases)
case (1 rv)
case (1 rv)
hence rv\in red-vertices prb \veerv= tgt re
hence rv\in red-vertices prb \veerv= tgt re
and rv\not= src re
and rv\not= src re
using assms(2,3,4,5) by (auto simp add : fringe-def vertices-def)
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)
with 1 show ?case using assms(2) by (auto simp add : fringe-def)
next
next
case (2 rv)

```
    case (2 rv)
```

```
    hence rv\in red-vertices prb' using assms(2) by (auto simp add : fringe-def
vertices-def)
    moreover
    have rv & subsumees (subs prb')
    using 2
    proof (elim disjE)
    assume rv\in fringe prb ^rv \not= src re
    thus ?thesis using assms(2) by (auto simp add: fringe-def)
    next
        assume rv = tgt re thus ?thesis
        using assms(1,2) unfolding sub-rel-of-def by fastforce
    qed
    moreover
    have ui-edge '(out-edges (red prb') rv) \subset out-edges (black prb') (fst rv)
    using 2
    proof (elim disjE)
    assume rv\in fringe prb ^rv\not= src re thus ?thesis
    using assms(2) by (auto simp add: fringe-def)
    next
        assume rv = tgt re thus ?thesis
        using assms(2,5) extends-tgt-out-edges[of re red prb red prb] by (elim conjE)
auto
    qed
    moreover
    have \neg marked prb' rv
    using 2
    proof (elim disjE, goal-cases)
    case 1
    moreover
    hence rv}\not=\mathrm{ tgt re using assms by (auto simp add : fringe-def)
    ultimately
    show ?thesis
    using assms 1 by (auto simp add : fringe-def)
next
    case 2 thus ?thesis using assms by auto
    qed
    ultimately
    show ?case by (simp add : fringe-def)
qed
```

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

```
lemma seE-fringe2 :
    assumes se-extends prb re c prb'
    assumes marked prb (src re)
    shows fringe prb' = fringe prb
unfolding set-eq-iff Un-iff singleton-iff
proof (intro allI iffI, goal-cases)
    case (1 rv)
    thus ?case
    unfolding fringe-def mem-Collect-eq
    using assms
    proof (intro conjI, goal-cases)
        case 1 thus?case by (auto simp add: fringe-def vertices-def)
    next
        case 2 thus ?case by auto
    next
        case 3
        moreover
        hence rv f 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 \(c^{\prime} p r b^{\prime}\)
    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 \(\mathrm{prb}^{\prime}=\) fringe prb
unfolding set-eq-iff Un-iff singleton-iff
proof (intro allI iffI, goal-cases)
    case (1rv)
    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 (2rv)
    moreover
    hence \(r v \in\) red-vertices \(p r b^{\prime}\)
    and \(\quad r v \neq t g t r e\)
    using \(\operatorname{assms}(1)\) by (auto simp add : fringe-def vertices-def)
    moreover
    have ui-edge ' (out-edges (red prb') rv) \(\subset\) out-edges (black prb) \((f s t r v)\)
    proof (case-tac rv \(=\operatorname{src} r e\) )
    assume \(r v=\) src re thus ?thesis using assms(2) by simp
    next
        assume \(r v \neq s r c\) 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) (fs
(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 \not= src re
    using 1 assms by (auto simp add: fringe-def vertices-def)
    moreover
    have }\neg\mathrm{ marked prb rv
    proof (intro notI)
        assume marked prb rv
        have marked prb'rv
        proof -
            have rv \not= tgt re using assms(1)\langlerv\in red-vertices prb> by auto
            thus ?thesis using assms(1)<marked prb rv〉 by auto
    qed
    thus False using 1 by (auto simp add : fringe-def)
    qed
    ultimately
    show ?case using assms(1) by (auto simp add : fringe-def)
next
    case (2rv)
    hence rv\in red-vertices prb' using assms(1) by (auto simp add : fringe-def
```

```
vertices-def)
    moreover
    have \(r v \notin\) subsumees (subs prb') using 2 assms(1) by (auto simp add : fringe-def)
    moreover
    have ui-edge ' (out-edges (red prb') rv) \(\subset\) out-edges (black prb') (fst rv)
    using 2 assms(1) by (auto simp add : fringe-def)
    moreover
    have \(\neg\) marked \(p r b^{\prime} r v\)
    proof -
        have \(r v \neq\) tgt re using assms(1) 2 by (auto simp add : fringe-def)
        thus ?thesis using assms(1) 2 by (auto simp add : fringe-def)
    qed
    ultimately
    show ?case by (simp add : fringe-def)
qed
Adding a subsumption to the subsumption relation removes the first member of the subsumption from the fringe.
lemma subsumE-fringe :
assumes subsum-extends prb sub prb'
shows fringe prb' \(=\) fringe prb \(-\{\) subsumee sub \(\}\)
using assms by (auto simp add : fringe-def)
```


### 12.8 Red-Black Sub-Paths and Paths

The set of red-black subpaths starting in red location $r v$ is the union of :

- the set of black sub-paths that have a red counterpart starting at $r v$ 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 $r v$ and leading to an element of the fringe. Moreover, the remainings of these black sub-paths must have no non-empty counterpart in the red part. Otherwise, the set of red-black paths would simply be the set of paths of the black part.

```
definition RedBlack-subpaths-from ::
    ('vert, 'var, 'd, 'x) pre-RedBlack-scheme }=>\mathrm{ ('vert }\times\mathrm{ nat) }=>\mathrm{ 'vert edge list set
where
```

```
RedBlack-subpaths-from prb rv \(\equiv\)
    ui-es ' \(\left\{\right.\) res. \(\exists r v^{\prime}\). subpath (red prb) rv res rv' (subs prb) \(\wedge \neg\) marked prb rv'\}
\(\cup\{u i-e s\) res1 @ bes2
    | res1 bes2. \(\exists\) rv1. rv1 \(\in\) fringe prb
                            \(\wedge\) subpath (red prb) rv res1 rv1 (subs prb)
                            \(\wedge \neg(\exists\) res21 bes22. bes2 \(=u i\)-es res21 @ bes22
                            \(\wedge\) res21 \(\neq[]\)
                            \(\wedge\) subpath-from (red prb) rv1 res21 (subs prb))
                            \(\wedge\) Graph.subpath-from (black prb) (fst rv1) bes2\}
```

