Appendix to the Article "On Time-Sensitive Control Dependencies"

Simon Bischof, Martin Hecker, Gregor Snelting

August 11, 2021

Contents

Appendix to the Article "On Time-Sensitive Control Dependencies" containing definitions and proofs for NTICD, NTSCD and TSCD.

In this theory, we use Isabelle's "theorem" command for results presented in the article, and the "lemma" command for all lemmas that are needed to prove the former.

theory *NTXCD-Proofs*
imports
  Slicing.Postdomination
  Coinductive.Coinductive-List
  Digraph-Basic
begin

The CFG locale gives us a graph structure. Loops are permitted, but multi-edges are not. Isolated nodes are not permitted (they are not interesting for us anyway). The graph is assumed to have an entry node (which does not have to be unique). There are no assumptions regarding exit nodes, reachability from the entry node or whether the graph is reducible.

There is no explicit node or edge set, instead there is a predicate *valid-edge* that describes whether an edge is valid. Nodes are valid if they are source or target node of a valid edge (this is the reason why isolated nodes are not permitted). Edges are labeled, but we do not use those labels in this theory.

In the CFG locale, a graph can be infinite. In this theory, however, we assume graphs to be finite, and add this assumption to lemmas if needed.

context *CFG*
begin

1 Basic Definitions and Lemmas

successor set of a node

definition *succs* :: *'node ⇒ 'node set*
  where *succs n* == {*targetnode e* | *e. valid-edge e* ∧ *sourcenode e* = *n*}

edge relation

definition *edge-rel* ≡ {(*n1*, *n2*). *n2* ∈ *succs n1*}

Definitions of a path. Note that in the node list, the start node is included (for non-empty paths) but the end node is not.

abbreviation *is-path* :: *'node ⇒ 'node list ⇒ 'node ⇒ bool*
  where *is-path n ns n'* == *Digraph-Basic.path edge-rel n ns n'* ∧ *valid-node n*

Definitions of a path reachability

definition *reaches* :: *'node ⇒ 'node ⇒ bool*
  where *reaches n m* == ∃ *ns. path n ns m*

Lemmas about Paths

lemma *succs-valid*: $y \in succs\ x \Longrightarrow valid\text{-}node\ x \wedge valid\text{-}node\ y$
using *succs-def* **by** *auto*

lemma *is-path-valid-node*: $is\text{-}path\ n\ ns\ m \Longrightarrow valid\text{-}node\ m$
using *path-append-conv*[*of edge-rel*] *edge-rel-def succs-def* **by** (*cases ns rule*: *rev-cases*) *auto*

lemma *succs-path*: $x \in succs\ p \Longrightarrow is\text{-}path\ p\ [p]\ x$
using *edge-rel-def succs-def* **by** (*auto intro*: *path1*)

lemma *is-path-succs-empty*: **assumes** *is-path n ns m*
succs n = {}
shows *ns* = [] ∧ *n* = *m*
proof−
from *assms* **have** *Digraph-Basic.path edge-rel n ns m* **by** *simp*
from *this assms* **show** *?thesis* **unfolding** *edge-rel-def* **by** *cases auto*
qed

lemma *path-to-is-path*: **assumes** *path n es n′*
shows *is-path n* (*sourcenodes es*) *n′*
using *assms*
proof (*induction rule*: *path.induct*)
case (*Cons-path n″ as n′ a n*)
with *edge-rel-def succs-def sourcenodes-def* **show** *?case* **by** (*auto intro*: *Digraph-Basic.path.intros*)
qed (*auto simp add*: *sourcenodes-def*)

lemma *path-append*: *is-path n ns n′* ⟹ *is-path n′ ns′ n″* ⟹ *is-path n* (*ns*@*ns′*) *n″*
using *path-conc* **by** *auto*

lemma *succs-path-extend*: *x* ∈ *succs p* ⟹ *is-path x ns y* ⟹ *is-path p* (*p*#*ns*) *y*
using *edge-rel-def succs-def* **by** (*auto intro*: *path-prepend*)

lemma *is-path-split*: **assumes** *is-path u* (*ns1*@*n*#*ns2*) *v*
shows *is-path u ns1 n is-path n* (*n*#*ns2*) *v*
proof−
from *assms path-conc-conv*[*of - u*] **obtain** *n′*
where *path-gen*: *Digraph-Basic.path edge-rel u ns1 n′*
Digraph-Basic.path edge-rel n′ (*n*#*ns2*) *v* **by** *auto*
with *this*[*unfolded path-cons-conv*] *edge-rel-def succs-def assms*
show *is-path u ns1 n is-path n* (*n*#*ns2*) *v* **by** *auto*
qed

lemma *path-split-elem*: **assumes** *is-path n ns n′*
m ∈ *set ns*
obtains *ns1 ns2* **where** *ns* = *ns1*@*m*#*ns2 is-path n ns1 m*
is-path m (*m*#*ns2*) *n′*
proof−

from *split-list*[*OF assms*(*2*)] **obtain** *ns1 ns2* **where** *ns* = *ns1*@*m*#*ns2* **by** *auto*
with *that is-path-split*[*OF assms*(*1*)[*unfolded this*]] **show** *?thesis* **by** *auto*
qed

lemma *path-split-elem2*: **assumes** *is-path n ns n′*
$m \in set\ ns \cup \{n'\}$
obtains *ns1 ns2* **where** *ns* = *ns1*@*ns2* *is-path n ns1 m is-path m ns2 n′*
proof (*cases* $m \in set\ ns$)
case *True*
with *path-split-elem*[*OF assms*(*1*) *True*] *that* **show** *?thesis* **by** *metis*
next
case *False*
with *assms path0 that*[*of ns* []] *is-path-valid-node* **show** *?thesis* **by** *auto*
qed

lemma *edge-rel-impl-path*:
$(a, b) \in edge\text{-}rel \Longrightarrow is\text{-}path\ a\ [a]\ b$
using *edge-rel-def succs-path* **by** *simp*

lemma *edge-impl-valid-target*: $(a,b) \in edge\text{-}rel \Longrightarrow valid\text{-}node\ b$
unfolding *edge-rel-def succs-def* **by** *auto*

lemma *edge-rel-rtrancl-path*:
assumes $(a,b) \in edge\text{-}rel^*$ **and** *valid-node a* **shows** $\exists ns.\ is\text{-}path\ a\ ns\ b$
using *assms*
proof (*induction rule*:*rtrancl-induct*)
case *base*
with *path0* **show** *?case* **by** *metis*
next
case (*step y z*)
then obtain *ns* **where** *is-path a ns y* **by** *blast*
with *step path-append edge-rel-impl-path* **have** *is-path a* (*ns*@[*y*]) *z* **by** *auto*
thus *?case* **by** *auto*
qed

lemma *reaches-intros*:
$valid\text{-}node\ n \Longrightarrow reaches\ n\ n$
$valid\text{-}edge\ e \Longrightarrow sourcenode\ e = n \Longrightarrow targetnode\ e = m \Longrightarrow reaches\ n\ m$
using *path.intros path-edge reaches-def* **by** *metis+*

lemma *reaches-trans*: $reaches\ n1\ n2 \Longrightarrow reaches\ n2\ n3 \Longrightarrow reaches\ n1\ n3$
using *path-Append reaches-def* **by** *metis*

lemma *scc-path*:
assumes $n \in scc\text{-}of\ edge\text{-}rel\ m$ **and** *valid-node m*
obtains *ns* **where** *is-path m ns n*
using *assms node-in-scc-of-node scc-of-is-scc is-scc-connected edge-rel-rtrancl-path* **by** *metis*

lemma *lset-split*: **assumes** $n \in$ *lset ns*
obtains *ns1 ns2* **where** *ns* = *lappend* (*llist-of ns1*) (*LCons n ns2*)
using *split-llist*[*OF assms*, *unfolded lfinite-eq-range-llist-of*] **by** *auto*

lemma *lset-split-first*: **assumes** $n \in$ *lset ns*
obtains *ns1 ns2* **where** *ns* = *lappend* (*llist-of ns1*) (*LCons n ns2*)
$n \notin$ *set ns1*
using *split-llist-first*[*OF assms*, *unfolded lfinite-eq-range-llist-of*] **by** *auto*

lemma *is-path-Cons*: *is-path n* (n'#*ns*) $m \Longrightarrow n = n' \wedge (\exists x.\ x \in$ *succs n* $\wedge$ *is-path x ns m*)
using *path-cons-conv*[*of edge-rel*] *edge-rel-def succs-valid* **by** *auto*

lemma *is-path-snoc*: *is-path n* (*ns*@[n']) $m \Longrightarrow m \in$ *succs* $n' \wedge$ *is-path n ns* n'
using *path-append-conv*[*of edge-rel*] *edge-rel-def* **by** *auto*

lemma *path-first*: **assumes** *is-path n ns m*
obtains ns' ns'' **where** *is-path n* ns' *m* $m \notin$ *set* ns' *ns* = ns'@ns''
using *assms*
proof (*cases* $m \in$ *set ns*)
case *True*
from *split-list-first*[*OF this*] **obtain** ns' *ns2* **where** *ns* = ns'@*m*#*ns2* $m \notin$ *set* ns' **by** *auto*
with *is-path-split*[*OF assms*[*unfolded this*(*1*)]] *that* **show** *?thesis* **by** *auto*
qed *auto*

lemma *path-last*: **assumes** *is-path n ns m*
ns $\neq$ []
obtains ns' ns'' **where** *is-path n* (*n*#ns'') *m* $n \notin$ *set* ns'' *ns* = ns'@*n*#ns''
using *assms*
proof (*cases ns*)
case (*Cons* n' *ns2*)
with *is-path-Cons assms* **have** $n \in$ *set ns* **by** *auto*
with *split-list-last* **obtain** *ns3 ns4* **where** *ns* = *ns3*@*n*#*ns4* $n \notin$ *set ns4* **by** *metis*
with *is-path-split assms that* **show** *?thesis* **by** *blast*
qed *auto*

lemma *path-end-unique*: **assumes** $\exists$ *ns*. *is-path n ns m*
$n \neq m$
obtains ns' **where** *is-path n* (*n*#ns') *m* $m \notin$ *set* ns' $n \notin$ *set* ns'
proof−
from *assms* **obtain** *ns* **where** *path*: *is-path n ns m ns* $\neq$ [] **by** *force+*
with *path-last assms* **obtain** *ns1* **where** *is-path n* (*n*#*ns1*) *m* $n \notin$ *set ns1* **by** *metis*
with *path-first*[*OF this*(*1*)] **obtain** *ns3 ns4*
where *second-split*: *is-path n ns3 m* $m \notin$ *set ns3 n*#*ns1* = *ns3*@*ns4* **by** *auto*

with *assms* **obtain** *n′ ns3′* **where** *ns3* = *n′#ns3′* **by** (*cases ns3*) *auto*
with *second-split* **have** *ns1* = *ns3′@ns4 m* ∉ *set ns3′* **by** *auto*
with *second-split* ‹*n* ∉ *set ns1*› *that* **show** *?thesis* **by** *auto*
qed

lemma *path-rev-last*: **assumes** *is-path p ns n*
shows *last* (*n#rev ns*) = *p*
using *assms*
proof (*cases ns*)
case *Cons*
with *assms*[*unfolded this*, *unfolded path-cons-conv*] **show** *?thesis* **by** *auto*
qed *auto*

lemma *is-path-induct*[*consumes 1*]:
assumes *is-path n ns m*
valid-node m ⟹ *P m* [] *m*
⋀*n x ns*. *is-path n* (*n#ns*) *m* ⟹ *x* ∈ *succs n* ⟹ *is-path x ns m* ⟹ *P x ns m*
⟹ *P n* (*n#ns*) *m*
shows *P n ns m*
proof−
from *assms* **have** *Digraph-Basic.path edge-rel n ns m valid-node n* **by** *auto*
from *this assms edge-rel-def assms succs-valid* **show** *?thesis* **by** *induction auto*
qed

end

2 Lemmas 1.1 and 1.2

2.1 Standard control dependency, Lemma 1.1

The assumption that there is a unique exit node reachable from all other nodes is given by the Postdomination locale.

context *Postdomination*
begin

lemma *Exit-is-path*: *valid-node n* ⟹ ∃ *ns*. *is-path n ns* (*-Exit-*)
using *Exit-path path-to-is-path* **by** *blast*

lemma *Exit-succs*: *succs* (*-Exit-*) = {}
using *succs-def Exit-source* **by** *auto*

The Postdomination framework does not allow the exit node to postdominate any node. However, in reality it postdominates every (valid) node. Therefore, this definition expresses the correct postdominance relation.

definition *postdom* :: *′node* ⇒ *′node* ⇒ *bool* (- *postdom* - [*51*,*50*])
where *n′ postdom n* ≡ *n′* = (*-Exit-*) ∨ *n′ postdominates n*

Definition of control dependence introduced by Wolfe [31]. This is the definition we use.

definition *cd* :: *'node* ⇒ *'node* ⇒ *bool*
where *cd n m* == (∃ *x1*∈*succs n*. *m postdom x1*) ∧ (∃ *x2*∈*succs n*. ¬ *m postdom x2*)

lemma *postdom-succs*: **assumes** *m postdom n*
x ∈ *succs n*
n ≠ *m*
shows *m postdom x*
proof−
from *assms succs-def* **obtain** *e*
where *e-gen*: *valid-edge e sourcenode e* = *n targetnode e* = *x* **by** *auto*
{
fix *es*
assume *path x es* (-*Exit*-) *m* ≠ (-*Exit*-)
with *path.intros e-gen assms postdominate-def postdom-def*
have *m* ∈ *set* (*sourcenodes* (*e#es*)) **by** *auto*
with *sourcenodes-def e-gen assms* **have** *m* ∈ *set* (*sourcenodes es*) **by** *auto*
}
with *assms postdominate-def e-gen postdom-def* **show** *?thesis* **by** *auto*
qed

lemma *postdom-refl*: *valid-node n* ⟹ *n postdom n*
using *postdominate-refl postdom-def* **by** *auto*

lemma *postdom-intro-all-succs*: **assumes** *succs n* ≠ {}
⋀*x*. *x* ∈ *succs n* ⟹ *m postdom x*
shows *m postdom n*
proof−
{
fix *es*
assume *path*: *path n es* (-*Exit*-) *m* ≠ (-*Exit*-)
with *empty-path-nodes assms Exit-succs* **have** *es* ≠ [] **by** *auto*
with *path path-split-Cons* **obtain** *e es'* **where** *split*: *es* = *e#es'*
valid-edge e sourcenode e = *n path* (*targetnode e*) *es'* (-*Exit*-) **by** *metis*
with *assms succs-def postdom-def postdominate-def sourcenodes-def path*
have *m* ∈ *set* (*sourcenodes es*) **by** *auto*
}
with *postdom-def postdominate-def assms succs-valid* **show** *?thesis* **by** *fastforce*
qed

Shows that for $n \neq m$, the definition of *cd* we use is equivalent to another often-used definition.

lemma *control-dependence-alt*: **assumes** *n* ≠ *m*
shows *cd n m* ⟷ (∃ *x1*∈*succs n*. *m postdom x1*) ∧ ¬ *m postdom n*
proof−
{
fix *x*

assume *not-postdom*: ¬ *m postdom n succs n* ≠ {} *m postdom x*
with *succs-def postdominate-def postdom-def* **have** *valid-node n valid-node m* **by** *auto*
with *postdominate-def not-postdom postdom-def* **obtain** *es*
where *no-m-path*: *path n es* (*-Exit-*) *m* ∉ *set* (*sourcenodes es*) **by** *auto*
from *this Exit-succs not-postdom path.intros* **obtain** *e es′*
where *valid-edge e sourcenode e* = *n es* = *e#es′ path* (*targetnode e*) *es′* (*-Exit-*)
by *cases auto*
with *succs-def postdominate-def postdom-def no-m-path sourcenodes-def not-postdom*
have ∃ *x2*∈*succs n*. ¬ *m postdom x2* **by** *auto*
}
with *cd-def postdom-succs assms* **show** *?thesis* **by** *fast*
qed

lemma *postdom-cd-variant*: **assumes** *n* ≠ *m* ¬ *m postdom n*
shows (∃ *x*∈*succs n*. *m postdom x*)
⟷ (∃ *ns*. *is-path n ns m* ∧ (∀ *z*∈*set ns* − {*n*,*m*}. *m postdom z*)) (**is** *?L* ⟷ *?R*)
proof−
{
fix *x*
assume *x-assms*: *x* ∈ *succs n m postdom x*
with *postdominate-implies-path postdom-def assms path-to-is-path*
obtain *ns1* **where** *is-path x ns1 m* **by** *metis*
with *path-first* **obtain** *ns* **where** *ns-gen*: *is-path x ns m m* ∉ *set ns* **by** *metis*
from *this x-assms*(*2*) **have** ∀ *z*∈*set ns* − {*n*,*m*}. *m postdom z*
proof (*induction rule*: *is-path-induct*)
case (*2 x x′ ns*)
with *postdom-succs*[*of m x*] **show** *?case* **by** *auto*
qed *auto*
with *x-assms ns-gen succs-path-extend* **have** *?R* **by** *fastforce*
}
note *succs-postdom-to-path-postdom* = *this*
{
fix *ns*
assume *is-path n ns m* ∀ *z*∈*set ns* − {*n*,*m*}. *m postdom z*
from *this assms* **have** *?L*
proof (*induction rule*: *is-path-induct*)
case (*2 n x ns*)
then show *?case*
proof (*cases x* ∈ {*n*,*m*})
case *True*
from *2 postdom-def* **have** *valid-node x m* ≠ (*-Exit-*) **by** *auto*
with *True 2 postdominate-refl postdom-def* **show** *?thesis* **by** *auto*
next
case *False*
with *2*(*3*) **obtain** *x′ ns′* **where** *ns* = *x′#ns′* **by** (*cases ns*) *auto*
with *2*(*3*) *is-path-Cons* **have** *ns* = *x#ns′* **by** *auto*

```
      with 2 False show ?thesis by auto
    qed
  qed auto
 }
 with succs-postdom-to-path-postdom show ?thesis by auto
qed
```