Red-black paths are red-black subpaths starting at the root of the red part.
abbreviation RedBlack-paths ::
('vert, 'var, 'd, 'x) pre-RedBlack-scheme $\Rightarrow$ '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 $f$ st $($ 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 $=(\lambda$ rv. False $)$
assumes strengthenings prb $=(\lambda r v .(\lambda \sigma$. 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 $=u i$－es res1＠bes2
and $\quad r v \in$ fringe $p r b$
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 $\langle r v \in$ fringe prb〉by（simp add ：fringe－def vertices－def）
qed
next
fix bes
assume bes $\in$ Graph．paths（black prb）
show bes $\in$ RedBlack－subpaths－from prb（root（red prb））
proof（case－tac out－edges（black prb）（init（black prb））$=\{ \})$
assume out－edges（black prb）（init（black prb））$=\{ \}$
show ？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 $\operatorname{assms}(3)$ by（simp add ：sub－rel－of－def subpath－def vertices－def）
next
show $\neg$ marked prb（root（red prb））using assms（5）by simp
next
show bes $=$ ui－es［］
using 〈bes $\in$ 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））$\neq\{ \}$
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) \in 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 \in Graph.paths (black prb)> by simp
                qed
            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 \subseteq 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 ? $x=b v 2$ 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 $\operatorname{simp}$

### 12.9 Preservation of feasible paths

The following theorem states that we do not loose feasible paths using our five operators, and moreover, configurations $c$ at the end of feasible red paths in some graph $p r b$ 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 R e d B l a c k - s u b p a t h s - f r o m ~ p r b ~ r v ~
using assms finite-RedBlack
proof (induct prb arbitrary : rv)
```

    case (base prb rv)
    moreover
    hence \(r v=\) 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-edges (black prb) $($ fst $($ root $($ red prb $)))=\{ \} \vee$
ui-edge 'out-edges(red prb) (root (red prb)) $\subset$ out-edges(black prb) $($ fst (root (red prb)))
using base by auto
ultimately
show ?case
using finite-RedBlack.base-RedBlack-paths[of prb]
by (auto simp only: finite-RedBlack-def)
next
case (se-step prb re c prb' rv)
have $R B^{\prime}$ : RedBlack prb' by (rule RedBlack.se-step[OF se-step $\left.(1,3)\right]$ )
show ?case
unfolding subset-iff
proof (intro allI impI)
fix bes
assume bes $\in$ feasible-subpaths-from (black prb') (confs prb' rv) (fst rv)
have $r v \in$ red-vertices $p r b \vee r v=$ tgt re
using se-step $(3,4)$ by (auto simp add : vertices-def)
thus bes $\in$ RedBlack-subpaths-from prb'rv
proof (elim disjE)
assume $r v \in$ red-vertices prb
moreover
hence $r v \neq$ tgt re using se-step by auto
ultimately
have bes $\in$ RedBlack-subpaths-from prb rv using se-step 〈bes $\in$ feasible-subpaths-from (black prb') (confs prb' rv)
(fst rv)>
by fastforce
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 rv ${ }^{\prime}$
assume bes $=$ ui-es res
and subpath (red prb) rv res rv' (subs prb)
and $\neg$ marked prb $r v^{\prime}$
moreover
hence $\neg$ marked prb' $r v^{\prime}$
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? $x=r v^{\prime}$ in $e x I$, auto)
next
fix res1 bes2 rv1 bl

```
assume A:bes =ui-es res1 @ bes2
and B:rv1\in fringe prb
and C : subpath (red prb) rv res1 rv1 (subs prb)
and \quadE:\neg(\exists res21 bes22. bes2 = ui-es res21 @ bes22
                            ^res21 = []
                            ^ subpath-from (red prb) rv1 res21 (subs prb))
and F:Graph.subpath (black prb) (fst rv1) bes2 bl
hence rv1 f= 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 f tgt re using se-step(3)<rv1 = src re〉 by auto
            thus \neg 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 \(\neq[]\)
    then obtain be bes2' where bes2 \(=\) be \# bes2' unfolding neq-Nil-conv
    by blast
show ?thesis
proof (case-tac be $=$ ui-edge re)
assume $b e=u i$-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 @ $[r e]$ in exI)
apply (intro conjI)
apply (rule-tac ? $x=$ tgt re in $e x I$ )
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 bes\mp@subsup{2}{}{\prime}= []
            using F〈bes\mathcal{Z = 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〉\langlebes2 = 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\rangle<bes2 = be # bes2'><be =ui-edge
re>
```

```
    by \(\operatorname{simp}\)
```

    by \(\operatorname{simp}\)
    next
next
case 2 show ?case
case 2 show ?case
proof (rule-tac ? $x=$ tgt re in exI, intro conjI)
proof (rule-tac ? $x=$ tgt re in exI, intro conjI)
have $\neg$ marked prb (src re)
have $\neg$ marked prb (src re)
using $B<r v 1=s r c$ re〉 by (simp add : fringe-def)
using $B<r v 1=s r c$ re〉 by (simp add : fringe-def)
thus tgt $r e \in$ fringe $p r b^{\prime}$
thus tgt $r e \in$ fringe $p r b^{\prime}$
using se-step (3) <out-edges (black prb) $($ fst (tgt re) $) \neq\{ \}\rangle$
using se-step (3) <out-edges (black prb) $($ fst (tgt re) $) \neq\{ \}\rangle$
seE-fringe $1[O F$ subs-sub-rel-of $[O F$ se-step (1)] se-step (3)]
seE-fringe $1[O F$ subs-sub-rel-of $[O F$ se-step (1)] se-step (3)]
seE-fringe $4[O F$ subs-sub-rel-of $[O F$ se-step (1)] se-step (3)]
seE-fringe $4[O F$ subs-sub-rel-of $[O F$ se-step (1)] se-step (3)]
by auto
by auto
next