Lemma 1.1. The right side is the original definition of control dependence by Ferrante et al. [11].

```
theorem control-dependence-alt2: assumes n ≠ m
  shows cd n m ⟷ (∃ ns. is-path n ns m ∧ (∀ z∈set ns − {n,m}. m postdom z))
                  ∧ ¬ m postdom n
  using assms control-dependence-alt postdom-cd-variant by metis

end
```

2.2 Example from Fig. 1 right, Lemma 1.2

Edge relation for Fig. 1 right.

```
definition node-rel-example1 :: nat × nat ⇒ bool
  where node-rel-example1 e == e ∈ {(1,2), (1,3), (2,3), (3,4), (1,5), (4,5)}

interpretation example1:
  CFG fst snd λx. Predicate (λs. False) node-rel-example1 1
proof unfold-locales qed (auto simp add: node-rel-example1-def)

interpretation example1:
  CFGExit fst snd λx. Predicate (λs. False) node-rel-example1 1 5
proof unfold-locales qed (auto simp add: node-rel-example1-def)

interpretation example1:
  Postdomination fst snd λx. Predicate (λs. False) node-rel-example1 1 5
proof unfold-locales
  let ?path = example1.path
  let ?valid-node = example1.valid-node
  let ?reaches = example1.reaches
  have Collect example1.valid-node = {1,2,3,4,5}
    using example1.valid-node-def node-rel-example1-def by auto
  then have valids: ⋀n. example1.valid-node n ⟷ n ∈ {1,2,3,4,5}
    by auto
  from valids example1.reaches-intros
  have self: ?reaches 1 1 ?reaches 5 5 by auto
  have node-rel-example1 (1,2)
    node-rel-example1 (1,3) node-rel-example1 (2,3)
    node-rel-example1 (3,4) node-rel-example1 (4,5)
    unfolding node-rel-example1-def by auto
  with example1.reaches-intros have step: ?reaches 1 2 ?reaches 1 3
     ?reaches 2 3 ?reaches 3 4 ?reaches 4 5 by auto
```

with *example1.reaches-trans* **have** *?reaches 1 4 ?reaches 1 5*
?reaches 2 5 ?reaches 3 5 **by** *metis+*
with *self step valids example1.reaches-def*
show $\bigwedge n.$ *?valid-node n* $\Longrightarrow \exists$ *ns. ?path 1 ns n*
$\bigwedge n.$ *?valid-node n* $\Longrightarrow \exists$ *ns. ?path n ns 5* **by** *auto*
qed

Following are the proofs for Lemma 1.2. The different statements are separated into different Isabelle theorems.

Part of Lemma 1.2

theorem *example1-y-postdom-n2*: *example1.postdom 4 3*
proof−
from *node-rel-example1-def example1.succs-def*
have *succs*: *example1.succs 3* = {*4*} **by** *simp*
with *example1.succs-valid example1.postdom-refl* **have** *example1.postdom 4 4* **by** *auto*
with *example1.postdom-intro-all-succs succs* **show** *?thesis* **by** *fastforce*
qed

Part of Lemma 1.2

theorem *example1-y-postdom-n1*: *example1.postdom 4 2*
proof−
from *node-rel-example1-def example1.succs-def* **have** *example1.succs 2* = {*3*} **by** *simp*
with *example1.postdom-intro-all-succs example1-y-postdom-n2* **show** *?thesis* **by** *fastforce*
qed

Part of Lemma 1.2

theorem *example1-y-not-postdom-Exit*: ¬ *example1.postdom 4 5*
proof
assume *example1.postdom 4 5*
with *example1.postdominate-implies-path* **obtain** *ns* **where** *example1.path 5 ns 4*
unfolding *example1.postdom-def* **by** *auto*
with *example1.path-Exit-source* **show** *False* **by** *auto*
qed

Part of Lemma 1.2

theorem *example1-cd-x-y*: *example1.cd 1 4*
proof−
from *example1.succs-def node-rel-example1-def*
have *2* ∈ *example1.succs 1 5* ∈ *example1.succs 1* **by** *auto*
with *example1.cd-def example1-y-postdom-n1 example1-y-not-postdom-Exit* **show** *?thesis* **by** *auto*
qed

3 Control Dependence in Arbitrary Graphs

3.1 Definitions for maximal paths and sink paths

context *CFG*
begin

Definition of a maximal path

coinductive *max-path* :: *'node* ⇒ *'node llist* ⇒ *bool*
  where *succs n'* = {} ⟹ *valid-node n'* ⟹ *max-path n'* (*llist-of* [*n'*])
    | *y* ∈ *succs x* ⟹ *max-path y ns* ⟹ *max-path x* (*LCons x ns*)

Nontermination-sensitive postdomination. *on-max-paths n m* ⟷ $m \sqsubseteq_{MAX} n$ ⟷ m lies on all maximal paths starting in n. See Definition 2.1.

definition *on-max-paths* :: *'node* ⇒ *'node* ⇒ *bool*
  where *on-max-paths n m* = (∀ *ns*. *max-path n ns* ⟶ *m* ∈ *lset ns*)

on-max-paths-prev n m1 m2 ⟷ on all maximal paths starting in n, m1 occurs before m2. Used to define $\rightarrow_{dod}$.

definition *on-max-paths-prev* :: *'node* ⇒ *'node* ⇒ *'node* ⇒ *bool*
  where *on-max-paths-prev n m1 m2* = (∀ *ns*. *max-path n ns* ⟶
      (∃ *ns1 ns2*. *ns* = *lappend* (*llist-of ns1*) (*LCons m1 ns2*) ∧ *m2* ∉ *set ns1*))

Helper definitions to define sinks. We use the condensation graph, where every SCC is shrunk to a single node.

definition *cond-edges* ≡ ((λ(*n1*,*n2*). (*scc-of edge-rel n1*, *scc-of edge-rel n2*)) ' *edge-rel*) − *Id*
definition *cond-nodes* ≡ {*scc*. ∃ *n*. *scc* = *scc-of edge-rel n* ∧ *valid-node n*}

lemma *cond-edges-no-self-loop*:
 assumes (*s1*,*s2*) ∈ *cond-edges* **shows** *s1* ≠ *s2* **using** *assms* **unfolding** *cond-edges-def*
by *auto*

lemma *cond-nodes-scc*: *s* ∈ *cond-nodes* ⟹ *n* ∈ *s* ⟹ *s* = *scc-of edge-rel n*
using *scc-of-unique*[*of n*] *cond-nodes-def* **by** *auto*

Lemma to ensure our definition of condensation graphs is correct

lemma *cond-edges-alt*:
  assumes *s1* ∈ *cond-nodes*
  and *s2* ∈ *cond-nodes*
shows (*s1*, *s2*) ∈ *cond-edges*
    ⟷ (∃ *n1* ∈ *s1*. ∃ *n2* ∈ *s2*. (*n1*, *n2*) ∈ *edge-rel* ∧ *scc-of edge-rel n1* ≠ *scc-of edge-rel n2*)
    (**is** *?P* ⟷ *?right*)
proof
  assume (*s1*, *s2*) ∈ *cond-edges*
  then obtain *n1 n2* **where**
    (*s1*,*s2*) = (*scc-of edge-rel n1*, *scc-of edge-rel n2*)

$(n1, n2) \in edge\text{-}rel$
$(scc\text{-}of\ edge\text{-}rel\ n1,\ scc\text{-}of\ edge\text{-}rel\ n2) \in cond\text{-}edges$
unfolding *cond-edges-def*
by (*metis* (*no-types*, *lifting*) *Diff-iff case-prod-conv imageE old.prod.exhaust*)
thus *?right* **using** *cond-edges-no-self-loop*
by (*metis node-in-scc-of-node prod.inject*)
next
assume *?right*
then obtain *n1 n2* **where** *n-props*: $n1 \in s1$ $n2 \in s2$ $(n1, n2) \in edge\text{-}rel$
$scc\text{-}of\ edge\text{-}rel\ n1 \neq scc\text{-}of\ edge\text{-}rel\ n2$ **by** *auto*
with *cond-nodes-scc assms* **have** $s1 = scc\text{-}of\ edge\text{-}rel\ n1$ $s2 = scc\text{-}of\ edge\text{-}rel\ n2$
by *auto*
with *n-props assms* **show** *?P* **unfolding** *cond-nodes-def cond-edges-def* **by** *auto*
qed

Definition of sink nodes

definition $sink\text{-}node\ n \equiv \neg(\exists scc.\ (scc\text{-}of\ edge\text{-}rel\ n,\ scc) \in cond\text{-}edges)$

Definition of sink paths

definition $sink\text{-}path :: 'node \Rightarrow 'node\ llist \Rightarrow bool$
where $sink\text{-}path\ n\ ns$
$== max\text{-}path\ n\ ns \wedge$
$(\exists n'.\ n' \in lset\ ns \wedge sink\text{-}node\ n'$
$\wedge\ (succs\ n' \neq \{\}$
$\longrightarrow (\forall n'' \in scc\text{-}of\ edge\text{-}rel\ n'.\ \neg\ lfinite\ (lfilter\ (\lambda x.\ x = n'')$
$ns))))$

Nontermination-insensitive postdomination. $on\text{-}sink\text{-}paths\ n\ m \longleftrightarrow m \sqsubseteq_{SINK}$ $n \longleftrightarrow$ m lies on all sink paths starting in n. See Definition 2.1.

definition $on\text{-}sink\text{-}paths :: 'node \Rightarrow 'node \Rightarrow bool$
where $on\text{-}sink\text{-}paths\ n\ m == \forall ns.\ sink\text{-}path\ n\ ns \longrightarrow m \in lset\ ns$

Definition that is equivalent to *on-sink-paths* but easier to work with

definition $on\text{-}ext\text{-}paths :: 'node \Rightarrow 'node \Rightarrow bool$
where $on\text{-}ext\text{-}paths\ x\ n == \forall ns\ n'.\ is\text{-}path\ x\ ns\ n'$
$\longrightarrow (\exists ns'\ n''.\ is\text{-}path\ n'\ ns'\ n''$
$\wedge\ n \in set\ (ns@ns'@[n'']))$

lemma *subseteq-mono*[*mono*]: $(\bigwedge x.\ P\ x \longrightarrow Q\ x) \Longrightarrow A \subseteq \{x.\ P\ x\} \longrightarrow A \subseteq \{x.$
$Q\ x\}$
by *auto*

Definition of NTSCD (Definition 2.2)

definition $ntscd :: 'node \Rightarrow 'node \Rightarrow bool$
where $ntscd\ p\ n == (\exists x1 \in succs\ p.\ on\text{-}max\text{-}paths\ x1\ n) \wedge (\exists x2 \in succs\ p.\ \neg$
$on\text{-}max\text{-}paths\ x2\ n)$

Definition of NTICD (Definition 2.2)

definition *nticd* :: *'node* ⇒ *'node* ⇒ *bool*
where *nticd p n* == (∃ *x1*∈*succs p*. *on-sink-paths x1 n*) ∧ (∃ *x2*∈*succs p*. ¬ *on-sink-paths x2 n*)

Rule system defined in Theorem 2.1 (least fixed point).

inductive *Ds* :: *'node* ⇒ *'node* ⇒ *bool*
where *Id*: *valid-node m* ⟹ *Ds m m*
| *Succ*: *succs n* ⊆ {*x*. *Ds m x*} ⟹ ∃ *ns*. *is-path n ns m* ⟹ *Ds m n*

Rule system defined in Theorem 2.1 (greatest fixed point).

coinductive *Di* :: *'node* ⇒ *'node* ⇒ *bool*
where *Id*: *valid-node m* ⟹ *Di m m*
| *Succ*: *succs n* ⊆ {*x*. *Di m x*} ⟹ ∃ *ns*. *is-path n ns m* ⟹ *Di m n*

3.2 Lemmas about maximal paths

lemma *max-path-hd*: *max-path n* (*LCons n' ns*) ⟹ *n* = *n'*
by (*cases rule*: *max-path.cases*) *auto*

lemma *max-path-LCons*: **assumes** *max-path n ns*
obtains *ns'* **where** *ns* = *LCons n ns'*
proof−
from *assms* **have** *ns* ≠ *LNil* **by** (*cases rule*: *max-path.cases*) *auto*
then obtain *n' ns'* **where** *ns* = *LCons n' ns'* **by** (*cases ns*) *auto*
with *max-path-hd assms that* **show** *?thesis* **by** *auto*
qed

lemma *max-path-valid-node*: *max-path n ns* ⟹ *valid-node n*
by (*cases rule*: *max-path.cases*) (*auto simp add*: *succs-def*)

lemma *max-path-no-succs*: **assumes** *max-path n ns*
succs n = {}
shows *ns* = *LCons n LNil*
using *assms* **by** *cases auto*

lemma *max-path-step*: **assumes** *max-path x ns*
succs x ≠ {}
obtains *y ns'* **where** *ns* = *LCons x ns' max-path y ns' y* ∈ *succs x*
using *assms* **by** (*cases rule*: *max-path.cases*) *simp*

lemma *max-path-step-LCons*: **assumes** *max-path x* (*LCons x' ns*)
ns ≠ *LNil*
obtains *y* **where** *x* = *x' max-path y ns y* ∈ *succs x*
using *assms* **by** (*cases rule*: *max-path.cases*) *auto*

lemma *max-path-append*: **assumes** *is-path n ns n'*
max-path n' ns'
shows *max-path n* (*lappend* (*llist-of ns*) *ns'*)

proof −
  from *assms* **have** *Digraph-Basic.path edge-rel n ns n′* **by** *auto*
  from *this assms*(*2*) *edge-rel-def max-path.intros*
  show *?thesis* **by** (*induction rule*: *Digraph-Basic.path.induct*) *auto*
qed

lemma *max-path-end*: **assumes** *is-path n ns n′*
    succs n′ = {}
  shows *max-path n* (*llist-of* (*ns*@[*n′*]))
proof −
  from *assms max-path.intros is-path-valid-node* **have** *max-path n′* (*llist-of* [*n′*]) **by** *auto*
  from *max-path-append*[*OF assms*(*1*) *this*, *unfolded lappend-llist-of-llist-of*] **show** *?thesis* **.**
qed

lemma *max-path-split*: **assumes** *max-path n* (*lappend* (*llist-of ns*) (*LCons n′ ns′*))
  shows *max-path n′* (*LCons n′ ns′*) ∧ *is-path n ns n′*
using *assms*
proof (*induction ns arbitrary*: *n*)
  case *Nil*
  with *max-path-hd*[*of n n′*] *max-path-valid-node* **show** *?case* **by** (*auto intro*: *max-path.intros*)
next
  case (*Cons a ns n*)
  with *max-path-hd* **have** *n* = *a* **by** *auto*
  have *lappend* (*llist-of ns*) (*LCons n′ ns′*) ≠ *LNil* **by** (*cases ns*) *auto*
  with *Cons*(*2*) *max-path-hd* **obtain** *n2* **where** *n2* ∈ *succs n*
   max-path n2 (*lappend* (*llist-of ns*) (*LCons n′ ns′*)) **by** (*cases rule*: *max-path.cases*) *auto*
  with *succs-path-extend*[*of n2 n*] *Cons* ‹*n* = *a*› **show** *?case* **by** *auto*
qed

lemma *max-path-split-elem*: **assumes** *max-path n ns*
    m ∈ *lset ns*
  obtains *ns1 ns2* **where** *is-path n ns1 m max-path m* (*LCons m ns2*)
    ns = *lappend* (*llist-of ns1*) (*LCons m ns2*)
  using *assms lset-split that max-path-split assms* **by** *metis*

Builds a cyclic repetition of the given list.