```
    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 \(R B^{\prime}\), of rv res1 re tgt re]
            by auto
        next
            show \(\neg(\exists\) res21 bes22. bes2' \(=\) ui-es res21 @ bes22
                                    \(\wedge\) res21 \(\neq[]\)
                        \(\wedge\) subpath-from (red prb) (tgt re) res21 (subs
    $\left.p r b^{\prime}\right)$ )
proof (intro notI, elim exE conjE)
fix res21 bes22 ru2
assume bes2' = ui-es res21 @ bes22
and $\quad$ res21 $\neq[]$
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 $b e \neq u i$－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 ？$x=b e s 2$ in $e x I$ ）
apply（intro conjI）
apply（rule 〈bes＝ui－es res1＠bes2〉）
apply（rule－tac ？$x=r v 1$ in exI）
proof（intro conjI）

```
show rv1 \(\in\) fringe \(p^{\prime} b^{\prime}\)
    unfolding fringe-def mem-Collect-eq
    proof (intro conjI)
    show rv1 \(\in\) red-vertices prb'
    using se-step(3) B by (auto simp add: fringe-def vertices-def)
    next
        show rv1 \(\notin\) subsumees (subs prb)
        using se-step (3) B by (auto simp add : fringe-def)
    next
        show \(\neg\) marked prb \({ }^{\prime}\) rv1
        using \(B\) se-step (3) \(\langle r v 1 \neq\) tgt re〉〈rv1 = src re〉
        by (auto simp add: fringe-def)
    next
        have be \(\notin\) ui-edge‘ out-edges (red prb') rv1
            proof (intro notI)
            assume be \(\in\) ui-edge' out-edges (red prb') rv1
            then obtain \(r e^{\prime}\) where \(b e=u i\)-edge \(r e^{\prime}\)
                    and \(r e^{\prime} \in\) out-edges \(\left(r e d ~ p r b^{\prime}\right) r v 1\)
                    by blast
            show False
            using \(E\)
            apply (elim notE)
            apply (rule-tac ? \(x=[r e]\) in \(e x I\) )
            apply (rule-tac ? \(x=\) bes2' \(^{\prime}\) in \(e x I\) )
            proof (intro conjI)
            show bes2 = ui-es [re] @ bes2'
            using 〈bes2 = be \# bes2'〉〈be = ui-edge re'〉 by simp
                    next
                    show \([r e] \neq[]\) by \(\operatorname{simp}\)
                    next
                    have \(r e^{\prime} \in\) edges (red prb)
                    using se-step \((3)\left\langle r v 1=\right.\) src re〉 \(\left\langle r e^{\prime} \in\right.\) out-edges (red
                            \(\left\langle b e \neq u i\right.\)-edge re〉〈be \(=\) ui-edge \(\left.r e^{\prime}\right\rangle\)
                    by (auto simp add : vertices-def)
                    thus subpath-from (red prb) rv1 [re] (subs prb)
                    using 〈ré \(\in\) out-edges (red prb') rv1〉
                            subs-sub-rel-of [OF se-step (1)]
                    by (rule-tac ? \(x=t g t r e^{\prime}\) in exI)
                    (simp add : rb-sp-one \([\) OF se-step (1)])
                    qed
```

```
                    qed
            moreover
            have be \in 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 \subset 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-subs)
next
    show }\neg(\exists\mathrm{ 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' \in 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] }\not=[]\mathrm{ by simp
                    next
                        assume re' \in 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}\mp@subsup{}{}{\prime}
                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)\langlebe \not= ui-edge re>\langlebe=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 f 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 \in fringe prb'
            using se-step(3) B <rv1 = src re\rangle\langlerv1 = 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) \subset
                                    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 \neg (\exists res21 bes22. bes2 = ui-es res21 @ bes22
                    ^res21 f []
                            ^ 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 }=[]>\mathrm{ unfolding neq-Nil-conv by blast
    have rv1 = src re'\vee (rv1,src re') \in subs prb
    and re'\in edges (red prb')
        using se-step(3) rb-sp-Cons[OF RB]
                <subpath (red prb') rv1 res21 rv2 (subs prb')\rangle\langleres21 = re'
# res21'>
        by auto
    moreover
    have re' \in edges (red prb)
        proof -
            have re' = re
                using <rv1 = src re'\vee (rv1,src re') \in subs prb>
                proof (elim disjE, goal-cases)
                    case 1 thus ?thesis using <rv1 }=\mathrm{ src re〉 by auto
                next
                    case 2 thus ?case
                            using B unfolding fringe-def subsumees-conv
```

```
by fast
                                    qed
                                    thus ?thesis using se-step(3)\langlere' \in edges (red prb')\rangle 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)
                            <v1 = src re '\vee (rv1,src re') \in subs prb>
                    <re' \in 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 \in feasible-subpaths-from (black prb)) (confs prb' rv) (fst
rv)>
                    <v}=tgt re> se-star-sat-imp-sat
                    by (auto simp add:feasible-def)
            thus }\neg\mathrm{ 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\rangle <out-edges (black prb) (fst (tgt re)) =
{}>
                <bes \in 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)) \not={}
    show ?thesis
    unfolding RedBlack-subpaths-from-def Un-iff mem-Collect-eq
    apply (intro disjI2)
    apply (rule-tac ? }x=[]\mathrm{ in exI)
    apply (rule-tac ? }x=b\mathrm{ 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\mathrm{ in exI)
        proof (intro conjI)
            have ᄀ marked prb (src re)
            proof -
            have sat (confs prb' (tgt re))
                            using «bes \in feasible-subpaths-from (black prb') (confs prb' rv) (fst
rv)>
                        <v}=tgt re> se-star-sat-imp-sa
                    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 f= 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 fringe prb'
    using se-step(3)<rv = tgt re><out-edges (black prb) (fst (tgt re)) \not=
        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) 〈rv = tgt re〉 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
$\wedge \operatorname{res21} \neq[]$
$\wedge$ subpath-from (red prb') rv res21 (subs prb'))
proof (intro notI, elim exE conjE)
fix res1 bes22 rv'
assume bes $=u i$-es res1 @ bes22
and $\quad$ res $1 \neq[]$
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 $=r e^{\prime} \#$ res2
using 〈res $1 \neq[]$ unfolding neq-Nil-conv by blast
hence $r v=s r c r e^{\prime} \vee\left(r v, s r c r e^{\prime}\right) \in$ subs $p r b$ using se－step（3）＜subpath（red prb＇）rv res1 rv＇（subs prb＇）〉 rb－sp－Cons［OF RB＇，of rv re＇res2 rv＇］ by auto
thus ？thesis
proof（elim disjE）
assume $r v=s r c r e^{\prime}$
moreover
hence $r e^{\prime} \in$ out－edges（red prb＇）（tgt re） using «subpath（red prb＇）rv res1 rv＇（subs prb＇）＞ $\left\langle r e s 1=r e{ }^{\prime} \#\right.$ res2 $\rangle\langle r v=$ tgt re〉 by（auto simp add ：sp－Cons）
ultimately
show ？thesis using se－step（3）by auto
next
assume（rv，src re$\left.{ }^{\prime}\right) \in$ subs prb
hence tgt re $\in$ red－vertices prb
using se－step（3）$\langle r v=$ tgt re〉subs－sub－rel－of $[O F$ se－step $(1)]$
unfolding sub－rel－of－def by force
thus ？thesis using se－step（3）by auto qed
qed
thus False
proof（elim disjE）
assume out－edges（red prb＇）（tgt re）$\neq\{ \}$
thus ？thesis using se－step（3）
by（auto simp add ：vertices－def image－def）
next
assume tgt re $\in$ subsumees（subs prb＇）
hence tgt re $\in$ red－vertices prb
using se－step（3）subs－sub－rel－of［OF se－step（1）］
unfolding subsumees－conv sub－rel－of－def by fastforce
thus ？thesis using se－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 \in 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 }\in\mathrm{ 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 \in feasible-subpaths-from (black prb') (confs prb'
rv1) (fst rv1)>
        by (simp add : feasible-def) blast
    have bes \in RedBlack-subpaths-from prb rv1
        using mark-step(2)[of rv1] mark-step(3-7)
            <bes \in feasible-subpaths-from (black prb') (confs prb' rv1) (fst rv1)>
        by auto
    thus bes }\in\mathrm{ 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 \neg 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 \(\exists r v^{\prime}\). subpath (red prb\(\left.{ }^{\prime}\right)\) rv1 res rv \({ }^{\prime}\left(\right.\) subs \(\left.p r b^{\prime}\right) \wedge \neg\) marked \(p r b^{\prime} r v^{\prime}\)
    apply (rule-tac ? \(x=r v 3\) in \(e x I\) )
    proof (intro conjI)
    show subpath (red prb') rv1 res rv3 (subs prb')
        using mark-step(3) 〈subpath (red prb) rv1 res rv3 (subs prb)〉
        by auto
    next
    show \(\neg\) marked prb' rv3
    proof -
        have sat (confs prb rv3)
                proof -
                    have \(c \sqsubseteq\) confs prb rv3
                    using mark-step (1)
                            〈subpath (red prb) rv1 res rv3 (subs prb)〉
                        \(\langle b e s=u i\)-es res〉
                        «se-star (confs prb rv1) (trace bes (labelling (black prb)))
    have sat（confs prb rv3）
〈finite-RedBlack prb〉
finite-RedBlack.SE-rel
by $\operatorname{simp}$
thus ?thesis
using 〈se-star (confs prb rv1) (trace bes (labelling (black prb)))
〈sat c>
sat-sub-by-sat
by fast
qed
thus ?thesis
using mark-step (3) 〈subpath (red prb) rv1 res rv3 (subs prb)〉
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 bes $=u i$-es res by (rule $\langle$ bes $=u i$-es res〉)
qed
next