primcorec *cycle* :: *′a list* ⇒ *′a llist*
  where *cycle ys* = (*case ys of* [] ⇒ *LNil*
    | (*x*#*xs*) ⇒ *LCons x* (*cycle* (*xs*@[*x*])))

lemma *cycle-hd*: **assumes** *cycle xs* = *LCons x ys*
  obtains *xs′* **where** *xs* = *x*#*xs′*
proof (*cases xs*)
  case *Nil*

```
  with cycle.code have cycle xs = LNil by auto
  with assms show ?thesis by auto
next
  case (Cons z zs)
  from cycle.code[of z#zs] Cons that assms show ?thesis by auto
qed

lemma cycle-lset: lset (cycle xs) ⊆ set xs
proof
  fix x
  assume x ∈ lset (cycle xs)
  with lset-split obtain ns1 ns2
  where cycle xs = lappend (llist-of ns1) (LCons x ns2) .
  then show x ∈ set xs
  proof (induction ns1 arbitrary: xs)
    case (Nil xs)
    with cycle-hd[of xs] obtain xs' where xs = x#xs' by auto
    with cycle.code show ?case by auto
  next
    case (Cons y ys xs)
    hence cycle-LCons: cycle xs = LCons y (lappend (llist-of ys) (LCons x ns2))
by auto
    with cycle-hd[of xs] obtain xs' where xs = y#xs' by auto
    with cycle.code[of y#xs'] cycle-LCons
    have cycle (xs'@[y]) = lappend (llist-of ys) (LCons x ns2) by auto
    with Cons(1)[OF this] ‹xs = y#xs'› show ?case by auto
  qed
qed

lemma cycle-infinite: assumes xs ≠ []
  shows ¬ lfinite (cycle xs)
proof
  assume lfinite (cycle xs)
  then obtain xs' where llist-of xs' = cycle xs by (auto simp add: lfinite-eq-range-llist-of)
  with assms show False
  proof (induction xs' arbitrary: xs)
    case Nil
    with cycle.code[of xs] show ?case by (cases xs) auto
  next
    case (Cons a xs')
    with cycle.code[of xs] show ?case by (cases xs) auto
  qed
qed

lemma cycle-lappend-unfold: cycle (xs@ys) = lappend (llist-of xs) (cycle (ys@xs))
proof (induction xs arbitrary: ys)
  case (Cons x xs)
  with cycle.code[of x#xs@ys] Cons[of ys@[x]] show ?case by auto
qed auto
```

```
lemma lfilter-cycle: lfilter P (cycle xs) = cycle (filter P xs)
proof (coinduction arbitrary: xs)
  case Eq-llist
  show ?case
  proof (cases ∃ x∈set xs. P x)
    case True
    with split-list-first-prop obtain x xs1 xs2
      where split: xs = xs1@x#xs2 ∀ x'∈set xs1. ¬ P x' P x by metis
    with cycle-lappend-unfold[of xs1] cycle.code[of x#-] show ?thesis by auto
  next
    case False
    with cycle-lset[of xs] lfilter-False filter-False show ?thesis by auto
  qed
qed

lemma cycle-max-path: is-path n (n#ns) n ⟹ max-path n (cycle (n#ns))
proof (coinduction arbitrary: n ns rule: max-path.coinduct)
  case (max-path n ns)
   from cycle.code[of n#ns] have cycle-unfold: cycle (n#ns) = LCons n (cycle
(ns@[n])) by auto
  show ?case
  proof (cases ns)
    case Nil
     with max-path path-append-conv[of - n []] edge-rel-def have n ∈ succs n by
auto
    with max-path cycle-unfold Nil show ?thesis by auto
  next
    case (Cons y ys)
    with max-path is-path-split[of - [n] y]
    have paths: is-path y (y#ys) n is-path n [n] y by auto
    with path-append-conv[of - n []] edge-rel-def have y ∈ succs n by auto
    from path-append[OF paths] have is-path y (y#ys@[n]) y by simp
    with Cons ‹y ∈ succs n› cycle-unfold show ?thesis by auto
  qed
qed

lemma cycle-max-path-neq-nil: is-path n ns n ⟹ ns ≠ [] ⟹ max-path n (cycle
ns)
using path-cons-conv[of - n] cycle-max-path by (cases ns) auto

lemma lappend-split-eq: assumes lappend (llist-of ns1) (LCons n ns2)
                                  = lappend (llist-of ms1) (LCons m ms2)
                                m ∉ set ns1
                                n ∉ set ms1
                        shows m = n
using assms
proof (induction ns1 arbitrary: ms1)
  case (Nil ms1)
```

then show *?case* **by** (*cases ms1*) *auto*
next
case (*Cons a ns1 ms1*)
then show *?case* **by** (*cases ms1*) *auto*
qed

Given a valid node, this function creates a maximal path starting in that node.

primcorec *ext-max-path* :: *'node* ⇒ *'node llist*
where *ext-max-path x* =
(*if succs x* = {}
then llist-of [*x*]
else LCons x (*ext-max-path* (*SOME y*. *y* ∈ *succs x*)))

lemma *max-path-ext*: *valid-node x* ⟹ *max-path x* (*ext-max-path x*)
proof (*coinduction arbitrary*: *x rule*: *max-path.coinduct*)
case *max-path*
show *?case*
proof (*cases succs x* = {})
let *?y* = *SOME y*. *y* ∈ *succs x*
case *False*
with *someI* **have** *y-props*: *?y* ∈ *succs x* **by** *fast*
with *ext-max-path.code* **have** *ext-max-path x* = *LCons x* (*ext-max-path ?y*) **by** *auto*
with *y-props succs-valid* **show** *?thesis* **by** *auto*
qed (*auto simp add*: *max-path ext-max-path.code*)
qed

lemma *on-max-paths-prev-trivial*: *on-max-paths-prev n n m*
unfolding *on-max-paths-prev-def*
proof *clarify*
fix *ns*
assume *max-path n ns*
with *max-path-LCons* **obtain** *ns'* **where** *ns* = *LCons n ns'* **by** *auto*
then show (∃ *ns1 ns2*. *ns* = *lappend* (*llist-of ns1*) (*LCons n ns2*) ∧ *m* ∉ *set ns1*)
by (*auto intro*: *exI*[*of* - []])
qed

lemma *on-max-paths-not-prev*: **assumes** *on-max-paths n m1*
¬ *on-max-paths-prev n m1 m2*
obtains *ns* **where** *is-path n ns m2 m1* ∉ *set ns*
proof−
from *assms on-max-paths-prev-def* **obtain** *ns1* **where** *ns1-gen*: *max-path n ns1*
∀ *ns2 ns3*. *ns1* = *lappend* (*llist-of ns2*) (*LCons m1 ns3*) ⟶ *m2* ∈ *set ns2* **by** *auto*
with *assms on-max-paths-def* **have** *m1* ∈ *lset ns1* **by** *auto*
with *lset-split-first* **obtain** *ns2 ns3*
where *ns23-gen*: *ns1* = *lappend* (*llist-of ns2*) (*LCons m1 ns3*) *m1* ∉ *set ns2*

by *metis*
with *ns1-gen split-list* **obtain** *ns2a ns2b* **where** *ns2 = ns2a@m2#ns2b* **by** *metis*
with *max-path-split ns1-gen ns23-gen* **have** *is-path n (ns2a@m2#ns2b) m1* **by** *auto*
with *that is-path-split[OF this] ns23-gen ‹ns2 = ns2a@m2#ns2b›* **show** *?thesis* **by** *simp*
qed

3.3 Proof of Theorem 2.1, $\sqsubseteq_{MAX}$ part

First, we prove multiple lemmas that help us prove Theorem 2.1

Proof of the Reflexivity of *on-max-paths* (and therefore $\sqsubseteq_{MAX}$). Also will be part of Observation 5.1.

theorem *on-max-paths-refl*: *on-max-paths x x*
unfolding *on-max-paths-def* **by** *clarify* (*cases rule*: *max-path.cases*, *auto*)

Proof of the Transitivity of *on-max-paths* (and therefore $\sqsubseteq_{MAX}$). Also will be part of Observation 5.1.

theorem *on-max-paths-trans*: **assumes** *on-max-paths x y*
on-max-paths y z
shows *on-max-paths x z*
proof−
{
fix *ns*
assume *max-path x ns*
with *assms on-max-paths-def max-path-split-elem ‹max-path x ns›* **obtain** *ns1 ns2*
where *ns = lappend (llist-of ns1) (LCons y ns2) max-path y (LCons y ns2)* **by** *metis*
with *assms on-max-paths-def* **have** *z ∈ lset ns* **by** *auto*
}
with *assms on-max-paths-def* **show** *?thesis* **by** *auto*
qed

lemma *Ds-valid-node*: **assumes** *Ds m n*
shows *valid-node m valid-node n*
using *assms* **by** (*induction rule*: *Ds.cases*) (*auto simp add*: *is-path-valid-node*)

lemma *Ds-imp-max-paths*: *Ds m n ⟹ on-max-paths n m*
proof (*induction rule*: *Ds.induct*)
next
case (*Succ n m*)
then obtain *ns'* **where** *is-path*: *is-path n ns' m* **by** *auto*
show *?case* **unfolding** *on-max-paths-def*
proof *clarify*
fix *ns*
assume *max-path*: *max-path n ns*
show *m ∈ lset ns*

proof (*cases succs n* = {})
  case *True*
  with *is-path-succs-empty is-path max-path-LCons max-path lset-intros*(*1*)
  show *?thesis* **by** *metis*
next
  case *False*
  with *max-path-step max-path* **obtain** *x ns2*
    where *ns* = *LCons n ns2 max-path x ns2 x* ∈ *succs n* **by** *metis*
  with *Succ on-max-paths-def* **show** *?thesis* **by** *auto*
qed
qed
qed (*simp add*: *on-max-paths-refl*)

This function constructs a maximal path that starts in the node given as second argument and that doesn't contain the node given as first argument. Precondition: ¬ *Ds n x*.

primcorec *avoid-path* :: *'node* ⇒ *'node* ⇒ *'node llist*
  where *avoid-path n x* =
    (*if succs x* = {}
     then llist-of [*x*]
     else LCons x (*avoid-path n* (*SOME y*. *y* ∈ *succs x* ∧ ¬ *Ds n y*)))

lemma *not-Ds-cont*: ¬ *Ds m n* ⟹ *succs n* ≠ {} ⟹ ∃ *x*. *x* ∈ *succs n* ∧ ¬ *Ds m x*
proof−
  have *not-Ds-cont*: ∀ *x*. *x* ∈ *succs n* ⟶ *Ds m x* ⟹ *succs n* ≠ {} ⟹ *Ds m n*
  proof
    assume ∀ *x*. *x* ∈ *succs n* ⟶ *Ds m x succs n* ≠ {}
    then obtain *x* **where** *x-gen*: *Ds m x x* ∈ *succs n* **by** *auto*
    from *this path0*[*of edge-rel*] **obtain** *ns* **where** *is-path x ns m* **by** *cases blast*+
    with *x-gen succs-path-extend* **show** ∃ *ns*. *is-path n ns m* **by** *blast*
  qed *auto*
  then show ¬ *Ds m n* ⟹ *succs n* ≠ {} ⟹ ∃ *x*. *x* ∈ *succs n* ∧ ¬ *Ds m x* **by** *auto*
qed

lemma *not-Ds-max-path*: ¬ *Ds n x* ⟹ *valid-node x* ⟹ *max-path x* (*avoid-path n x*)
proof (*coinduction arbitrary*: *x rule*: *max-path.coinduct*)
  case (*max-path x*)
  then show *?case*
  proof (*cases succs x* = {})
    case *True*
    with *max-path avoid-path.code* **show** *?thesis* **by** *auto*
  next
    let *?y* = *SOME y*. *y* ∈ *succs x* ∧ ¬ *Ds n y*
    case *False*
    with *avoid-path.code* **have** *path*: *avoid-path n x* = *LCons x* (*avoid-path n ?y*) **by** *auto*

from *max-path not-Ds-cont*[*THEN someI-ex*] *False* **have** *?y* ∈ *succs x* ¬ *Ds n ?y* **by** *auto*
with *path succs-def* **show** *?thesis* **by** *auto*
qed
qed

lemma *not-Ds-avoid-n*: ¬ *Ds n x* ⟹ *valid-node x* ⟹ *n* ∉ *lset* (*avoid-path n x*)
proof (*rule ccontr*)
assume *assm*: ¬ *Ds n x valid-node x* ¬ *n* ∉ *lset* (*avoid-path n x*)
with *lset-split*[*of n avoid-path n x*] **obtain** *ns1 ns2*
where *avoid-path n x* = *lappend* (*llist-of ns1*) (*LCons n ns2*) **by** *auto*
with *assm* **show** *False*
proof (*induction ns1 arbitrary*: *x*)
case (*Nil x*)
with *Ds.intros avoid-path.code* **show** *?case* **by** (*cases succs x* = {}) *auto*
next
case (*Cons a ns1 x*)
hence *path*: *avoid-path n x* = *LCons a* (*lappend* (*llist-of ns1*) (*LCons n ns2*)) **by** *auto*
with *avoid-path.code* **have** *cont*: *succs x* ≠ {} **by** (*cases ns1*) *auto*
let *?y* = *SOME y*. *y* ∈ *succs x* ∧ ¬ *Ds n y*
from *Cons avoid-path.code cont* **have** *avoid-path n x* = *LCons x* (*avoid-path n ?y*) **by** *auto*
with *path* **have** *path'*: *avoid-path n ?y* = *lappend* (*llist-of ns1*) (*LCons n ns2*) **by** *auto*
from *Cons not-Ds-cont*[*THEN someI-ex*] *cont* **have** *?y* ∈ *succs x* ¬ *Ds n ?y* **by** *auto*
with *succs-def Cons*(*1*)[*OF this*(*2*)] *path'* **show** *?thesis* **by** *auto*
qed
qed

lemma *max-paths-imp-Ds*: *on-max-paths x n* ⟹ *valid-node x* ⟹ *Ds n x*
proof (*rule ccontr*)
assume *on-max-paths x n valid-node x* ¬ *Ds n x*
with *not-Ds-max-path on-max-paths-def not-Ds-avoid-n* **show** *False* **by** *blast*
qed

Proof of the $\sqsubseteq_{MAX}$ part of Theorem 2.1.

theorem *Ds-max-paths*: *Ds n x* ⟷ *on-max-paths x n* ∧ *valid-node x*
using *max-paths-imp-Ds Ds-imp-max-paths Ds-valid-node* **by** *auto*

lemma *on-max-paths-ex-path*: *on-max-paths n m* ⟹ *valid-node n* ⟹ ∃ *ns*. *is-path n ns m*
using *Ds-max-paths Ds.cases path0* **by** *metis*

lemma *ntscd-cond-succ*: **assumes** ¬ *on-max-paths p n*
x ∈ *succs p*
on-max-paths x n
shows *ntscd p n*

unfolding *ntscd-def*
proof
  from *assms on-max-paths-def* **obtain** *ns* **where** *ns-gen*: *max-path p ns n* ∉ *lset ns* **by** *auto*
  with *assms max-path-step* **obtain** *x2 ns'*
  where *max-path x2 ns' ns* = *LCons p ns' x2* ∈ *succs p* **by** *blast*
  with *ns-gen on-max-paths-def* **show** ∃ *x2*∈*succs p*. ¬ *on-max-paths x2 n* **by** *auto*
qed (*insert assms*, *blast*)

This function itself is never used in this theory. It is only defined to use the resulting induction rule.

function *ntscd-steps* :: *'node* ⇒ *'node list* ⇒ *'node list*
  where *ntscd-steps p* (*n*#*ns*) = (*if n* = *p then* (*n*#*ns*)
                    else ntscd-steps p (*dropWhile* (λ*m*. *on-max-paths m n*) (*n*#*ns*)))
    | *ntscd-steps p* [] = []
proof−
  fix *Q x*
  assume (⋀*p n ns*. (*x*::*'node* × *'node list*) = (*p*, *n* # *ns*) ⟹ *Q*) (⋀*p*. *x* = (*p*, []) ⟹ *Q*)
  thus *Q* **by** (*cases x*, *cases snd x*) *auto*
qed *auto*
termination
proof (*relation measure* (*length o snd*))
  fix *p n ns*
  from *on-max-paths-refl length-dropWhile-le*[*of* λ*m*. *on-max-paths m n ns*]
  show ((*p*::*'node*, *dropWhile* (λ*m*. *on-max-paths m n*) (*n* # *ns*)), (*p*, *n* # *ns*))
      ∈ *measure* (*length* ∘ *snd*) **by** *auto*
qed *auto*

lemma *ntscd-rtranclpI'*: **assumes** *is-path p ns n*
      ∀ *m*∈*set* (*n*#*rev ns*). *p* ≠ *m* ⟶ ¬ *on-max-paths p m*
    shows *ntscd*** *p n*
using *assms*
proof (*induction p n*#*rev ns arbitrary*: *n ns rule*: *ntscd-steps.induct*)
  case (*1 p n ns*)
  show *?case*
  proof (*cases n* = *p*)
    let *?ds* = *dropWhile* (λ*m*. *on-max-paths m n*) (*n*#*rev ns*)
    let *?ts* = *takeWhile* (λ*m*. *on-max-paths m n*) (*n*#*rev ns*)
    from *on-max-paths-refl* **have** *?ts* ≠ [] **by** *auto*
    then obtain *ts-h ts'* **where** *ts-split*: *?ts* = *ts-h*#*ts'* **by** (*cases ?ts*) *auto*
    case *False*
    with *1* **have** *not-max*: ¬ *on-max-paths p n* **by** *simp*
    from *1*(*2*) *path-rev-last last-in-set*[*of n* # *rev ns*] **have** *p* ∈ *set* (*n*#*rev ns*) **by** *auto*
    with *1 dropWhile-eq-Nil-conv not-max* **have** *?ds* ≠ [] **by** *auto*
    then obtain *n' ns-r* **where** *?ds* = *n'*#*rev* (*rev ns-r*) **by** (*cases ?ds*) *auto*
    then obtain *ns'* **where** *ds-split*: *?ds* = *n'*#*rev ns'* **by** *blast*

with *takeWhile-dropWhile-id* **have** *split*: *n#rev ns = ?ts@n'#rev ns'* **by** *metis*
with *ts-split* **have** *rev ns = ts'@n'#rev ns'* **by** *auto*
with *rev-rev-ident[of ns]* **have** *ns = ns'@n'#rev ts'* **by** *auto*
with *1(2) is-path-split[of - ns']*
have *split-path*: *is-path p ns' n' is-path n' (n'#rev ts') n* **by** *auto*
from *split* **have** *set (n'#rev ns')* ⊆ *set (n#rev ns)* **by** *auto*
with *1* **have** ∀ *m*∈*set (n' # rev ns'). p* ≠ *m* ⟶ ¬ *on-max-paths p m* **by** *auto*
with *1 False ds-split split-path* **have** *ntscd** p n'* **by** *auto*
from *ds-split[unfolded dropWhile-eq-Cons-conv]* **have** ¬ *on-max-paths n' n* **by** *auto*
obtain *x2* **where** *on-max-paths x2 n x2* ∈ *succs n'*
proof (*cases rev ts'*)
case *Nil*
with *split-path path-last-is-edge[of - - [n']] edge-rel-def* **have** *n* ∈ *succs n'* **by** *auto*
with *that on-max-paths-refl* **show** *?thesis* **by** *auto*
next
case (*Cons t' ts''*)
with *split-path* **have** *is-path n' (n'#t'#ts'') n* **by** *auto*
with *is-path-split[of - [n']]* **have** *is-path n' [n'] t'* **by** *auto*
with *path-last-is-edge[of - - [n']] edge-rel-def* **have** *t'* ∈ *succs n'* **by** *auto*
from *ts-split Cons* **have** *t'* ∈ *set ?ts* **by** *auto*
hence *on-max-paths t' n* **by** (*auto dest*: *set-takeWhileD*)
with ‹*t'* ∈ *succs n'*› *that* **show** *?thesis* **by** *auto*
qed
with ‹¬ *on-max-paths n' n*› *ntscd-cond-succ* **have** *ntscd n' n* **by** *auto*
with ‹*ntscd** p n'*› **show** *?thesis* **by** *auto*
qed *auto*
qed

lemma *ntscd-rtranclpI*: **assumes** *is-path p ns n*
∀ *m*∈*set ns* ∪ {*n*}. *p* ≠ *m* ⟶ ¬ *on-max-paths p m*
shows *ntscd** p n*
using *assms ntscd-rtranclpI'* **by** *auto*

3.4 Lemmas about sink paths

lemma *on-ext-pathsE*: *on-ext-paths x n* ⟹ *is-path x ns n'*
⟹ (∃ *ns' n''*. *is-path n' ns' n''* ∧ *n* ∈ *set (ns@ns')* ∪ {*n''*})
using *on-ext-paths-def* **by** *auto*

lemma *sink-node-reachable*:
assumes *sink-node n is-path n ns m*
shows *m* ∈ *scc-of edge-rel n*
using *assms*
proof (*induction ns arbitrary*: *m rule*: *rev-induct*)
case (*snoc x xs m*)
hence *x-rel*: *x* ∈ *scc-of edge-rel n (x, m)* ∈ *edge-rel* **unfolding** *path-append-conv* **by** *auto*

show *?case*
proof (*rule ccontr*)
assume $m \notin$ *scc-of edge-rel n*
with *scc-of-unique* **have** *scc-change*: *scc-of edge-rel m* $\neq$ *scc-of edge-rel n* **by** *auto*
from *x-rel* **have** (*scc-of edge-rel x*, *scc-of edge-rel m*)
$\in$ (λ(*n1*, *n2*). (*scc-of edge-rel n1*, *scc-of edge-rel n2*)) ' *edge-rel* **by** *auto*
with *x-rel scc-change cond-edges-def*
have (*scc-of edge-rel n*, *scc-of edge-rel m*) $\in$ *cond-edges* **by** (*auto dest*!: *scc-of-unique*)
with *assms sink-node-def* **show** *False* **by** *auto*
qed
qed *simp*

lemma *sink-node-path*: **assumes** *sink-node n*
is-path n ns y
shows $\forall$ *m*$\in$*set* (*ns*@[*y*]). $m \in$ *scc-of edge-rel n*
proof
fix *m*
assume *in-set*: $m \in$ *set* (*ns*@[*y*])
show $m \in$ *scc-of edge-rel n*
proof (*cases* $m = y$)
case *True*
with *assms sink-node-reachable* **show** *?thesis* **by** *blast*
next
case *False*
with *in-set* **have** $m \in$ *set ns* **by** *auto*
with *path-split-elem assms sink-node-reachable* **show** *?thesis* **by** *blast*
qed
qed

lemma *cond-nodes-edges*: *cond-edges* $\subseteq$ *cond-nodes* $\times$ *cond-nodes*
unfolding *cond-edges-def cond-nodes-def edge-rel-def succs-def* **by** *auto*

lemma *cond-edge-impl-path*:
assumes (*a*, *b*) $\in$ *cond-edges*
assumes ($\varphi_a \in a$)
assumes ($\varphi_b \in b$)
shows (φ_a, φ_b) $\in$ *edge-rel**
unfolding *cond-edges-def*
proof –
from *assms*(*1*)
obtain *x y* **where** *x-y-props*:
(x, y) $\in$ *edge-rel*
$a =$ *scc-of edge-rel x*
$b =$ *scc-of edge-rel y*
unfolding *cond-edges-def* **by** *auto*
hence $x \in a$ $y \in b$ **by** *auto*

with *assms*(*2*) *x-y-props*(*2*)

have $(\varphi_a, x) \in$ *edge-rel** **by** (*meson is-scc-connected scc-of-is-scc*)
moreover with *assms*(*3*) *x-y-props*(*3*) ‹$y \in b$›
have $(y, \varphi_b) \in$ *edge-rel** **by** (*meson is-scc-connected scc-of-is-scc*)
ultimately
show $(\varphi_a, \varphi_b) \in$ *edge-rel** **using** *x-y-props*(*1*)
by (*meson rtrancl.rtrancl-into-rtrancl rtrancl-trans*)
qed

lemma *path-in-cond-impl-path*:
assumes $(a, b) \in$ *cond-edges*$^+$
assumes $(\varphi_a \in a)$
assumes $(\varphi_b \in b)$
shows $(\varphi_a, \varphi_b) \in$ *edge-rel**
using *assms*
proof (*induction arbitrary*: φ_b *rule*:*trancl-induct*)
case *step*
fix y z φ_b
assume $(y, z) \in$ *cond-edges*

hence *is-scc edge-rel y* **unfolding** *cond-edges-def* **by** *auto*
hence $\exists \varphi_y.\ \varphi_y \in y$ **using** *scc-non-empty′* **by** *auto*
then obtain φ_y **where** φ_y*-in-y*: $\varphi_y \in y$ **by** *auto*

assume φ_b*-elem*: $\varphi_b \in z$
assume $\bigwedge\varphi_b.\ \varphi_a \in a \Longrightarrow \varphi_b \in y \Longrightarrow (\varphi_a, \varphi_b) \in$ *edge-rel**
with *assms*(*2*) φ_y*-in-y*
have φ_a*-to-*φ_y: $(\varphi_a, \varphi_y) \in$ *edge-rel** **using** *cond-edge-impl-path* **by** *auto*

from φ_b*-elem* φ_y*-in-y* ‹$(y, z) \in$ *cond-edges*›
have $(\varphi_y, \varphi_b) \in$ *edge-rel** **using** *cond-edge-impl-path* **by** *auto*
with φ_a*-to-*φ_y
show $(\varphi_a, \varphi_b) \in$ *edge-rel** **by** *auto*
next
case (*base* φ_b y)
thus *?case*
using *assms*(*2*) *cond-edge-impl-path* **by** *blast*
qed

lemma *cond-edges-acyclic*: *acyclic cond-edges*
proof (*rule acyclicI*, *rule allI*, *rule ccontr*, *clarify*)
fix x

Assume there is a cycle in the condensation graph.

assume *cyclic*: $(x, x) \in$ *cond-edges*$^+$
have *nonrefl*: $(x, x) \notin$ *cond-edges* **unfolding** *cond-edges-def* **by** *auto*

from *this cyclic*
obtain b **where** *b-on-path*: $(x, b) \in$ *cond-edges* $(b, x) \in$ *cond-edges*$^+$
by (*meson converse-tranclE*)

hence $x \in$ *cond-nodes* $b \in$ *cond-nodes* **using** *cond-nodes-edges* **by** *auto*
hence *nodes-are-scc*: *is-scc edge-rel x is-scc edge-rel b*
using *scc-of-is-scc* **unfolding** *cond-nodes-def* **by** *auto*

have $\exists \varphi_x. \varphi_x \in x \ \exists \varphi_b. \varphi_b \in b$ **using** *nodes-are-scc scc-non-empty′ ex-in-conv* **by** *auto*
then obtain φ_x φ_b **where** *φxb-elem*: $\varphi_x \in x$ $\varphi_b \in b$ **by** *metis*
with *nodes-are-scc(1) b-on-path path-in-cond-impl-path cond-edge-impl-path φxb-elem(2)*
have $\varphi_b \in x$
by − (*rule is-scc-closed*)

with *nodes-are-scc φxb-elem*
have $x = b$ **using** *is-scc-unique*[*of edge-rel*] **by** *simp*
hence $(x, x) \in$ *cond-edges* **using** *b-on-path* **by** *simp*
with *nonrefl*
show *False* **by** *simp*
qed

lemma *finite-CFG-impl-finite-condensation*: **assumes** *finite* (*Collect valid-node*)
shows *finite cond-edges*
proof−
from *edge-rel-def succs-valid* **have** *edge-rel* ⊆ *Collect valid-node* × *Collect valid-node* **by** *auto*
with *assms finite-subset* **have** *finite edge-rel* **by** *auto*
with *finite-Diff finite-imageI cond-edges-def* **show** *?thesis* **by** *auto*
qed

For each node, we can find a sink that is reachable from it.

lemma *leafE*:
assumes *valid-node n* **and** *finite cond-edges*
shows $\exists$ *sink*. (*scc-of edge-rel n*, *sink*) ∈ *cond-edges** ∧ ¬(∃ *out*. (*sink*, *out*) ∈ *cond-edges*)
proof −
define *reachable-cond* **where** [*simp*]:
reachable-cond ≡ {(*m2*, *m1*). (*scc-of edge-rel n*, *m1*) ∈ *cond-edges** ∧ (*m1*, *m2*) ∈ *cond-edges*$^+$}
show *?thesis*
proof (*rule wfE-min*[*of reachable-cond - fst ‘ reachable-cond* ∪ {*scc-of edge-rel n*}])
have *subset*: *reachable-cond* ⊆ *converse* (*cond-edges*$^+$) **by** *auto*
hence *finite reachable-cond* **using** *assms* **by** (*simp add*: *finite-subset*)
thus *wf* (*reachable-cond*)
by (*meson assms acyclic-converse cond-edges-acyclic cyclic-subset finite-acyclic-wf subset wf-acyclic wf-trancl*)
next
from *assms(1)*
show *scc-of edge-rel n* ∈ *fst ‘ reachable-cond* ∪ {*scc-of edge-rel n*} **by** *auto*
next

```
    fix sink
    assume sink1: sink ∈ fst ‘ reachable-cond ∪ {scc-of edge-rel n}
    assume sink2: scc ∉ fst ‘ reachable-cond ∪ {scc-of edge-rel n}
              if (scc, sink) ∈ reachable-cond for scc
    have left: (scc-of edge-rel n, sink) ∈ cond-edges* using sink1 by auto
    {
      fix out
      have (sink, out) ∉ cond-edges
      proof (rule ccontr, simp)
        assume (sink, out) ∈ cond-edges
        with left
        have (out, sink) ∈ reachable-cond
          by auto
        with sink2
        show False by auto
      qed
    }
    hence right: ¬(∃ out. (sink, out) ∈ cond-edges) by auto
    with left show ?thesis by −(rule exI, rule conjI)
  qed
qed

lemma path-sink-path-append:
  assumes is-path n ns n' and sink-path n' ns'
  shows sink-path n (lappend (llist-of ns) ns')
using assms sink-path-def max-path-append by auto

lemma sink-path-exists: assumes valid-node n and finite (Collect valid-node)
  obtains ns where sink-path n ns
proof −
  from assms finite-CFG-impl-finite-condensation obtain sink
     where sink: (scc-of edge-rel n, sink) ∈ cond-edges* ¬(∃ out. (sink, out) ∈
cond-edges)
    by (auto dest: leafE)
   with assms(1) have sink-scc: sink ∈ cond-nodes unfolding cond-nodes-def
cond-edges-def
  proof (cases sink = scc-of edge-rel n)
    case False
    with assms(1) sink(1)
    have (scc-of edge-rel n, sink) ∈ cond-edges+
      unfolding cond-edges-def by (metis rtranclD)
    from this edge-impl-valid-target cond-edges-def
    show sink ∈ {scc-of edge-rel n |n. valid-node n} by cases auto
  qed auto

 with node-in-scc-of-node obtain n' where n': n' ∈ sink unfolding cond-nodes-def
by fastforce
  have n: n ∈ scc-of edge-rel n by (rule node-in-scc-of-node)
```

```
  obtain ns where ns: is-path n ns n′
  proof (−, cases (scc-of edge-rel n) = sink)
    case True
    thus ?thesis
      using scc-path that n′ assms(1) by metis
  next
    case False
     thus ?thesis using n n′ edge-rel-rtrancl-path path-in-cond-impl-path sink(1)
assms(1) that
      by (metis rtrancl-eq-or-trancl)
  qed

  from ns n′ sink-scc
 have scc: scc-of edge-rel n′ = sink using scc-of-unique unfolding cond-nodes-def
by fast
 with sink ns have sink-node: sink-node n′ unfolding sink-path-def sink-node-def
by fast
  show ?thesis
  proof (cases succs n′ = {})
    case True
   with max-path-end[OF ns] sink-node sink-path-def that show ?thesis by fastforce
  next
    case False
   from scc-path is-path-valid-node scc ns have sink ⊆ Collect valid-node by blast
    with assms finite-subset scc have finite sink sink ⊆ scc-of edge-rel n′ by auto
    then obtain ns2 where ns2-gen: is-path n′ ns2 n′ ∀ m ∈ sink − {n′}. m ∈
set ns2
    proof (induction arbitrary: thesis rule: finite-subset-induct)
      case empty
      with ns is-path-valid-node path0 show ?case by fast
    next
      case (insert m F)
      with scc-path is-path-valid-node ns obtain ns1 where path1: is-path n′ ns1
m by blast
      with insert scc-of-unique have n′ ∈ scc-of edge-rel m by fastforce
       with scc-path is-path-valid-node path1 obtain ns2 where path2: is-path m
ns2 n′ by blast
       from insert obtain ns3 where path3: is-path n′ ns3 n′ ∀ m∈F−{n′}. m ∈
set ns3 by auto
      with path1 path2 path-append have cycle-path: is-path n′ (ns1@ns2@ns3) n′
by auto
      {
        assume m ≠ n′
        with path2 is-path-Cons have m ∈ set ns2 by (cases ns2) auto
      }
      with path3 insert cycle-path show ?case by fastforce
    qed
    from False obtain n2 where n2-gen: n2 ∈ succs n′ by auto
    with succs-path sink-node-reachable sink-node scc-of-unique
```

have *n′ ∈ scc-of edge-rel n2* **by** *fastforce*
with *scc-path n2-gen succs-valid* **obtain** *ns3* **where** *is-path n2 ns3 n′* **by** *blast*
with *ns2-gen succs-path-extend path-append n2-gen*
have *full-path*: *is-path n′ (n′#ns3@ns2) n′ ∀ m ∈ sink. m ∈ set (n′#ns3@ns2)*
by *auto*
with *cycle-max-path-neq-nil* **have** *max-path*: *max-path n′ (cycle (n′#ns3@ns2))*
by *auto*
from *cycle.code[of n′#-]* **have** *cycle-n′*: *n′ ∈ lset (cycle (n′#ns3@ns2))* **by**
auto
{
fix *n′′*
assume *n′′ ∈ scc-of edge-rel n′*
with *scc full-path*
have *filter (λx. x = n′′) (n′#ns3@ns2) ≠ []* **by** *(auto simp add: filter-empty-conv)*
with *lfilter-cycle cycle-infinite*
have *¬ lfinite (lfilter (λx. x = n′′) (cycle (n′#ns3@ns2)))* **by** *metis*
}
with *max-path sink-node ns cycle-n′ sink-path-def path-sink-path-append that*
show *?thesis* **by** *blast*
qed
qed

Equivalence of *on-ext-paths* and *on-sink-paths*. This allows us to use the easier to handle *on-ext-paths* in proofs and then convert them to *on-sink-paths*.

lemma *on-sink-ext-paths-equiv*: **assumes** *finite (Collect valid-node)*
shows *on-ext-paths x n ⟷ on-sink-paths x n*
proof
assume *ext-paths*: *on-ext-paths x n*
{
fix *ns m*
assume *assm*: *sink-path x ns*
with *sink-path-def* **obtain** *n′* **where** *n′-gen*: *max-path x ns n′ ∈ lset ns*
sink-node n′
succs n′ ≠ {} ⟶ (∀ n′′ ∈ scc-of edge-rel n′. ¬ lfinite (lfilter (λx. x = n′′)
ns)) **by** *auto*
with *max-path-split-elem* **obtain** *ns1 ns2*
where *ns-split*: *ns = lappend (llist-of ns1) (LCons n′ ns2) is-path x ns1 n′*
by *metis*
have *n ∈ lset ns*
proof *(cases n ∈ scc-of edge-rel n′)*
case *True*
show *?thesis*
proof *(cases succs n′ = {})*
case *True*
with *‹n ∈ scc-of edge-rel n′› scc-path ns-split is-path-valid-node*
obtain *ns′* **where** *is-path n′ ns′ n* **by** *blast*
with *is-path-succs-empty True n′-gen* **show** *?thesis* **by** *auto*
next
case *False*

with *n′-gen* ‹*n* ∈ *scc-of edge-rel n′*› **have** (*lfilter* (λ*x*. *x* = *n*) *ns*) ≠ *LNil* **by** *auto*
with *lfilter-eq-LNil* **show** *?thesis* **by** *auto*
qed
next
case *False*
with *ext-paths on-ext-paths-def ns-split* **obtain** *ns′ n″*
where *is-path n′ ns′ n″ n* ∈ *set* (*ns1*@*ns′*@[*n″*]) **by** *blast*
with *sink-node-path n′-gen False ns-split* **show** *?thesis* **by** *auto*
qed
}
with *on-sink-paths-def* **show** *on-sink-paths x n* **by** *auto*
next
assume *sink-paths*: *on-sink-paths x n*
show *on-ext-paths x n* **unfolding** *on-ext-paths-def*
proof (*clarify del*: *conjE*)
fix *ns n′*
assume *path1*: *is-path x ns n′*
with *sink-path-exists assms finite-CFG-impl-finite-condensation is-path-valid-node*[*OF this*]
obtain *ns1* **where** *sink-ext*: *sink-path n′ ns1* **by** *auto*
with *path-sink-path-append*[*OF path1*] **have** *sink-path x* (*lappend* (*llist-of ns*) *ns1*) **by** *auto*
with *sink-paths on-sink-paths-def* **have** *n-elem*: *n* ∈ *lset* (*lappend* (*llist-of ns*) *ns1*) **by** *auto*
show ∃ *ns′ n″*. *is-path n′ ns′ n″* ∧ *n* ∈ *set* (*ns* @ *ns′* @ [*n″*])
proof (*cases n* ∈ *set ns*)
case *True*
with *is-path-valid-node path1 path0* **show** *?thesis* **by** *fastforce*
next
case *False*
with *n-elem* **have** *n* ∈ *lset ns1* **by** *auto*
with *sink-ext sink-path-def max-path-split lset-split*
obtain *ns2* **where** *n-ext*: *is-path n′ ns2 n* **by** *metis*
then show *?thesis* **by** *auto*
qed
qed
qed