```
    fix res1 bes2 rv3 bl
    assume A : bes = ui-es res1 @ bes2
    and B:rv3 \in fringe prb
    and C : subpath (red prb) rv1 res1 rv3 (subs prb)
    and E:\neg(\existsres21 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}\mp@subsup{c}{}{\prime
            where se-star (confs prb rv1) (trace (ui-es res1) (labelling (black prb)))
c'
            and se-star c'(trace bes2 (labelling (black prb))) c
            and sat c'
                    using A<se-star (confs prb rv1) (trace bes (labelling (black prb)))
c> <sat c>
                by (simp add : se-star-append se-star-sat-imp-sat) blast
            moreover
            hence }\mp@subsup{c}{}{\prime}\sqsubseteq\mathrm{ 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 \(\in\) fringe \(\mathrm{prb}^{\prime}\) 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

$$
\begin{gathered}
\text { show } \neg(\exists \text { res21 bes22. bes2 }=\text { ui-es res21 @ bes22 } \\
\wedge \text { res21 } \neq[] \\
\wedge \text { subpath-from }\left(\text { red prb') rv3 res21 }\left(\text { subs prb }^{\prime}\right)\right) \\
\text { proof }(\text { intro notI, elim exE conjE })
\end{gathered}
$$

## fix res21 bes22 rv4

assume bes2＝ui－es res21＠bes22
and $\quad$ res21 $\neq[]$
and subpath（red prb＇）rv3 res21 rv4（subs prb）

## show False

using $E$
proof（elim notE，rule－tac ？$x=$ res21 in exI，
rule－tac ？$x=b e s 22$ in exI，intro conjI）
show bes2＝ui－es res21＠bes22 by（rule 〈bes2＝ui－es res21
＠bes22＞）
next
show res21 $\neq[]$ by（rule $\langle$ res21 $\neq[]\rangle$ ）
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 $R B^{\prime}:$ 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 $=u i$-es res
and subpath (red prb) rv res rv' (subs prb)
and $\neg$ marked prb $r v^{\prime}$
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 ? $x=r v^{\prime}$ in exI, auto)
next
fix res1 bes2 $r v^{\prime} b l$
assume $A$ : bes $=u i$-es res1 @ bes2
and $B: r v^{\prime} \in$ fringe $p r b$
and $\quad C$ : subpath (red prb) rv res1 rv' (subs prb)
and $\quad E: \neg(\exists$ res21 bes22. bes2 $=$ ui-es res21 @ bes22
$\wedge$ res21 $\neq[]$
$\wedge$ subpath-from (red prb) rv' res21 (subs prb))
and $\quad F$ : Graph.subpath (black prb) (fst rv') bes2 bl
show bes $\in$ RedBlack-subpaths-from prb' rv
proof (case-tac rv' $=$ subsumee sub)
assume $r v^{\prime}=$ subsumee sub
show ?thesis
using «bes $\in$ feasible-subpaths-from (black prb') (confs prb'rv) (fst rv) >
$A C F$
proof (induct bes2 arbitrary : bes bl rule : rev-induct, goal-cases)
case ( 1 bes bl) thus ?case
using subsum-step(3) B 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 1 in exI, intro conjI)
(rule-tac ? $x=r v^{\prime}$ in exI, auto simp add : fringe-def)
next
case (2 be bes2 bes bl)
then obtain $c 1 c 2 c 3$
where se-star (confs prb' rv) (trace (ui-es res1) (labelling (black prb)))
c1
and se-star c1 (trace bes2 (labelling (black prb))) c2
and se c2 (labelling (black prb) be) c3
and sat c3
using subsum-step (3)
by (simp add: feasible-def se-star-append se-star-append-one se-star-one)
blast
have ui-es res1 @ bes2 $\in$ RedBlack-subpaths-from prb' rv
proof -
have ui-es res1 @ bes2 $\in$ feasible-subpaths-from (black prb') (confs prb'
$r v)(f s t r v)$