3.5 Proof of Theorem 2.1, $\sqsubseteq_{SINK}$ part

First, we prove multiple lemmas that help us prove Theorem 2.1

lemma *on-ext-paths-ex*: *on-ext-paths x n* ⟹ *valid-node x* ⟹ ∃ *ns*. *is-path x ns n*
using *path0 on-ext-pathsE path-split-elem2* **by** (*metis append-Nil*)

Proof of the Reflexivity of *on-sink-paths* (and therefore $\sqsubseteq_{SINK}$). Part of Observation 5.1.

theorem *on-sink-paths-refl*: *on-sink-paths x x*

proof−
{
fix *ns*
assume *sink-path x ns*
with *sink-path-def max-path-LCons* **obtain** *ns′* **where** *ns* = *LCons x ns′* **by** *blast*
then have *x* ∈ *lset ns* **by** *auto*
}
with *on-sink-paths-def* **show** *?thesis* **by** *auto*
qed

lemma *on-ext-paths-trans*: **assumes** *on-ext-paths x y*
on-ext-paths y z
shows *on-ext-paths x z*
unfolding *on-ext-paths-def*
proof (*clarify del*: *conjE*)
fix *ns n′*
assume *path*: *is-path x ns n′*
with *assms on-ext-paths-def* **obtain** *ns1 n1′*
where *ext1*: *is-path n′ ns1 n1′ y* ∈ *set* (*ns*@*ns1*@[*n1′*]) **by** *blast*
show ∃ *ns′ n″*. *is-path n′ ns′ n″* ∧ *z* ∈ *set* (*ns* @ *ns′* @ [*n″*])
proof (*cases y* = *n1′*)
case *True*
with *on-ext-paths-ex*[*OF assms*(*2*)] *ext1 is-path-valid-node* **obtain** *ns2*
where *is-path y ns2 z* **by** *auto*
with *ext1 True path-append* **have** *is-path n′* (*ns1*@*ns2*) *z z* ∈ *set* (*ns*@*ns1*@*ns2*@[*z*]) **by** *auto*
thus *?thesis* **by** *auto*
next
case *False*
with *ext1* **have** *y* ∈ *set* (*ns*@*ns1*) **by** *auto*
with *path-split-elem path-append*[*OF path ext1*(*1*)] **obtain** *ys1 ys2*
where *y-split*: *ns*@*ns1* = *ys1*@*y*#*ys2 is-path y* (*y*#*ys2*) *n1′* **by** *blast*
from *on-ext-pathsE*[*OF assms*(*2*) *this*(*2*)] **obtain** *ns2 n2′*
where *is-path n1′ ns2 n2′ z* ∈ *set* ((*y*#*ys2*)@*ns2*@[*n2′*]) **by** *auto*
with *ext1 path-append y-split*
have *path2*: *is-path n′* (*ns1*@*ns2*) *n2′ z* ∈ *set* ((*ys1*@*y*#*ys2*)@*ns2*@[*n2′*]) **by** *auto*
from *this*[*folded y-split*(*1*)] **have** *z* ∈ *set* (*ns*@(*ns1*@*ns2*)@[*n2′*]) **by** *auto*
with *path2* **show** *?thesis* **by** *blast*
qed
qed

Proof of the Transitivity of *on-sink-paths* (and therefore $\sqsubseteq_{SINK}$). Also will be part of Observation 5.1.

theorem *on-sink-paths-trans*: **assumes** *finite* (*Collect valid-node*)
on-sink-paths x y
on-sink-paths y z
shows *on-sink-paths x z*

using *assms on-sink-ext-paths-equiv on-ext-paths-trans* **by** *blast*

lemma *Di-ex-path*: *Di n x* $\Longrightarrow$ $\exists$ *ns*. *is-path x ns n*
by (*cases rule*: *Di.cases*) (*auto intro*: *path0*)

lemma *Di-imp-ext-paths*: **assumes** *Di m n*
shows *on-ext-paths n m*
unfolding *on-ext-paths-def*
proof (*clarify del*: *conjE*)
fix *ns n'*
assume *is-path*: *is-path n ns n'*
from *this assms* **show** $\exists$ *ns' n''*. *is-path n' ns' n''* $\wedge$ *m* $\in$ *set* (*ns* @ *ns'* @ [*n''*])
proof (*induction ns arbitrary*: *n*)
case (*Nil n*)
with *Di-ex-path*[*of m n'*] *path0* **show** *?case* **by** *auto*
next
case (*Cons a ns n*)
with *is-path-Cons* **obtain** *x* **where** *x-gen*: *n* = *a x* $\in$ *succs n is-path x ns n'*
by *blast*
from *Cons*(*3*) **show** *?case*
proof *cases*
case *Id*
with *x-gen path0*[*of edge-rel n'*] *is-path-valid-node*[*of x*] **show** *?thesis* **by**
fastforce
next
case *Succ*
with *x-gen Cons*(*1*)[*of x*] **show** *?thesis* **by** *auto*
qed
qed
qed

lemma *ext-paths-imp-Di*: *on-ext-paths x n* $\Longrightarrow$ *valid-node x* $\Longrightarrow$ *Di n x*
proof (*coinduction arbitrary*: *x rule*: *Di.coinduct*)
case (*Di x*)
show *?case*
proof (*cases n* = *x*)
case *False*
from *Di on-ext-paths-ex* **have** *path-ex*: $\exists$ *ns*. *is-path x ns n* **by** *auto*
have $\bigwedge$*y*. *y* $\in$ *succs x* $\Longrightarrow$ *on-ext-paths y n* **unfolding** *on-ext-paths-def*
proof (*clarify del*: *conjE*)
fix *y ns n'*
assume *y* $\in$ *succs x is-path y ns n'*
with *succs-path-extend* **have** *is-path x* (*x*#*ns*) *n'* **by** *auto*
from *Di on-ext-pathsE*[*OF Di*(*1*) *this*] *False*
show $\exists$ *ns' n''*. *is-path n' ns' n''* $\wedge$ *n* $\in$ *set* (*ns* @ *ns'* @ [*n''*]) **by** *auto*
qed
with *succs-def path-ex Di* **show** *?thesis* **by** *auto*
qed (*simp add*: *Di*)
qed

lemma *Di-ext-paths*: **assumes** *valid-node x*
shows *Di n x* ⟷ *on-ext-paths x n*
using *Di-imp-ext-paths ext-paths-imp-Di assms* **by** *auto*

Proof of the $\sqsubseteq_{SINK}$ part of Theorem 2.1.

theorem *Di-sink-paths*: **assumes** *valid-node x*
finite (*Collect valid-node*)
shows *Di n x* ⟷ *on-sink-paths x n*
using *Di-ext-paths on-sink-ext-paths-equiv assms* **by** *auto*

Noted in Section 2.2 directly after Definition 2.1.

theorem *on-max-paths-implies-on-sink-paths*: **assumes** *on-max-paths n m*
shows *on-sink-paths n m*
using *on-max-paths-def on-sink-paths-def sink-path-def assms* **by** *auto*

Definition 2.3.

definition *dod* :: *'node* ⇒ *'node* ⇒ *'node* ⇒ *bool*
where *dod n m1 m2* == *m1* ≠ *m2* ∧ *n* ≠ *m1* ∧ *n* ≠ *m2* ∧ *on-max-paths n m1* ∧ *on-max-paths n m2*
∧ (∃ *x1*∈*succs n*. *on-max-paths-prev x1 m1 m2*)
∧ (∃ *x2*∈*succs n*. *on-max-paths-prev x2 m2 m1*)

4 Timing Sensitive Control Dependence

4.1 Basic Properties of Timing Sensitive Control Dependence

Part of Definition 3.1: *at-pos k ns n* = $m \in^k ns$

definition *at-pos* :: *nat* ⇒ *'node llist* ⇒ *'node* ⇒ *bool*
where *at-pos k ns n* == *llength ns* > *k* ∧ *lnth ns k* = *n*

Part of Definition 3.1: *at-pos-first k ns n* = $m \in^k{}_{FIRST}\ ns$

definition *at-pos-first* :: *nat* ⇒ *'node llist* ⇒ *'node* ⇒ *bool*
where *at-pos-first k ns n* == *llength ns* > *k* ∧ *lnth ns k* = *n* ∧ (∀ *k'*<*k*. *lnth ns k'* ≠ *n*)

Part of Definition 3.2 ($\sqsubseteq^k{}_{TIME[FIRST]}$)

definition *on-max-paths-pos-k-first* :: *'node* ⇒ *nat* ⇒ *'node* ⇒ *bool*
where *on-max-paths-pos-k-first n k m* == ∀ *ns*. *max-path n ns* ⟶ *at-pos-first k ns m*

Part of Definition 3.2 ($\sqsubseteq_{TIME[FIRST]}$)

definition *on-max-paths-pos-first* :: *'node* ⇒ *'node* ⇒ *bool*
where *on-max-paths-pos-first n m* == ∃ *k*. *on-max-paths-pos-k-first n k m*

lemma *at-pos-succ*: *at-pos* (*k+1*) (*LCons n ns*) *m* ⟷ *at-pos k ns m*

using *at-pos-def Suc-ile-eq* **by** *auto*

lemma *not-at-pos-first-to-at-pos*: **assumes** ¬ *at-pos-first k ns m*
shows ¬ *at-pos k ns m* ∨ (∃ *k'<k. at-pos k' ns m*)
using *assms at-pos-first-def at-pos-def*
proof (*cases enat k < llength ns* ∧ *lnth ns k = m*)
case *True*
with *assms at-pos-first-def* **obtain** *k'* **where** *k'-gen*: *lnth ns k' = m k' < k* **by** *auto*
with *True enat-ord-simps less-trans* **have** *enat k' < llength ns* **by** *metis*
with *k'-gen at-pos-def* **show** *?thesis* **by** *auto*
next
case *False*
with *assms at-pos-first-def at-pos-def* **show** *?thesis* **by** *auto*
qed

Lemma 3.1.

theorem *on-max-paths-pos-k-first-k-unique*: **assumes** *valid-node n*
on-max-paths-pos-k-first n k1 m
on-max-paths-pos-k-first n k2 m
shows *k1 = k2*
proof (*rule ccontr*)
assume *k1 ≠ k2*
with *assms* **obtain** *k k'*
where *k-gen*: *on-max-paths-pos-k-first n k m on-max-paths-pos-k-first n k' m k < k'*
by (*cases k1 < k2*) *auto*
from *assms max-path-ext* **obtain** *ns* **where** *max-path n ns* **by** *auto*
with *k-gen on-max-paths-pos-k-first-def at-pos-first-def* **show** *False* **by** *auto*
qed

lemma *on-max-paths-pos-k-first-m-unique*: **assumes** *valid-node n*
on-max-paths-pos-k-first n k m1
on-max-paths-pos-k-first n k m2
shows *m1 = m2*
proof−
from *assms max-path-ext* **obtain** *ns* **where** *max-path n ns* **by** *auto*
with *assms on-max-paths-pos-k-first-def at-pos-first-def* **show** *?thesis* **by** *auto*
qed

Definition 3.3.

definition *tscd* :: *'node ⇒ 'node ⇒ bool*
where *tscd n m == ∃ k.* (∃ *x1∈succs n. on-max-paths-pos-k-first x1 k m*)
∧ (∃ *x2∈succs n.* ¬ *on-max-paths-pos-k-first x2 k m*)

Rule System from Theorem 3.1.

inductive *Tfirst* :: *'node ⇒ nat ⇒ 'node ⇒ bool*
where *Tfirst n 0 n*

| ∀ *x*∈*succs n*. *Tfirst x k m* ⟹ *m* ≠ *n* ⟹ *is-path n ns m* ⟹ *ns* ≠ [] ⟹ *Tfirst n* (*k*+*1*) *m*

lemma *on-max-paths-pos-k-first-refl*: *on-max-paths-pos-k-first n 0 n*
proof−
  {
    fix *ns*
    assume *max-path n ns*
    with *max-path-LCons* **obtain** *ns'* **where** *ns* = *LCons n ns'* **by** *auto*
    with *at-pos-first-def zero-enat-def* **have** *at-pos-first 0 ns n* **by** *auto*
  }
  with *on-max-paths-pos-k-first-def* **show** *?thesis* **by** *auto*
qed

lemma *on-max-path-pos-first-0*: *valid-node n* ⟹ *on-max-paths-pos-k-first n 0 m* ⟹ *n* = *m*
  using *on-max-paths-pos-k-first-m-unique on-max-paths-pos-k-first-refl* **by** *metis*

lemma *on-max-paths-pos-first-refl*: *on-max-paths-pos-first n n*
  using *on-max-paths-pos-first-def on-max-paths-pos-k-first-refl* **by** *metis*

lemma *on-max-paths-pos-k-first-0*: *valid-node n* ⟹ *on-max-paths-pos-k-first n 0 m* ⟹ *n* = *m*
  using *on-max-paths-pos-k-first-m-unique on-max-paths-pos-k-first-refl* **by** *metis*

lemma *at-pos-first-step*: **assumes** *n* ≠ *m*
                at-pos-first k ns m
        shows *at-pos-first* (*k*+*1*) (*LCons n ns*) *m*
proof−
  {
    fix *k'*
    assume *k'* < *k*+*1*
    with *assms at-pos-first-def* **have** *lnth* (*LCons n ns*) *k'* ≠ *m* **by** (*cases k'*) *auto*
  }
  with *assms at-pos-first-def Suc-ile-eq* **show** *at-pos-first* (*k*+*1*) (*LCons n ns*) *m* **by** *auto*
qed

lemma *at-pos-first-succ-Suc*: **assumes** *at-pos-first* (*k*+*1*) (*LCons n ns*) *m*
        shows *at-pos-first k ns m*
  using *assms at-pos-first-def Suc-ile-eq* **by** *auto*

lemma *at-pos-first-succ-neq*: **assumes** *n* ≠ *m*
                at-pos-first k (*LCons n ns*) *m*
        shows *k* > *0 at-pos-first* (*k*−*1*) *ns m*
proof−
  from *assms at-pos-first-def* **show** *k* > *0* **by** *force*
  with *at-pos-first-succ-Suc assms* **show** *at-pos-first* (*k*−*1*) *ns m* **by** (*cases k*) *auto*
qed

lemma *on-max-paths-pos-k-first-end-node*: **assumes** *valid-node n*
on-max-paths-pos-k-first n k m
succs n = {}
shows *k* = *0 n* = *m*
proof−
from *assms max-path.intros* **have** *max-path n* (*llist-of* [*n*]) **by** *auto*
with *assms on-max-paths-pos-k-first-def* **have** *at-pos-first k* (*llist-of* [*n*]) *m* **by** *auto*
with *at-pos-first-def enat-0-iff* **show** *k* = *0 n* = *m* **by** *auto*
qed

lemma *Tfirst-path*: *valid-node n* ⟹ *Tfirst n k m* ⟹ ∃ *ns*. *is-path n ns m*
by (*cases rule*: *Tfirst.cases*) (*auto intro*: *exI*[*of* - []])

Proof of Theorem 3.1.