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

## moreover

have feasible (confs prb' rv)
(trace (ui-es res1 @ bes2) (labelling (black prb')))
proof -
have se-star (confs prb'rv)
(trace (ui-es res1@bes2) (labelling (black 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\rangle \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 \in ui-edge 'out-edges (red prb') rv'')
            assume be \inui-edge 'out-edges (red prb') rv"
            then obtain re where be = ui-edge re
                and re\in 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? \(x=r e s @[r e]\) in \(e x I\) )
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 \(\in\) out-edges (red prb') rv \(\left.{ }^{\prime \prime}\right\rangle\)
        by (simp add : sp-append-one)
next
    show \(\neg\) marked prb \(^{\prime}\) (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 \(\in\) out-edges (red prb') \(\left.r v^{\prime \prime}\right\rangle\)
                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 \((3,5,6,7) R B^{\prime}\)
                        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 \((3,5,6,7)\)
                        «subpath (red prb\(\left.{ }^{\prime}\right)\) rv (res@[re]) (tgt re) (subs
    $\left.p r b^{\prime}\right)$ 〉
〈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 $\in$ out－edges（red prb${ }^{\prime}$ ）rv $\left.{ }^{\prime \prime}\right\rangle$
sat－not－marked［OF $R B^{\prime}$ ，of tgt re］
by（auto simp add ：vertices－def）
qed
next
show bes $=$ ui－es $(r e s @[r e])$ using $1(1) 2(3)<b e=u i$－edge
re〉 by simp
qed

## next

assume be $\notin u i$－edge＇out－edges（red prb＇）rv＂
show ？thesis
proof（case－tac rv ${ }^{\prime \prime} \in$ subsumees（subs prb＇））
assume $r v^{\prime \prime} \in$ subsumees（subs prb）
then obtain $a r v^{\prime \prime}$ where $\left(r v^{\prime \prime}, a r v^{\prime \prime}\right) \in\left(s u b s ~ p r b b^{\prime}\right)$ 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 $\in$ ui－edge＇out－edges（red prb＇）arv＇）
assume be $\in$ ui－edge＇out－edges（red prb＇）arv＂
then obtain $r e$ where re $\in$ out－edges（red prb＇）arv＇${ }^{\prime \prime}$
and $\quad b e=u i$－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 ? \(x=r e s @[r e]\) 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')〉
    \(\left\langle r e \in\right.\) out-edges (red prb') arv \(\left.{ }^{\prime \prime}\right\rangle\)
    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')〉
            \(\left\langle r e \in\right.\) out-edges (red prb') arv \(\left.{ }^{\prime \prime}\right\rangle\)
        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 \((3,5,6,7) R B^{\prime}\)
            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〉
            \(\langle b e=u i\)-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]
                <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 \neg marked prb' (tgt re)
        using «re \in 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'\prime
    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'\prime (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)
(fst rv)>
    using <bes \in feasible-subpaths-from (black prb') (confs prb'rv)
        <bes = ui-es res1 @ bes2 @ [be]>
    by (auto simp add:Graph.sp-append Graph.sp-append-one
```


## ultimately

show ？thesis
using sp－same－src－imp－same－tgt by fast
qed
show ？thesis
unfolding RedBlack－subpaths－from－def Un－iff mem－Collect－eq apply（intro disjI2）
apply（rule－tac ？$x=$ res in exI）
apply（rule－tac ？$x=[b e]$ 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 $\operatorname{simp}$
next
case 2 show ？case
apply（rule－tac ？$x=a r v^{\prime \prime}$ in exI）
proof（intro conjI）
show arv $^{\prime \prime} \in$ fringe $p r b^{\prime}$
unfolding fringe－def mem－Collect－eq
proof（intro conjI）
show arv＂$\in$ red－vertices prb＇
using＜subpath（red prb＇）rv res arv＂（subs prb＇）〉
by（simp add ：lst－of－sp－is－vert）
next
show arv＂$\notin$ subsumees（subs prb＇）
using $\left\langle\left(r v^{\prime \prime}, a r v^{\prime \prime}\right) \in\right.$ subs prb＇〉subs－wf－sub－rel［OF RB］
unfolding wf－sub－rel－def Ball－def
by（force simp del ：split－paired－All）
next
show $\neg$ marked prb＇arv＂
using $\left\langle\left(r v^{\prime \prime}\right.\right.$, arv $\left.^{\prime \prime}\right) \in($ subs prb＇$\left.)\right\rangle$ subsumer－not－marked $[O F$
$R B^{\dagger}$
by fastforce
next
have be $\in$ 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'` }\subset\mathrm{ out-edges (black
```

$\left.p r b^{\prime}\right)$
$R B^{\prime}$, of $\left.a r v^{\prime \prime}\right]$
$\left.p r b^{\prime}\right)$ )

$$
\begin{aligned}
& \text { by auto } \\
& \text { qed }
\end{aligned}
$$

next
show subpath (red prb') rv res arv" (subs prb') by (rule «subpath (red prb') rv res arv" (subs prb') ))
next

```
show \(\neg(\exists\) res21 bes22. \([b e]=\) ui-es res21 @ bes22
                                    \(\wedge\) res21 \(\neq[]\)
                                    \(\wedge\) subpath-from (red prb') arv" res21 (subs
```

proof (intro notI, elim exE conjE, goal-cases)
case (1 res21 bes22 rv ${ }^{\prime \prime \prime}$ )
have be $\in$ 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 $=u$ i-edge re and $r e \in$ out-edges (red prb') arv"
proof -
show be $=$ ui-edge re using $1(1)\langle r e s 21=r e \# r e s 21\rangle$
by $\operatorname{simp}$
next
have re edges (red prb')
using 1 (3) 〈res21 = re \# res21'〉 by (simp add :
sp-Cons)
moreover
have src re $=a r v^{\prime \prime}$
proof -
have ( $a r v^{\prime \prime}$, src re) $\notin$ subs prb'
using «(rv"',arv $\left.{ }^{\prime \prime}\right) \in$ subs $\left.p r b^{\prime}\right\rangle$ subs-wf-sub-rel $[O F$
$R B^{\prime}$
unfolding wf-sub-rel-def Ball-def
by（force simp del ：split－paired－All）
thus ？thesis
using $1(3)\langle r e s 21=r e \#$ res21＇〉
by（simp add ：rb－sp－Cons［OF RB］）
qed
ultimately
show re $\in$ 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 ${ }^{\prime \prime}$ ）［be］ using subsum－step（3）

〈Graph．subpath（black prb）（fst rv＇）（bes2＠［be］）bl＞〈（rv $\left.{ }^{\prime \prime}, a^{\prime} v^{\prime \prime}\right) \in$ subs prb ${ }^{\prime}>$〈subpath（red prb＇）rv res arv＂（subs prb＇）〉 $\left\langle s r c\right.$ be $\left.=f s t a r v^{\prime \prime}\right\rangle$ $R B^{\prime}$ 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"' }\not=\mathrm{ subsumees (subs prb')
show ?thesis
proof (case-tac be \inui-edge' out-edges (red prb') rv')
    assume be \inui-edge'out-edges (red prb')rv'
    then obtain re where be = ui-edge re
            and re\in out-edges (red prb')rv'\prime
        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') >
            \(\left\langle r e \in\right.\) out-edges \(\left(\right.\) red \(\left.\left.p r b^{\prime}\right) r v^{\prime \prime}\right\rangle\)
        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') 〉
                    \(\left\langle r e \in\right.\) out-edges (red prb') rv \(\left.{ }^{\prime \prime}\right\rangle\)
            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 \((3,5,6,7) R B^{\prime}\)
            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]
                        <subpath (red prb') rv (res@[re]) (tgt re) (subs
prb')
    next
        assume A : be & ui-edge 'out-edges (red prb') rv''
    show ?thesis
            unfolding RedBlack-subpaths-from-def Un-iff Bex-def
mem-Collect-eq
                by (simp add: finite-RedBlack-def)
            ultimately
            show ?thesis by (simp add: sat-sub-by-sat)
        qed
            thus ?thesis
            using 〈re \in 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 <bes=ui-es res1@ bes2 @ [be]>
            <ui-es res1 @ bes2 = ui-es res`
            <e = ui-edge re`
        by simp
qed
    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 «ui-es res1@ bes2 = ui-es res`
                            <bes=ui-es res1 @ bes2 @ [be]>
                            by simp
            next
```

            case 2
            have \(s r c b e=f s t r v^{\prime \prime}\)
            proof -
    be）
$\left.\left.\left.p r b^{\prime}\right)\right\rangle\right]$
［be］）bl＞
have Graph．subpath（black prb＇）（fst rv）（ui－es res）（src
using «bes $\in$ 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
［OF $R B^{\prime}$ «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 $R B^{\prime}$ «subpath（red prb$\left.b^{\prime}\right)$ rv res rv ${ }^{\prime \prime}\left(\right.$ subs prb$\left.\left.b^{\prime}\right)\right\rangle$ ］
by（rule sp－same－src－imp－same－tgt）
qed
show ？case
apply（rule－tac ？$x=r v^{\prime \prime}$ in $e x I$ ）
proof（intro conjI）
show $r v^{\prime \prime} \in$ fringe prb ${ }^{\prime}$
unfolding fringe－def mem－Collect－eq
proof（intro conjI）
show $r v^{\prime \prime} \in$ red－vertices $p r b^{\prime}$
using 〈subpath（red prb＇）rv res rv＂（subs prb＇）〉
by（simp add：lst－of－sp－is－vert）
next
show $r v^{\prime \prime} \notin$ subsumees（subs prb＇）
by（rule $\left.\left.\left.\left\langle r v^{\prime \prime} \notin \text { subsumees（subs prb}\right)^{\prime}\right)\right\rangle\right)$
next
show $\neg$ marked $p r b^{\prime} r v^{\prime \prime}$ by（rule $\neg$ marked $\left.\left.p r b^{\prime} r v^{\prime \prime}\right\rangle\right)$
next
have be $\in$ edges（black prb＇）
using subsum－step（3）
〈Graph．subpath（black prb）（fst rv’）（bes2＠
by（simp add ：Graph．sp－append－one）
thus ui－edge＇out－edges（red prb＇）rv＇$\subset$ out－edges（black prb＇）（fst rv＂）
using $\left\langle s r c\right.$ be $\left.=f s t r v^{\prime \prime}\right\rangle A$ red－OA－subset－black－OA［OF RB＇，of $\left.r v^{\prime \prime}\right]$
by auto
qed
next
show subpath（red prb＇）rv res rv ${ }^{\prime \prime}$（subs prb＇）
by（rule 〈subpath（red prb＇）rv res rv＂（subs prb＇）〉）
next

```
show \(\neg(\exists\) res21 bes22. \([b e]=u i\)-es res21 @ bes22
            \(\wedge\) res21 \(\neq[]\)
            \(\wedge\) SubRel.subpath-from (red prb\(\left.{ }^{\prime}\right)\left(r v^{\prime \prime}\right)\)
                    (res21) (subs prb'))
proof (intro notI, elim exE conjE, goal-cases)
    case (1 res21 bes22 rv"')
    have be \(\in u i\)-edge 'out-edges (red prb') rv"
    proof -
        obtain re res21' where res21 \(=\) re \# res21'
            using 1 (2) unfolding neq-Nil-conv by blast
        have be \(=\) ui-edge re
        and re \(\in\) out-edges (red prb') rv"
        proof -
    show be \(=\) ui-edge re using \(1(1)\langle r e s 21=r e \# r e s 21 〉\)
```

by $\operatorname{simp}$
next
have re $\in$ edges (red prb')
using 1 (3) 〈res21 = re \# res21'〉 by $(\operatorname{simp}$ add :
sp-Cons)
moreover
have src $r e=r v^{\prime \prime}$
proof -
have $\left(r v^{\prime \prime}, s r c ~ r e\right) \notin s u b s ~ p r b^{\prime}$
using $\left\langle r v^{\prime \prime} \notin\right.$ subsumees (subs prb') $\rangle$ by force
thus ?thesis
using $1(3)\left\langle r e s 21=r e \#\right.$ res21 $\left.{ }^{\prime}\right\rangle$
by (simp add : rb-sp-Cons[OF RB $]$ )
qed
ultimately
show re $\in$ out-edges (red prb') rv ${ }^{\prime \prime}$ 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 ${ }^{\prime \prime}$ ）［be］ using subsum－step（3）〈Graph．subpath（black prb）（fst rv＇）（bes2＠［be］）
next

```
case (2 res1' bes2' rv'' bl')
show ?thesis
proof (case-tac bes2' = [])
```

    assume bes2 \(^{\prime}=[]\)
    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 \({ }^{\prime}\) (subs prb')
                using subsum-step (3) «subpath (red prb) rv res1 rv' (subs
    $p r b)$ 〉