theorem *on-max-paths-pos-first-Tfirst-equiv*: **assumes** *valid-node n*
shows *Tfirst n k m* ⟷ *on-max-paths-pos-k-first n k m*
proof
assume *Tfirst n k m*
then show *on-max-paths-pos-k-first n k m*
proof (*induction*)
case (*1 n*)
with *on-max-paths-pos-k-first-refl* **show** *?case* **by** *auto*
next
case (*2 n k m ns*)
with *is-path-Cons*[*of n*] **have** *has-succs*: *succs n* ≠ {} **by** (*cases ns*) *auto*
{
fix *ns*
assume *max-path n ns*
with *2 max-path-step has-succs* **obtain** *x ns'*
where *ns* = *LCons n ns' max-path x ns' x* ∈ *succs n* **by** *metis*
with *2 on-max-paths-pos-k-first-def at-pos-first-step* **have** *at-pos-first* (*k*+*1*) *ns m* **by** *auto*
}
with *on-max-paths-pos-k-first-def* **show** *?case* **by** *auto*
qed
next
assume *on-max-paths-pos-k-first n k m*
with *assms* **show** *Tfirst n k m*
proof (*induction k arbitrary*: *n*)
case (*0 n*)
with *on-max-path-pos-first-0 Tfirst.intros* **show** *?case* **by** *auto*
next
case (*Suc k n*)
with *max-path-ext* **have** *max-path n* (*ext-max-path n*) **by** *auto*
with *max-path-LCons* **obtain** *ns* **where** *max-path*: *max-path n* (*LCons n ns*) **by** *metis*

```
    with Suc on-max-paths-pos-k-first-def at-pos-first-def
    have llength (LCons n ns) > Suc k ∀ k'<k+1. lnth (LCons n ns) k' ≠ m by
auto
    with max-path enat-0-iff obtain x where x-gen: x ∈ succs n by cases auto
    from ‹∀ k'<k+1. lnth (LCons n ns) k' ≠ m› have n ≠ m by auto
    {
      fix x1
      assume succ: x1 ∈ succs n
      {
        fix ns
        assume max-path x1 ns
        with succ max-path.intros have max-path n (LCons n ns) by auto
        with at-pos-first-succ-Suc on-max-paths-pos-k-first-def Suc(3)
        have at-pos-first k ns m by fastforce
      }
     with Suc succ succs-valid on-max-paths-pos-k-first-def have Tfirst x1 k m by
auto
    }
    note succs-Tfirst = this
    with x-gen succs-valid[of x n] Tfirst-path succs-path-extend obtain ns
      where is-path n (n#ns) m by metis
    with succs-Tfirst Tfirst.intros ‹n ≠ m› show ?case by auto
  qed
qed

lemma lset-at-pos-first: assumes m ∈ lset ns
  obtains k where at-pos-first k ns m
proof−
  from assms lset-split-first obtain ns1 ns2
    where ns = lappend (llist-of ns1) (LCons m ns2) m ∉ set ns1 by metis
  then have at-pos-first (length ns1) ns m
  proof (induction ns1 arbitrary: ns)
    case Nil
    with at-pos-first-def enat-0 show ?case by auto
  next
    case (Cons n ns1)
    with at-pos-first-step show ?case by auto
  qed
  with that show ?thesis by auto
qed

lemma on-max-paths-prev-at-pos-first: assumes on-max-paths-prev n m1 m2
                                               max-path n ns
                                               at-pos-first k1 ns m1
                                               at-pos-first k2 ns m2
                                               m1 ≠ m2
                                          shows k1 < k2
proof−
  from assms on-max-paths-prev-def obtain ns1 ns2
```

```
    where ns = lappend (llist-of ns1) (LCons m1 ns2) m2 ∉ set ns1 by auto
  with assms(2−5) show ?thesis
  proof (induction ns1 arbitrary: n k1 k2 ns)
    case Nil
    with at-pos-first-def show ?case by fastforce
  next
    case (Cons n' ns1)
    then show ?case
    proof (cases n' = m1)
      case True
      with Cons at-pos-first-def show ?thesis by (cases k2 = 0) auto
    next
      case False
      let ?ns' = lappend (llist-of ns1) (LCons m1 ns2)
      have ?ns' ≠ LNil by (cases ns1) auto
      with Cons(2,6) max-path-step-LCons[of n n'] obtain x
        where x-gen: x ∈ succs n max-path x ?ns' n = n' by auto
      with Cons at-pos-first-succ-neq False
      have at-post-first-m1: at-pos-first (k1−1) ?ns' m1 by auto
      from Cons have n' ≠ m2 by auto
      with Cons at-pos-first-succ-neq x-gen have at-pos-first (k2−1) ?ns' m2 by
auto
      with at-post-first-m1 Cons x-gen have k1 − 1 < k2 − 1 by auto
      then show ?thesis by auto
    qed
  qed
qed

lemma on-max-paths-pos-k-first-step: assumes on-max-paths-pos-k-first n k m
                                             n ≠ m
                                             x ∈ succs n
                                     shows on-max-paths-pos-k-first x (k−1) m
proof−
  from on-max-path-pos-first-0 assms succs-valid have k = (k−1)+1 by (cases
k) auto
  {
    fix ns
    assume max-path x ns
    with max-path.intros on-max-paths-pos-k-first-def at-pos-first-succ-neq assms
    have at-pos-first (k−1) ns m by metis
  }
  with on-max-paths-pos-k-first-def show ?thesis by auto
qed

lemma on-max-paths-pos-first-chain: assumes on-max-paths-pos-k-first x k1 y
                                            on-max-paths-pos-k-first y k2 z
                                            max-path x ns
                                            at-pos-first k ns z
                                    shows k < k1 ∨ k = k1 + k2
```

using *assms*
proof (*induction k1 arbitrary*: *x ns k*)
case (*0 x ns k*)
with *on-max-paths-pos-k-first-0 max-path-valid-node* **have** *valid-node x x = y* **by** *auto*
with *0 on-max-paths-pos-k-first-def* **have** *at-pos-first k2 ns z* **by** *auto*
with *at-pos-first-def 0* **show** *?case* **by** (*cases rule*: *linorder-cases*) *auto*
next
case (*Suc k1 x ns k*)
with *max-path-LCons* **obtain** *ns′* **where** *ns-split*: *ns = LCons x ns′* **by** *auto*
from *on-max-paths-pos-k-first-refl* **have** *on-max-paths-pos-k-first x 0 x* **by** *auto*
with *on-max-paths-pos-k-first-k-unique Suc max-path-valid-node* **have** *x ≠ y* **by** *blast*
show *?case*
proof (*cases x = z*)
case *True*
with *ns-split Suc at-pos-first-def* **show** *?thesis* **by** *auto*
next
case *False*
with *Suc ns-split at-pos-first-def* **obtain** *k′* **where** *k = k′ + 1* **by** (*cases k*) *auto*
with *at-pos-first-succ-Suc Suc ns-split* **have** *pos-k′*: *at-pos-first k′ ns′ z* **by** *blast*
with *at-pos-first-def* **have** *ns′ ≠ LNil* **by** *auto*
from *Suc*(*4*) *ns-split Suc this* **obtain** *x2* **where** *max-path x2 ns′ x2 ∈ succs x* **by** *cases auto*
with *pos-k′ Suc on-max-paths-pos-k-first-step*[*OF Suc*(*2*)] ‹*x ≠ y*› ‹*k = k′ + 1*›
show *?thesis* **by** *auto*
qed
qed

lemma *on-max-paths-pos-first-step*: **assumes** *on-max-paths-pos-first n m*
n ≠ m
x ∈ succs n
shows *on-max-paths-pos-first x m*
using *on-max-paths-pos-first-def on-max-paths-pos-k-first-step assms* **by** *metis*

lemma *on-max-paths-pos-k-first-Suc*: **assumes** *on-max-paths-pos-k-first n* (*k+1*) *m*
x ∈ succs n
shows *on-max-paths-pos-k-first x k m*
proof−
from *on-max-paths-pos-k-first-refl assms succs-valid on-max-paths-pos-k-first-k-unique*
have *n ≠ m* **by** *fastforce*
with *assms on-max-paths-pos-k-first-step* **show** *?thesis* **by** *fastforce*
qed

lemma *on-max-paths-pos-k-implies-on-max-paths*: **assumes** *on-max-paths-pos-k-first n k m*
shows *on-max-paths n m*

proof −
  {
    fix *ns*
    assume *max-path n ns*
    with *assms on-max-paths-pos-k-first-def* **have** *at-pos-first k ns m* **by** *auto*
    with *lset-conv-lnth at-pos-first-def* **have** *m ∈ lset ns* **by** *fastforce*
  }
  with *on-max-paths-def* **show** *?thesis* **by** *auto*
qed

lemma *on-max-paths-pos-k-first-diff*: **assumes** *max-path n ns*
at-pos-first k1 ns m1
on-max-paths-pos-k-first n k2 m2
k1 ≤ k2
shows *on-max-paths-pos-k-first m1 (k2−k1) m2*
  using *assms*
proof (*induction k1 arbitrary*: *n ns k2*)
  case *0*
  with *max-path-LCons* **obtain** *ns′* **where** *ns = LCons n ns′* **by** *auto*
  with *0 at-pos-first-def* **show** *?case* **by** *auto*
next
  case (*Suc k1*)
  with *max-path-LCons* **obtain** *ns′* **where** *ns-split*: *ns = LCons n ns′* **by** *auto*
  with *Suc at-pos-first-def enat-0-iff* **have** *ns′ ≠ LNil* **by** *auto*
  with *max-path-step-LCons ns-split Suc*(*2*) **obtain** *x*
    where *x-gen*: *max-path x ns′ x ∈ succs n* **by** *blast*
  with *at-pos-first-succ-Suc Suc*(*3*) *ns-split* **have** *at-pos*: *at-pos-first k1 ns′ m1* **by** *auto*
  from *x-gen Suc on-max-paths-pos-k-first-Suc* **have** *on-max-paths-pos-k-first x (k2−1) m2* **by** *auto*
  with *at-pos Suc x-gen* **show** *?case* **by** *fastforce*
qed

lemma *tscd-cond-succ-k*: **assumes** ¬ *on-max-paths-pos-k-first n (k+1) m*
x ∈ succs n
on-max-paths-pos-k-first x k m
n ≠ m
shows *tscd n m*
proof −
  from *assms on-max-paths-pos-first-Tfirst-equiv succs-valid* **have** *Tfirst x k m* **by** *auto*
  with *assms succs-valid Tfirst-path succs-path-extend* **obtain** *ns*
    where *path*: *is-path n (n#ns) m* **by** *metis*
  {
    assume ∀ *x2∈succs n. on-max-paths-pos-k-first x2 k m*
    with *on-max-paths-pos-first-Tfirst-equiv assms succs-valid*
    have ∀ *x2∈succs n. Tfirst x2 k m* **by** *auto*
    with *path Tfirst.intros on-max-paths-pos-first-Tfirst-equiv assms* **have** *False* **by** *blast*

}
with *assms tscd-def* **show** *?thesis* **by** *auto*
qed

lemma *tscd-cond-succ*: **assumes** ¬ *on-max-paths-pos-first n m*
x ∈ *succs n*
on-max-paths-pos-first x m
shows *tscd n m*
using *assms on-max-paths-pos-first-def on-max-paths-pos-first-refl tscd-cond-succ-k*
by *metis*

4.2 Timing Sensitive Slicing

Definition of the combined slice of a binary and ternary relation. Used in Theorem 3.2 as ⊍. See Definition 3.4.

inductive-set *combined-slice*
:: ('*node* ⇒ '*node* ⇒ *bool*) ⇒ ('*node* ⇒ '*node* ⇒ '*node* ⇒ *bool*) ⇒ ('*node set*) ⇒ '*node set*
for *cd* :: '*node* ⇒ '*node* ⇒ *bool*
and *od* :: '*node* ⇒ '*node* ⇒ '*node* ⇒ *bool*
and *M* :: '*node set*
where *m* ∈ *M* ⟹ *m* ∈ *combined-slice cd od M*
| *cd n m* ⟹ *m* ∈ *combined-slice cd od M* ⟹ *n* ∈ *combined-slice cd od M*
| *od n m1 m2* ⟹ *m1* ∈ *combined-slice cd od M* ⟹ *m2* ∈ *combined-slice cd od M*
⟹ *n* ∈ *combined-slice cd od M*

Definition 3.4: The backward slice of a binary relation.

abbreviation *backward-slice* :: ('*node* ⇒ '*node* ⇒ *bool*) ⇒ ('*node set*) ⇒ '*node set*
where *backward-slice cd M* == *combined-slice cd* (λ*n m1 m2*. *False*) *M*

lemma *combined-slice-cd-rtranclp*: cd^{**} *n m* ⟹ *m* ∈ *combined-slice cd od M*
⟹ *n* ∈ *combined-slice cd od M*
by (*induction rule*: *rtranclp.induct*) (*auto intro*: *combined-slice.intros*)

This function itself is never used in this theory. It is only defined to use the resulting induction rule.

function *tscd-steps* :: '*node* ⇒ '*node list* ⇒ '*node list*
where *tscd-steps p* (*n*#*ns*) =
(*if n* = *p then* (*n*#*ns*)
else tscd-steps p (*dropWhile* (λ*m*. *on-max-paths-pos-first m n*) (*n*#*ns*)))
| *tscd-steps p* [] = []
proof−
fix *Q x*
assume (⋀*p n ns*. (*x*::'*node* × '*node list*) = (*p*, *n* # *ns*) ⟹ *Q*) (⋀*p*. *x* = (*p*, []) ⟹ *Q*)

thus *Q* **by** (*cases x*, *cases snd x*) *auto*
qed *auto*
termination
proof (*relation measure* (*length o snd*))
fix *p n ns*
from *on-max-paths-pos-first-refl length-dropWhile-le*[*of* λ*m*. *on-max-paths-pos-first m n ns*]
show ((*p*::′*node*, *dropWhile* (λ*m*. *on-max-paths-pos-first m n*) (*n* # *ns*)), (*p*, *n* # *ns*))
∈ *measure* (*length* ∘ *snd*) **by** *auto*
qed *auto*

lemma *tscd-rtranclpI*′: **assumes** *is-path p ns n*
∀ *m*∈*set* (*n*#*rev ns*). *p* ≠ *m* ⟶ ¬ *on-max-paths-pos-first p m*
shows *tscd*** *p n*
using *assms*
proof (*induction p n*#*rev ns arbitrary*: *n ns rule*: *tscd-steps.induct*)
case (*1 p n ns*)
show *?case*
proof (*cases n* = *p*)
let *?ds* = *dropWhile* (λ*m*. *on-max-paths-pos-first m n*) (*n*#*rev ns*)
let *?ts* = *takeWhile* (λ*m*. *on-max-paths-pos-first m n*) (*n*#*rev ns*)
from *on-max-paths-pos-first-refl* **have** *?ts* ≠ [] **by** *auto*
then obtain *ts-h ts*′ **where** *ts-split*: *?ts* = *ts-h*#*ts*′ **by** (*cases ?ts*) *auto*
case *False*
with *1* **have** *not-max*: ¬ *on-max-paths-pos-first p n* **by** *simp*
from *1*(*2*) *path-rev-last last-in-set*[*of n* # *rev ns*] **have** *p* ∈ *set* (*n*#*rev ns*) **by** *auto*
with *1 dropWhile-eq-Nil-conv not-max* **have** *?ds* ≠ [] **by** *auto*
then obtain *n*′ *ns-r* **where** *?ds* = *n*′#*rev* (*rev ns-r*) **by** (*cases ?ds*) *auto*
then obtain *ns*′ **where** *ds-split*: *?ds* = *n*′#*rev ns*′ **by** *blast*
with *takeWhile-dropWhile-id* **have** *split*: *n*#*rev ns* = *?ts*@*n*′#*rev ns*′ **by** *metis*
with *ts-split* **have** *rev ns* = *ts*′@*n*′#*rev ns*′ **by** *auto*
with *rev-rev-ident*[*of ns*] **have** *ns* = *ns*′@*n*′#*rev ts*′ **by** *auto*
with *1*(*2*) *is-path-split*[*of - ns*′]
have *split-path*: *is-path p ns*′ *n*′ *is-path n*′ (*n*′#*rev ts*′) *n* **by** *auto*
from *split* **have** *set* (*n*′#*rev ns*′) ⊆ *set* (*n*#*rev ns*) **by** *auto*
with *1* **have** ∀ *m*∈*set* (*n*′ # *rev ns*′). *p* ≠ *m* ⟶ ¬ *on-max-paths-pos-first p m* **by** *auto*
with *1 False ds-split split-path* **have** *tscd*** *p n*′ **by** *auto*
from *ds-split*[*unfolded dropWhile-eq-Cons-conv*] **have** ¬ *on-max-paths-pos-first n*′ *n* **by** *auto*
obtain *x2* **where** *on-max-paths-pos-first x2 n x2* ∈ *succs n*′
proof (*cases rev ts*′)
case *Nil*
with *split-path path-last-is-edge*[*of - -* [*n*′]] *edge-rel-def* **have** *n* ∈ *succs n*′ **by** *auto*
with *that on-max-paths-pos-first-refl* **show** *?thesis* **by** *auto*

```
  next
    case (Cons t′ ts′′)
    with split-path have is-path n′ (n′#t′#ts′′) n by auto
    with is-path-split[of - [n′]] have is-path n′ [n′] t′ by auto
    with path-last-is-edge[of - - [n′]] edge-rel-def have t′ ∈ succs n′ by auto
    from ts-split Cons have t′ ∈ set ?ts by auto
    hence on-max-paths-pos-first t′ n by (auto dest: set-takeWhileD)
    with ‹t′ ∈ succs n′› that show ?thesis by auto
  qed
  with ‹¬ on-max-paths-pos-first n′ n› tscd-cond-succ have tscd n′ n by auto
  with ‹tscd** p n′› show ?thesis by auto
 qed auto
qed

lemma tscd-rtranclpI: assumes is-path p ns n
 ∀ m∈set ns ∪ {n}. p ≠ m ⟶ ¬ on-max-paths-pos-first p m
              shows tscd** p n
using assms tscd-rtranclpI′ by auto

lemma on-max-paths-pos-k-first-less-eq: assumes on-max-paths-pos-k-first n k1 m1
                                  on-max-paths-pos-k-first n k2 m2
                                  k1 ≤ k2
                                  max-path n (lappend ns1 ns2)
                                  m2 ∈ lset ns1
                            shows m1 ∈ lset ns1
proof−
 from assms in-lset-conv-lnth obtain k where k-gen: m2 = lnth ns1 k k < llength ns1 by metis
 with lnth-lappend1 have m2 = lnth (lappend ns1 ns2) k by metis
 with assms on-max-paths-pos-k-first-def at-pos-first-def not-less have k ≥ k2 by metis
 with assms k-gen enat-ord-simps less-le-trans not-less have k1 < llength ns1 by metis
 with assms on-max-paths-pos-k-first-def at-pos-first-def lnth-lappend1 in-lset-conv-lnth
 show ?thesis by metis
qed

lemma on-max-paths-prev-ccontr: assumes on-max-paths-prev x n m
                           n ≠ m
                           is-path x ms m
                           n ∉ set ms
                     shows False
proof−
 from assms is-path-valid-node max-path-ext have max-path m (ext-max-path m) by auto
 with max-path-LCons obtain ems′ where ext-eq: ext-max-path m = LCons m ems′ by auto
 let ?ms′ = lappend (llist-of ms) (LCons m ems′)
```

from ‹*max-path m* (*ext-max-path m*)› *ext-eq assms max-path-append* **have** *max-path x ?ms'* **by** *auto*
with *assms on-max-paths-prev-def* **obtain** *ms1 ms2* **where** $m \notin$ *set ms1*
?ms' = *lappend* (*llist-of ms1*) (*LCons n ms2*) **by** *auto*
with *assms lappend-split-eq*[*OF this*(*2*)] **show** *?thesis* **by** *auto*
qed

lemma *on-max-paths-prev-split*:
assumes *on-max-paths-prev n m1 m2*
valid-node n
obtains *ns1 ns2* **where** *is-path n ns1 m1 max-path m1* (*LCons m1 ns2*)
m1 $\notin$ *set ns1 m2* $\notin$ *set ns1*
proof−
from *max-path-ext assms* **have** *max-path n* (*ext-max-path n*) **by** *simp*
with *assms on-max-paths-prev-def* **obtain** *ns1' ns2*
where *ns-gen*: *max-path n* (*lappend* (*llist-of ns1'*) (*LCons m1 ns2*)) *m2* $\notin$ *set ns1'* **by** *auto*
with *max-path-split* **have** *split1*: *is-path n ns1' m1 max-path m1* (*LCons m1 ns2*) **by** *auto*
with *path-first* **obtain** *ns1 nsx* **where** *is-path n ns1 m1 m1* $\notin$ *set ns1 ns1'* = *ns1*@*nsx* **by** *metis*
with *ns-gen split1 that* **show** *thesis* **by** *auto*
qed