```
        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 \in edges (black prb')
and tgt be = bl
    by (simp-all add : Graph.sp-append-one)
have fst rv"' = src be
proof -
    have Graph.subpath (black prb') (fst rv) (ui-es res1') (fst rv')
        using <subpath (red prb') rv res1'rv'\prime (subs prb')>
            red-sp-imp-black-sp[OF RB]
        by fast
    thus ?thesis
        using <Graph.subpath (black prb') (fst rv) (ui-es res1') (src
be)>
            by (simp add : sp-same-src-imp-same-tgt)
qed
show ?thesis
proof (case-tac be \in ui-edge` out-edges (red prb') rv')
    assume be \inui-edge 'out-edges (red prb')rv'
    then obtain re where be = ui-edge re
```

```
            and re\inout-edges (red prb')rv'\prime
    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 \in out-edges (red prb') rv'>
        by (simp add : sp-append-one)
    next
    show ᄀ 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 \in 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\rangle\langlebes2' = []>
                        <ui-es res1@ bes2 = ui-es res1'@
bes2'>
                        by simp
```

thus ?thesis
using subsum-step (3) se-star-succs-states[OF
,

$$
\text { «bes } \in \text { feasible-subpaths-from (black }
$$

```
show bes = ui-es res1' @ [be]
    using <bes = ui-es res1 @ bes2 @ [be]>
            <ui-es res1 @ bes2 = ui-es res1'@ bes2'>
            <bes2' = []>
    by simp
```


## next

```
    case 2 show ?case
    apply (rule-tac ? }x=r\mp@subsup{v}{}{\prime\prime}\mathrm{ in exI)
    proof (intro conjI)
        show rv '\prime }\in\mathrm{ fringe prb' by (rule <rv'' }\in\mathrm{ fringe prb'>)
    next
    show subpath (red prb') rv res1'rv' (subs prb')
        by (rule <subpath (red prb') rv res1' rv'' (subs prb')>)
```

next

```
    show \neg(\exists res21 bes22. [be] = ui-es res21 @ bes22
                        ^res21 f= []
                        ^ subpath-from (red prb') (rv'')
                (res21) (subs prb'))
    proof (intro notI, elim exE conjE, goal-cases)
        case (1 res21 bes22 rv'\prime')
        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)\langlerv'\prime }\in\mathrm{ fringe prb'> RB'
            unfolding subsumees-conv by (force simp add :
                rb-sp-Cons)
    ultimately
    show False using A by auto
    qed
```

fringe-def

```
        next
            show Graph.subpath-from (black prb') (fst rv'') [be]
                    using <Graph.subpath (black prb')(fst rv)(ui-es
res1'@[be])bl>
                            <st rv '\prime = src be>
                            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 <bes =ui-es res1 @ bes2 @ [be]>
            〈ui-es res1 @ bes2 = ui-es res1'@ bes2'`
        by simp
    next
        case 2 show ?case
            apply (rule-tac ? }x=r\mp@subsup{v}{}{\prime\prime}\mathrm{ in exI)
            proof (intro conjI)
            show rv'\prime}\in\mathrm{ fringe prb' by (rule <rv'' }\in\mathrm{ fringe prb'>)
            next
```

                show subpath (red prb') rv res1' rv' (subs prb')
            by (rule 〈subpath (red prb') rv res1' rv' (subs prb')>)
    next

$$
\text { show } \neg(\exists \text { res21 bes22. bes2' @ }[\text { be }]=\text { ui-es res21 @ }
$$

$$
\begin{aligned}
& \wedge \text { res21 } \neq[] \\
& \wedge \text { subpath-from }(\text { red prb' })\left(r v^{\prime \prime}\right)
\end{aligned}
$$

$$
(\text { res21 })(\text { subs prb' }))
$$

proof（intro notI，elim exE conjE，goal－cases）

$$
\text { case ( } 1 \text { res21 bes22 rv"') }
$$

then obtain re res21＇where res21＝re \＃res21＇
and $b e^{\prime}=$ ui－edge re
using $\left\langle b e s 2^{\prime}=b e^{\prime} \#\right.$ bes2 ${ }^{\prime \prime}>$ unfolding neq－Nil－conv
by auto
show False using $\langle\neg(\exists$ res21 bes22．bes2＇$=$ ui－es res21＠

$$
\wedge \operatorname{res} 21 \neq[]
$$

$\wedge$ subpath－from（red prb＇）（rv ${ }^{\prime \prime}$ ）
（res21）（subs prb＇））＞
apply（elim notE）
apply（rule－tac ？$x=[r e]$ in $e x I$ ）
apply（rule－tac ？$x=b e s 2^{\prime \prime}$ in exI）
proof（intro conjI）
show bes2＇$=$ ui－es $[r e] @ b e s 2^{\prime \prime}$
using 〈bes2 ${ }^{\prime}$＠$[b e]=u i$－es res21＠bes22〉
$\left\langle b e s 2^{\prime}=b e^{\prime} \#\right.$ bes2 $\left.{ }^{\prime \prime}\right\rangle$
$\left\langle b e^{\prime}=u i\right.$－edge re〉 by $\operatorname{simp}$
next
show $[r e] \neq[]$ by $\operatorname{simp}$
next
show subpath－from（red prb＇）rv ${ }^{\prime \prime}[r e]$（subs prb＇）
using «subpath（red prb＇）rv＂res21 rv ${ }^{\prime \prime \prime}($ subs
$\left\langle r e s 21=r e \# r e s 21{ }^{\prime}\right\rangle$
by（fastforce simp add ：sp－Cons Nil－sp
vertices－def）

## qed

qed
next
show Graph．subpath－from（black prb＇）（fst rv ${ }^{\prime \prime}$ ）（bes2＇＠