Proof of Theorem 3.2.

theorem *tscd-slice-includes-ntscd-dod*:
combined-slice ntscd dod M $\subseteq$ *backward-slice tscd M*
proof
fix *x*
assume *x* $\in$ *combined-slice ntscd dod M*
then show *x* $\in$ *backward-slice tscd M*
proof *induction*
case (*2 n m*)
with *ntscd-def* **obtain** *x1 x2* **where** *succs*: *x1* $\in$ *succs n on-max-paths x1 m*
x2 $\in$ *succs n* $\neg$ *on-max-paths x2 m* **by** *auto*
with *on-max-paths-ex-path succs-valid* **obtain** *ns'* **where** *is-path x1 ns' m* **by** *blast*
with *path-first* **obtain** *ns* **where** *path1*: *is-path x1 ns m m* $\notin$ *set ns* **by** *metis*
with *succs succs-path-extend* **have** *path2*: *is-path n* (*n#ns*) *m* **by** *blast*
{
fix *m'*
assume *m'-gen*: *m'*$\in$*set ns* $\cup$ {*m*} *n* $\neq$ *m' on-max-paths-pos-first n m'*
with *path-split-elem2 path1* **obtain** *ns1 ns2*
where *ns-split*: *is-path x1 ns1 m' ns* = *ns1*@*ns2* **by** *metis*
with *path1* **have** *m* $\notin$ *set ns1* **by** *auto*
from *succs on-max-paths-def* **obtain** *ms* **where** *ms-gen*: *max-path x2 ms m* $\notin$ *lset ms* **by** *auto*
from *m'-gen on-max-paths-pos-first-step succs* **have** *on-max-paths-pos-first x2 m'* **by** *auto*

with *on-max-paths-pos-first-def on-max-paths-pos-k-first-def ms-gen* **obtain** *k*
where *at-pos-first k ms m′* **by** *auto*
with *at-pos-first-def lset-conv-lnth* **have** *m′* ∈ *lset ms* **by** *fastforce*
with *max-path-split-elem ms-gen* **obtain** *ms1 ms2*
where *ms-split*: *max-path m′ ms2 ms* = *lappend* (*llist-of ms1*) *ms2* **by** *metis*
with *ns-split max-path-append* **have** *max-path x1* (*lappend* (*llist-of ns1*) *ms2*)
by *auto*
with *on-max-paths-def succs ms-gen ms-split ns-split path1 lset-lappend-lfinite*
have *False* **by** *auto*
}
with *path2 tscd-rtranclpI* **have** *tscd*** *n m* **by** *fastforce*
with *combined-slice-cd-rtranclp 2* **show** *?case* **by** *auto*
next
case (*3 n m1 m2*)
with *dod-def* **obtain** *x1 x2* **where** *succs*: *x1* ∈ *succs n on-max-paths-prev x1*
m1 m2
x2 ∈ *succs n on-max-paths-prev x2 m2 m1 m1* ≠ *m2* **by** *auto*
with *succs-valid on-max-paths-prev-split* **obtain** *ns11 ns12*
where *path1*: *is-path x1 ns11 m1 max-path m1* (*LCons m1 ns12*)
m1 ∉ *set ns11 m2* ∉ *set ns11* **by** *metis*
have *tscd*** *n m1* ∨ *tscd*** *n m2*
proof (*cases* ∀ *m1′*∈*set ns11* ∪ {*m1*}. *n* ≠ *m1′* ⟶ ¬ *on-max-paths-pos-first*
n m1′)
case *True*
from *succs succs-path-extend path1* **have** *is-path n* (*n#ns11*) *m1* **by** *auto*
with *True tscd-rtranclpI* **show** *?thesis* **by** *auto*
next
case *False*
then obtain *m1′*
where *m1′-gen*: *m1′*∈*set ns11* ∪ {*m1*} *n* ≠ *m1′ on-max-paths-pos-first n*
m1′ **by** *auto*
from *succs succs-valid on-max-paths-prev-split* **obtain** *ns21 ns22*
where *path2*: *is-path x2 ns21 m2 max-path m2* (*LCons m2 ns22*)
m1 ∉ *set ns21 m2* ∉ *set ns21* **by** *metis*
with *succs succs-path-extend* **have** *path3*: *is-path n* (*n#ns21*) *m2* **by** *auto*
{
fix *m2′*
assume *m2′-gen*: *m2′*∈*set ns21* ∪ {*m2*} *n* ≠ *m2′ on-max-paths-pos-first n*
m2′
with *m1′-gen on-max-paths-pos-first-def* **obtain** *k1 k2*
where *k-gen*: *on-max-paths-pos-k-first n k1 m1′*
on-max-paths-pos-k-first n k2 m2′ **by** *auto*
obtain *m′* **where** *m′* ∈ *set ns11* ∪ {*m1*} *m′* ∈ *set ns21* ∪ {*m2*}
proof (*cases k1* ≤ *k2*)
case *True*
from *max-path-append*[*OF path2*(*1*,*2*)] *succs max-path.intros*
have *max-path n* (*lappend* (*llist-of* (*n#ns21*@[*m2*])) *ns22*)
by (*auto simp*: *lappend-llist-of-LCons*)
with *on-max-paths-pos-k-first-less-eq*[*OF k-gen - this*] *True m2′-gen m1′-gen*

that
        show *?thesis* **by** *fastforce*
      next
        case *False*
        from *max-path-append*[*OF path1*(*1*,*2*)] *succs max-path.intros*
        have *max-path n* (*lappend* (*llist-of* (*n#ns11*@[*m1*])) *ns12*)
          by (*auto simp*: *lappend-llist-of-LCons*)
        with *on-max-paths-pos-k-first-less-eq*[*OF k-gen*(*2*,*1*) - *this*] *False m2'-gen m1'-gen that*
        show *?thesis* **by** *fastforce*
      qed
      with *path-split-elem2 path1 path2 m1'-gen m2'-gen* **obtain** *ns1a ns1b ns2a ns2b*
        where *split*: *ns11* = *ns1a*@*ns1b is-path x1 ns1a m'*
                  ns21 = *ns2a*@*ns2b is-path m' ns2b m2* **by** *metis*
      with *path-append path2* **have** *is-path x1* (*ns1a*@*ns2b*) *m2* **by** *metis*
    with *split on-max-paths-prev-ccontr succs path1 path2* **have** *False* **by** *fastforce*
    }
    with *path3 tscd-rtranclpI* **show** *?thesis* **by** *fastforce*
  qed
  with *combined-slice-cd-rtranclp 3* **show** *?case* **by** *auto*
 qed (*auto intro*: *combined-slice.intros*)
qed

4.3 Soundness and Minimality of Timing Sensitive Control Dependence

4.3.1 Definition of (clocked) Traces and Time-Sensitive Non-Interference

Definition of the set of input nodes (nodes with more than one successor).

definition *input-nodes* :: *'node set*
 where *input-nodes* = {*n* . ∃ *x y*. *x* ∈ *succs n* ∧ *y* ∈ *succs n* ∧ *x* ≠ *y*}

A trace (unclocked) is a (potentially infinite) list of partial edges.

type-synonym *'a trace* = (*'a* × *'a option*) *llist*

An input is a map from nodes to a (potentially infinite) list of nodes. The k-th element of the list for a node n gives the successor chosen at the k-th visit of n.

To guarantee that valid maximal traces are produced when using an input i, we require that for each n, each element of the list i n has to be a successor of n. Also, if n is not an exit node, the list i n has to be infinite.

definition *is-input* :: (*'node* ⇒ *'node llist*) ⇒ *bool*
 where *is-input i* == ∀ *n*. (∀ *m*∈*lset* (*i n*). *m* ∈ *succs n*) ∧ (*succs n* ≠ {} ⟶ ¬ *lfinite* (*i n*))

Definition of the next node of the trace, which is read from input. If we choose a node m as a successor, this function returns *Some m*. If the current node is an exit node, we return *None*, resulting in a partial edge.

fun *read* :: ('*node* ⇒ '*node llist*) ⇒ '*node* ⇒ '*node option*
  where *read i n* = (*if succs n* = {} *then None else Some* (*lhd* (*i n*)))

Constructs the trace with given start node according to the given input. Ends in a partial edge if we reach an exit node, otherwise produces an infinite trace.

primcorec *exec* :: '*node* ⇒ ('*node* ⇒ '*node llist*) ⇒ '*node trace*
  where *exec n i* = *LCons* (*n*, *read i n*)
    (*if succs n* = {} *then LNil else exec* (*lhd* (*i n*)) (*i*(*n*:=*ltl* (*i n*))))

Definition of Observational equivalence of inputs given an observable node set. Inputs are equivalent with regards to a given set if the input lists are equal for each node of the observable set (i.e. if the chosen successors are the same at observable nodes).

definition *input-obs-equiv* :: '*node set* ⇒ ('*node* ⇒ '*node llist*) ⇒ ('*node* ⇒ '*node llist*) ⇒ *bool*
  where *input-obs-equiv S i1 i2* == ∀ *n* ∈ *S*. *i1 n* = *i2 n*

A clocked trace is a (potentially infinite) list of partial edges annotated with the time at which it is executed.

type-synonym '*a t-trace* = (*nat* × '*a* × '*a option*) *llist*

Definition of the timed observable sub-trace, given an observable node set and a starting time. We take a given trace, annotate it with timing information (starting at the given time), and then filter out every non-observable node. Helper definition to describe suffixes of timed observable sub-traces.

fun *trace-time-obs'* :: '*node set* ⇒ *nat* ⇒ '*node trace* ⇒ '*node t-trace*
  where *trace-time-obs' S k ns* = *lfilter* (λ*p*. *fst* (*snd p*) ∈ *S*) (*lzip* (*iterates Suc k*) *ns*)

Definition 3.7: Definition of the timed observable sub-trace, given an observable node set, starting at time 0.

fun *trace-time-obs* :: '*node set* ⇒ '*node trace* ⇒ '*node t-trace*
  where *trace-time-obs S ns* = *trace-time-obs' S 0 ns*

Definition 3.7: Definition of Observational equivalence of timed traces given an observable node set.

definition *trace-time-obs-equiv* :: '*node set* ⇒ '*node trace* ⇒ '*node trace* ⇒ *bool*
  where *trace-time-obs-equiv S ns1 ns2* == *trace-time-obs S ns1* = *trace-time-obs S ns2*

Definition 3.8: Time-sensitive Noninterference. If it holds, an attacker gains no information about choices made at non-observable nodes by observing the resulting trace at observable nodes. This is true even if they have a clock.

definition *noninterferent-time* :: *'node set* ⇒ *bool*
  where *noninterferent-time S* == ∀ *i1 i2 n*. *input-obs-equiv S i1 i2*
    ⟶ *valid-node n* ⟶ *is-input i1* ⟶ *is-input i2*
    ⟶ *trace-time-obs-equiv S* (*exec n i1*) (*exec n i2*)

4.3.2 Soundness of Timing Sensitive Control Dependence

Alternate definition of equality for potentially infinite lists, which is sometimes easier to work with in proofs.

coinductive *llist-eq* :: *'a llist* ⇒ *'a llist* ⇒ *bool*
  where *llist-eq LNil LNil*
  | *llist-eq xs ys* ⟹ *llist-eq* (*LCons x xs*) (*LCons x ys*)

Proof that the alternate definition of equality for potentially infinite lists is correct.

lemma *llist-eq-is-eq*: *llist-eq xs ys* ⟷ *xs* = *ys*
proof
  assume *llist-eq xs ys*
  then show *xs* = *ys* **by** (*coinduction arbitrary*: *xs ys*) (*auto elim*: *llist-eq.cases*)
next
  assume *xs* = *ys*
  then show *llist-eq xs ys*
  proof (*coinduction arbitrary*: *xs ys*)
    case (*llist-eq xs ys*)
    then show *?case* **by** (*cases xs*; *cases ys*) *auto*
  qed
qed

Next observable node (annotated with a time). Might not be unique if the program is not non-interferent. Includes the "non-observation" (no more observable events) as an explicit observation. Helper definition for the proof of Theorem 3.3.

inductive *next-obs-t* :: *'node set* ⇒ *'node* ⇒ (*'node* × *nat*) *option* ⇒ *bool*
  where *is-path n ns m* ⟹ *length ns* = *k* ⟹ ∀ *n'*∈*set ns*. *n'* ∉ *S* ⟹ *m* ∈ *S*
    ⟹ *next-obs-t S n* (*Some* (*m*, *k*))
  | *max-path n ns* ⟹ ∀ *n'*∈*lset ns*. *n'* ∉ *S* ⟹ *next-obs-t S n None*

lemma *next-obs-t-in-S*: **assumes** *valid-node n*
    n ∈ *S*
  shows *next-obs-t S n* (*Some* (*n*, *0*))
  using *assms next-obs-t.intros*(*1*)[*of n* []] **by** *auto*

lemma *next-obs-t-prev-Some*: **assumes** *next-obs-t S x* (*Some* (*m*, *k*))
    x ∈ *succs n*
    n ∉ *S*

shows *next-obs-t S n* (*Some* (*m*, *k*+*1*))
using *assms succs-path-extend* **by** *cases* (*auto intro*!: *next-obs-t.intros*)

Helper definition for the proof of Theorem 3.3. *tcc S* holds if all nodes have only one possible next observation.

definition *tcc* :: *'node set* ⇒ *bool*
where *tcc S* == ∀ *n o1 o2*. *valid-node n* ∧ *next-obs-t S n o1* ∧ *next-obs-t S n o2* ⟶ *o1* = *o2*

lemma *is-input-step*: **assumes** *is-input i*
shows *is-input* (*i*(*n* := *ltl* (*i n*))) *succs n* ≠ {} ⟶ *lhd* (*i n*) ∈ *succs n*
proof−
from *assms is-input-def lset-ltl*[*of i n*] **show** *is-input*: *is-input* (*i*(*n* := *ltl* (*i n*))) **by** *auto*
from *assms is-input-def* **show** *succs n* ≠ {} ⟶ *lhd* (*i n*) ∈ *succs n* **by** (*cases i n*) *auto*
qed

lemma *is-input-max-path*: **assumes** *valid-node n*
is-input i
shows *max-path n* (*lmap fst* (*exec n i*))
using *assms*
proof (*coinduction arbitrary*: *n i*)
case (*max-path n i*)
show *?case*
proof (*cases succs n* = {})
case *True*
with *max-path exec.code* **show** *?thesis* **by** *auto*
next
let *?n'* = *lhd* (*i n*)
let *?i'* = *i*(*n* := *ltl* (*i n*))
case *False*
with *exec.code*[*of n i*]
have *lmap fst* (*exec n i*) = *LCons n* (*lmap fst* (*exec ?n' ?i'*)) **by** *auto*
with *max-path is-input-step False exec.code*[*of n i*] *succs-valid* **show** *?thesis* **by** *blast*
qed
qed

lemma *tscd-slice-sound*: **shows** *tcc* (*backward-slice tscd M*) (**is** *tcc ?S*)
proof−
{
fix *n m k*
assume *next-obs-t ?S n* (*Some* (*m*, *k*))
then obtain *ns*
where *is-path n ns m* ∀ *n'*∈*set ns*. *n'* ∉ *?S length ns* = *k m* ∈ *?S* **by** *cases auto*
then have *on-max-paths-pos-k-first n k m*
proof (*induction ns arbitrary*: *n k*)

case *Nil*
with *on-max-paths-pos-k-first-refl* **show** *?case* **by** *auto*
next
case (*Cons n′ ns n k*)
with *is-path-Cons* **obtain** *n″*
where *split*: *n* = *n′ n″* ∈ *succs n is-path n″ ns m* **by** *metis*
{
assume ¬ *on-max-paths-pos-k-first n k m*
with *Cons tscd-cond-succ-k split* **have** *tscd n m* **by** *fastforce*
with *Cons split* **have** *False* **by** (*auto intro*: *combined-slice.intros*)
}
then show *?case* **by** *auto*
qed
}
note *next-obs-Some* = *this*
{
fix *n m k*
assume *assm1*: *next-obs-t ?S n* (*Some* (*m*, *k*))
then have *m* ∈ *?S* **by** *cases auto*
from *assm1 next-obs-Some* **have** *pos-k*: *on-max-paths-pos-k-first n k m* **by** *auto*
assume *next-obs-t ?S n None*
then obtain *ns* **where** *max-path n ns* ∀ *n′*∈*lset ns*. *n′* ∉ *?S* **by** *cases auto*
with *pos-k on-max-paths-pos-k-first-def at-pos-first-def lset-conv-lnth* ‹*m* ∈ *?S*›
have *False* **by** *fastforce*
}
note *not-Some-None* = *this*
{
fix *n m1 k1 m2 k2*
assume *obs*: *next-obs-t ?S n* (*Some* (*m1*, *k1*)) *next-obs-t ?S n* (*Some* (*m2*, *k2*)) *k1* < *k2*
with *next-obs-t.cases next-obs-Some*
have *m1-obs-pos*: *m1* ∈ *?S on-max-paths-pos-k-first n k1 m1* **by** *blast+*
from *obs*(*2*) **obtain** *ns*
where *ns-gen*: *m2* ∈ *?S* ∀ *n′*∈*set ns*. *n′* ∉ *?S length ns* = *k2 is-path n ns m2*
by *cases auto*
with *is-path-valid-node max-path-ext* **obtain** *ns′* **where** *max-path m2 ns′* **by** *blast*
with *ns-gen max-path-append* **have** *max-path n* (*lappend* (*llist-of ns*) *ns′*) **by** *auto*
with *m1-obs-pos on-max-paths-pos-k-first-def at-pos-first-def*
have *lnth* (*lappend* (*llist-of ns*) *ns′*) *k1* = *m1* **by** *auto*
with *m1-obs-pos ns-gen obs* **have** *False* **by** (*auto simp add*: *lnth-lappend-llist-of*)
}
note *not-Some-Some-unequal-k* = *this*
{
fix *n obs1 obs2*
assume *obs*: *next-obs-t ?S n obs1 next-obs-t ?S n obs2 valid-node n*
have *obs1* = *obs2*
proof (*cases obs1*)