```
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 \(\langle\) bes \(\in\) feasible-subpaths-from (black prb')
                                    (confs prb'rv)
                                    ( \(f s t r v\) ) >
                    〈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'>
bes2') $b l^{\prime}$
$b l>$
Graph.sp-one)
qed
moreover
have Graph.subpath (black prb')(fst rv)(ui-es res1' @
using 〈Graph.subpath (black prb') (fst rv ${ }^{\prime \prime}$ ) bes2' ${ }^{\prime} l^{\prime}$ 〉
red-sp-imp-black-sp[OF RB'
«subpath (red prb$\left.{ }^{\prime}\right)(r v)\left(r e s 1^{\prime}\right)$
(rv't) (subs prb')>]
by (auto simp add : Graph.sp-append)
ultimately
have src be $=b l^{\prime}$ by (rule sp-same-src-imp-same-tgt)
moreover
have Graph.subpath (black prb') (src be) [be] (tgt be)
using subsum-step(3)
〈Graph.subpath (black prb) (fst rv') (bes2@[be])
by (auto simp add : Graph.sp-append-one
ultimately
show ?thesis
using 〈Graph.subpath (black prb') (fst rv ' $)$ bes2 ${ }^{\prime}$ bl'>
by (simp add : Graph.sp-append-one Graph.sp-one)
qed
qed
qed

```
                    qed
            qed
        qed
    next
    assume rv'\not= 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=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 fringe prb'
            using subsum-step(3) subsumE-fringe[OF subsum-step(3)] B<rv'\not=
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\mathrm{ res21 bes22.bes2 = ui-es res21 @ bes22
                    ^res21 f []
                            ^ subpath-from (red prb')rv' res21 (subs prb'))
            proof (intro notI, elim exE conjE)
                fix res21 bes22 rv"
            assume bes2 = ui-es res21 @ bes22
            and res21 f= []
            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\mathrm{ uses-sub rv ' [re] (tgt re) sub using <rv' # subsumee sub>
```

by auto

```
            thus ?thesis
            using subsum-step(3)
            <subpath (red prb') rv` res21 rv' (subs prb')\rangle\langleres21 = re #
res21'>
simp
prb}\mp@subsup{}{}{\prime})
                    <res21 = re # res21}\mp@subsup{}{}{\prime}
                    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 $c_{a}$ prb' rv1)
have $R B^{\prime}$ : RedBlack prb' by (rule RedBlack.abstract-step[OF abstract-step $\left.(1,3)\right]$ )
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 $\in$ feasible-subpaths-from (black prb') (confs prb' rv1) (fst rv1)
show bes $\in$ RedBlack-subpaths-from prb' rv1
proof (case-tac rv2 $=$ rv1 $)$
assume rv2 $=r v 1$
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 ? $x=r v 1$ in $e x I$ )
proof (intro conjI)
show subpath (red prb') rv1 [] rv1 (subs prb')
using abstract-step (4) rb-Nil-sp[OF RB] by fast
next
show $\neg$ marked $p r b^{\prime}$ rv1 using abstract-step (3) 〈rv2 $\left.=r v 1\right\rangle$
by $\operatorname{simp}$
next
show bes $=$ ui-es []
using $\langle$ bes $\in$ feasible-subpaths-from (black prb') (confs prb' rv1)
(fst rv1)>
〈out-edges (black prb') (fst rv1) $=\{ \}\rangle$
by (cases bes) (auto simp add : Graph.sp-Cons)
qed

```
    next
    assume out-edges (black prb') (fst rv1) \not={}
    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 \in 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 ~ r e s 2 1 ~ = ~ [ ] ~
                            ^ subpath-from (red prb') rv1 res21 (subs prb})
        proof (intro notI, elim exE conjE)
        fix res21 rv3
            assume res21 f= []
            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 $\in$ 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 $\langle$ bes $\in$ feasible－subpaths－from（black prb＇）（confs prb＇rv1） （fst rv1）＞
by $\operatorname{simp}$
qed
qed
qed
next
assume rv2 $\neq r v 1$
moreover
hence feasible（confs prb rv1）（trace bes（labelling（black prb）））
using abstract－step（3）
〈bes $\in$ feasible－subpaths－from（black prb＇）（confs prb＇rv1）（fst rv1）＞
by $\operatorname{simp}$
ultimately
have bes $\in$ RedBlack－subpaths－from prb rv1
using abstract－step（2）［of rv1］abstract－step（3－7）
$\langle b e s \in$ 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 \neg 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\mathrm{ in exI, simp-all)
            next
                    fix res1 bes2 rv3 bl
                assume A : bes = ui-es res1 @ bes2
                    and }B:rv3\in fringe prb
            and C : subpath (red prb) rv1 res1 rv3 (subs prb)
            and E : \neg(\exists 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
    qed
next
    case (strengthen-step prb rv2 e prb' rv1)
    show?case
    unfolding subset-iff
    proof (intro allI impI)
```

fix bes
assume bes $\in$ feasible-subpaths-from (black prb') (confs prb' rv1) (fst rv1) hence bes $\in$ RedBlack-subpaths-from prb rv1
using strengthen-step(2)[of rv1] strengthen-step(3-7) by auto
thus bes $\in$ 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 ru2
assume bes $=$ ui-es res
and subpath (red prb) rv1 res rv2 (subs prb)
and $\neg$ 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 $\quad B: r v 3 \in$ fringe $p r b$
and $\quad C$ : subpath (red prb) rv1 res1 rv3 (subs prb)
and $\quad E: \neg(\exists$ res21 bes22. bes2 $=$ ui-es res21 @ bes22
$\wedge$ res21 $\neq[]$
$\wedge$ subpath-from (red prb) rv3 res21 (subs prb))
and $\quad F$ : 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 $e x I$ )
apply (rule-tac ? $x=b e s 2$ in exI)
proof (intro conjI, goal-cases)
show bes $=$ ui-es res1 @ bes2 by (rule 〈bes $=u i$-es res1 @ bes2〉)
next
case 2
show ?case
using strengthen-step(3) $B C E F$ unfolding fringe-def by (rule-tac ? $x=r v 3$ in exI) auto

## qed

## qed <br> qed <br> 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 ca 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))) $(f s t(\operatorname{root}($ red $p r b)$ ))
unfolding mem-Collect-eq
proof (elim exE conjE, goal-cases) case (1 bl')
show ?case
proof (rule-tac ? $x=b l^{\prime}$ 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 $\forall e \in p r e d$ (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 $\forall e \in p r e d ~(i n i t-c o n f ~ p r b)$. finite (Bexp.vars e)
using assms by (intro finite-init-pred-symvars)

## moreover

have init-conf prb $\sqsubseteq$ 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 CEGARlike 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 Assignlabels 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 $C A V^{\prime}$ '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, machinechecked 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 Objectoriented Data Models in HOL with an Application to IMP++," Journal of Automated Reasoning (JAR), vol. Selected Papers of the AVOCSVERIFY Workshop 2006, no. 3-4, pp. 219-249, 2008, serge Autexier, Heiko Mantel, Stephan Merz, and Tobias Nipkow (eds).