case *None*
with *obs not-Some-None* **show** *?thesis* **by** (*cases obs2*) *auto*
next
case (*Some o1*)
then obtain *m1 k1* **where** *obs1*: *obs1* = *Some* (*m1*, *k1*) **by** *fastforce*
with *obs not-Some-None* **show** *?thesis*
proof (*cases obs2*)
case (*Some o2*)
then obtain *m2 k2* **where** *obs2*: *obs2* = *Some* (*m2*, *k2*) **by** *fastforce*
with *obs1 obs not-Some-Some-unequal-k* **have** *k1* = *k2* **by** (*cases rule*: *linorder-cases*) *auto*
with *obs1 obs2 obs on-max-paths-pos-k-first-m-unique*
show *?thesis* **by** (*auto dest*!: *next-obs-Some*)
qed *auto*
qed
}
with *tcc-def* **show** *?thesis* **by** *auto*
qed

lemma *trace-time-obs-LNil*: **assumes** *trace-time-obs′ S k* (*exec n i*) = *LNil*
is-input i
valid-node n
shows *next-obs-t S n None*
proof−
{
fix *m*
assume *m* ∈ *lset* (*lmap fst* (*exec n i*))
then obtain *obs1* **where** *obs1-gen*: *obs1* ∈ *lset* (*exec n i*) *fst obs1* = *m* **by** *auto*
with *in-lset-conv-lnth* **obtain** *k1*
where *lnth* (*exec n i*) *k1* = *obs1 k1* < *llength* (*exec n i*) **by** *metis*
with *lset-lzip llength-iterates*
have (*k*+*k1*, *obs1*) ∈ *lset* (*lzip* (*iterates Suc k*) (*exec n i*)) **by** *force*
with *assms obs1-gen* **have** *m* ∉ *S* **by** (*auto simp add*: *lfilter-eq-LNil*)
}
with *is-input-max-path assms next-obs-t.intros* **show** *?thesis* **by** *metis*
qed

lemma *trace-time-obs-LCons*: **assumes** *trace-time-obs′ S k* (*exec n i*) = *LCons* (*k′*, *m*, *m′*) *ns′*
is-input i
valid-node n
shows *m* ∈ *S*
next-obs-t S n (*Some* (*m*, *k′*−*k*))
k′ ≥ *k*
m′ = *read i m*
succs m = {} ⟶ *ns′* = *LNil*
succs m ≠ {} ⟶
(∃ *i′*. *ns′* = *trace-time-obs′ S* (*k′*+*1*) (*exec* (*lhd* (*i*

m)) *i*′)

$\wedge$ *is-input i*′
$\wedge$ *input-obs-equiv S i*′ (*i*(*m*:=*ltl* (*i m*)))
$\wedge$ *lhd* (*i m*) $\in$ *succs m*)
(**is** - $\longrightarrow$ *?cont*)

proof−
from *assms lfilter-eq-LConsD*[*of* λ*obs*. *fst* (*snd obs*) ∈ *S lzip* (*iterates Suc k*) (*exec n i*)]
obtain *ns1*′ *ns2*
where *split*: *lzip* (*iterates Suc k*) (*exec n i*) = *lappend ns1*′ (*LCons* (*k*′, *m*, *m*′) *ns2*)
lfinite ns1′ ∀ *m*′∈*lset ns1*′. *fst* (*snd m*′) ∉ *S m* ∈ *S*
ns′ = *lfilter* (λ*obs*. *fst* (*snd obs*) ∈ *S*) *ns2*
by *fastforce*
with *lfinite-eq-range-llist-of* **obtain** *ns1* **where** *ns1-gen*: *ns1*′ = *llist-of ns1* **by** *auto*
with *split*
have *lzip* (*iterates Suc k*) (*exec n i*) = *lappend* (*llist-of ns1*) (*LCons* (*k*′, *m*, *m*′) *ns2*)
∀ *m*′∈*set ns1*. *fst* (*snd m*′) ∉ *S* **by** *auto*
with *assms*(*2*,*3*) *split*(*4*,*5*) **have** *next-obs-t S n* (*Some* (*m*, *k*′−*k*)) ∧ *k*′ ≥ *k* ∧ *m*′ = *read i m*
∧ (*succs m* = {} ⟶ *ns*′ = *LNil*) ∧ (*succs m* ≠ {} ⟶ *?cont*)
proof (*induction ns1 arbitrary*: *n i k*)
case (*Nil n i k*)
let *?i*′ = *i*(*m*:=*ltl* (*i m*))
let *?n*′ = *lhd* (*i n*)
from *Nil* **obtain** *ks ns*′′
where *dezip*: *iterates Suc k* = *LCons k*′ *ks exec n i* = *LCons* (*m*, *m*′) *ns*′′
ns2 = *lzip ks ns*′′ **by** (*auto simp add*: *lzip-eq-LCons-conv*)
with *exec.code*[*of n i*] **have** *exec*: *n* = *m m*′ = *read i m*
ns′′ = (*if succs n* = {} *then LNil else exec ?n*′ *?i*′) **by** *auto*
with *next-obs-t-in-S Nil* **have** *obs*: *next-obs-t S n* (*Some* (*m*, *0*)) **by** *auto*
from *dezip iterates.code*[*of Suc k*] **have** *iterate*: *k* = *k*′ *ks* = *iterates Suc* (*k*+*1*) **by** *auto*
with *exec dezip obs Nil* **show** *?case*
proof (*cases succs m* = {})
case *False*
with *iterate Nil exec dezip*
have *ns*′: *ns*′ = *trace-time-obs*′ *S* (*k*′+*1*) (*exec* (*lhd* (*i m*)) *?i*′) **by** *auto*
from *Nil False is-input-step* **have** *input-step*: *is-input ?i*′ *lhd* (*i m*) ∈ *succs m* **by** *auto*
from *input-obs-equiv-def*
have *input-obs-equiv S ?i*′ (*i*(*m*:=*ltl* (*i m*))) **by** *auto*
with *ns*′ *input-step exec obs iterate* **show** *?thesis* **by** *fastforce*
qed *auto*
next
case (*Cons obs ns1 n i k*)
let *?kx* = *fst obs*

```
    let ?x = fst (snd obs)
    let ?x' = snd (snd obs)
    let ?i' = i(n:=ltl (i n))
    let ?n' = lhd (i n)
    from Cons obtain ks ns''
      where dezip: iterates Suc k = LCons ?kx ks exec n i = LCons (?x, ?x') ns''
              lzip ks ns'' = lappend (llist-of ns1) (LCons (k', m, m') ns2)
      by (auto simp add: lzip-eq-LCons-conv)
    then have ns'' ≠ LNil by (cases ns1) auto
    with exec.code[of n i] dezip have succs n ≠ {} by auto
    with exec.code[of n i] have exec n i = LCons (n, read i n) (exec ?n' ?i') by
auto
    from this dezip(2)
    have exec: ?x = n ?x' = read i n ns'' = exec ?n' ?i' by auto
    from Cons is-input-step ‹succs n ≠ {}› succs-valid
    have valid: is-input ?i' ?n' ∈ succs n valid-node ?n' by metis+
    from Cons have ns': ns' = lfilter (λobs. fst (snd obs) ∈ S) ns2
                  ∀ m'∈set ns1. fst (snd m') ∉ S by auto
    from dezip exec iterates.code[of Suc k]
    have lzip (iterates Suc (k+1)) (exec ?n' ?i') = lappend (llist-of ns1) (LCons
(k',m,m') ns2)
      by auto
    with Cons valid ns'
    have step: next-obs-t S ?n' (Some (m, k' − (k+1))) ∧ (k+1) ≤ k'
            ∧ m' = read ?i' m
            ∧ (succs m = {} ⟶ ns' = LNil)
            ∧ (succs m ≠ {}
                ⟶ (∃ i'. ns' = trace-time-obs' S (k' + 1) (exec (lhd (?i' m)) i')
                          ∧ is-input i'
                          ∧ input-obs-equiv S i' (?i'(m := ltl (?i' m)))
                          ∧ lhd (?i' m) ∈ succs m))
      by blast
     with step add-diff-assoc2 diff-cancel2 have k-diff: k'−(k+1)+1 = k'−k by
metis
    from Cons exec have n ∉ S by auto
    with next-obs-t-prev-Some[where ?k=k' − (k+1)] k-diff step valid
    have obs-step: next-obs-t S n (Some (m, k' − k)) ∧ k ≤ k' by auto
    from step ‹n ∉ S› ‹m ∈ S› have read: ?i' m = i m m' = read i m by auto
    with step obs-step show ?case
    proof (cases succs m = {})
      case False
       with step obtain i' where i'-gen: ns' = trace-time-obs' S (k' + 1) (exec
(lhd (?i' m)) i')
        is-input i' input-obs-equiv S i' (?i'(m := ltl (?i' m)))
        lhd (?i' m) ∈ succs m by auto
      with read input-obs-equiv-def ‹n ∉ S›
      have ns' = trace-time-obs' S (k' + 1) (exec (lhd (i m)) i')
        input-obs-equiv S i' (i(m := ltl (i m)))
        lhd (i m) ∈ succs m by auto
```

```
      with False obs-step read i′-gen show ?thesis by blast
    qed auto
  qed
  with ‹m ∈ S› show m ∈ S next-obs-t S n (Some (m, k′−k)) k′ ≥ k m′ = read
i m
                succs m = {} ⟶ ns′ = LNil succs m ≠ {} ⟶ ?cont by auto
qed

lemma trace-time-obs-equiv-subset: assumes S1 ⊆ S2
                                        trace-time-obs-equiv S2 ns1 ns2
                                shows trace-time-obs-equiv S1 ns1 ns2
proof−
  {
    fix ns :: ′node t-trace
    from assms have (λp. fst (snd p) ∈ S1) = (λp. fst (snd p) ∈ S1 ∧ fst (snd
p) ∈ S2) by auto
    then have lfilter (λp. fst (snd p) ∈ S1) ns
             = lfilter (λp. fst (snd p) ∈ S1 ∧ fst (snd p) ∈ S2) ns by metis
    with lfilter-lfilter[symmetric] have lfilter (λp. fst (snd p) ∈ S1) ns
        = lfilter (λp. fst (snd p) ∈ S1) (lfilter (λp. fst (snd p) ∈ S2) ns) by metis
  }
  from assms this[of lzip - ns1] this[of lzip - ns2] trace-time-obs-equiv-def
  show ?thesis by auto
qed

lemma singleton-repeat: assumes ∀ m ∈ lset ns. m ∈ {x}
                              ¬ lfinite ns
                        shows ns = repeat x
  using assms
proof (coinduction arbitrary: ns)
  case Eq-llist
  then obtain n ns′ where ns = LCons n ns′ by (cases ns) auto
  with Eq-llist show ?case by auto
qed

lemma is-input-linear-repeat: assumes is-input i
                                    succs n ≠ {}
                                    n ∉ input-nodes
                              shows i n = repeat (THE x. x ∈ succs n)
proof−
  from assms input-nodes-def obtain x where succs n = {x} by auto
  with assms is-input-def singleton-repeat show ?thesis by fastforce
qed

lemma input-obs-equiv-input-nodes: assumes input-obs-equiv (S ∩ input-nodes)
i1 i2
                                        is-input i1
                                        is-input i2
                                  shows input-obs-equiv S i1 i2
```

```
proof−
  {
    fix n
    assume n-gen: n ∈ S n ∉ input-nodes
    have i1 n = i2 n
    proof (cases succs n = {})
      case True
      with assms is-input-def have ∀ m∈ lset (i1 n). False ∀ m∈ lset (i2 n). False
by blast+
      then show ?thesis by (cases i1 n; cases i2 n) auto
    next
      case False
      with assms n-gen is-input-linear-repeat show ?thesis by metis
    qed
  }
  with input-obs-equiv-def assms show ?thesis by fastforce
qed

lemma tcc-noninterferent-time: assumes tcc S
                               shows noninterferent-time S
proof−
  {
    obtain k :: nat where k = 0 by simp
    fix n i1 i2
    assume valid: valid-node n is-input i1 is-input i2
    assume input-obs-equiv S i1 i2
    with valid
    have llist-eq (trace-time-obs' S k (exec n i1)) (trace-time-obs' S k (exec n i2))
    proof (coinduction arbitrary: k n i1 i2)
      case (llist-eq k n i1 i2)
      show ?case
      proof (cases trace-time-obs' S k (exec n i1))
        case LNil
        then show ?thesis
        proof (cases trace-time-obs' S k (exec n i2))
          case (LCons x21 x22)
          with trace-time-obs-LCons[where ?m=fst (snd x21)] llist-eq
           have Some-obs: next-obs-t S n (Some ((fst (snd x21)), fst x21 − k)) by
(cases x21) auto
          from LNil llist-eq trace-time-obs-LNil have next-obs-t S n None by auto
          with Some-obs assms tcc-def llist-eq show ?thesis by auto
        qed auto
      next
        case split1: (LCons p1 ns1)
        obtain k1' n1 n1' where p1-split: p1 = (k1', n1, n1') by (cases p1)
        with trace-time-obs-LCons[where ?m=n1] llist-eq split1
        have obs1: next-obs-t S n (Some (n1, k1'−k)) ∧ n1 ∈ S k1' ≥ k by auto
        show ?thesis
        proof (cases trace-time-obs' S k (exec n i2))
```

case *LNil*
with *llist-eq trace-time-obs-LNil* **have** *next-obs-t S n None* **by** *auto*
with *obs1 assms tcc-def llist-eq* **show** *?thesis* **by** *auto*
next
case *split2*: (*LCons p2 ns2*)
obtain *k2′ n2 n2′* **where** *p2-split*: *p2* = (*k2′*, *n2*, *n2′*) **by** (*cases p2*)
with *trace-time-obs-LCons*[**where** *?m=n2*] *llist-eq split2*
have *next-obs-t S n* (*Some* (*n2*, *k2′−k*)) *k2′* ≥ *k* **by** *auto*
with *obs1 tcc-def llist-eq assms eq-diff-iff*[*of k k1′ k2′*]
have *n-eq*: *n1* = *n2 k1′* = *k2′* **by** *auto*
note *splits* = *split1 split2 p1-split p2-split*
from *llist-eq splits trace-time-obs-LCons*
have *n′-reads*: *n1′* = *read i1 n1 n2′* = *read i2 n2* **by** *metis+*
show *?thesis*
proof (*cases succs n1* = {})
case *True*
with *n-eq* **have** *read i1 n1* = *read i2 n2* **by** *auto*
with *True llist-eq splits trace-time-obs-LCons n-eq llist-eq.intros*(*1*)
show *?thesis* **by** *metis*
next
case *False*
let *?n1′* = *lhd* (*i1 n1*)
let *?n2′* = *lhd* (*i2 n2*)
from *llist-eq splits n-eq trace-time-obs-LCons*(*6*) *False*
obtain *i1′ i2′* **where** *cont*: *ns1* = *trace-time-obs′ S* (*k1′+1*) (*exec ?n1′ i1′*)
is-input i1′
input-obs-equiv S i1′ (*i1*(*n1*:=*ltl* (*i1 n1*)))
?n1′ ∈ *succs n1*
ns2 = *trace-time-obs′ S* (*k2′+1*) (*exec ?n2′ i2′*)
is-input i2′
input-obs-equiv S i2′ (*i2*(*n2*:=*ltl* (*i2 n2*)))
?n2′ ∈ *succs n2*
by *metis*
with *input-obs-equiv-def llist-eq n-eq*
have *input-equiv*: *input-obs-equiv S i1′ i2′* **by** *auto*
from *llist-eq cont n-eq input-obs-equiv-def obs1 input-nodes-def*
have *n′-gen*: *?n1′* = *?n2′* **by** (*cases n1* ∈ *input-nodes*) *auto*
with *llist-eq splits n-eq n′-reads cont input-equiv succs-valid*[*of ?n2′ n2*]
show *?thesis* **by** *auto*
qed
qed
qed
qed
with *trace-time-obs-equiv-def llist-eq-is-eq* ‹*k* = *0*›
have *trace-time-obs-equiv S* (*exec n i1*) (*exec n i2*) **by** *fastforce*
}
with *noninterferent-time-def* **show** *?thesis* **by** *auto*
qed

Proof of Theorem 3.3 (Soundness of Time-Sensitive Control Dependence).

theorem *tscd-slice-noninterferent-time*: **assumes** $S = \textit{backward-slice tscd } M$
shows *noninterferent-time S*
proof−
from *assms tscd-slice-sound combined-slice.intros* **have** *tcc S* **by** *auto*
with *tcc-noninterferent-time* **show** *noninterferent-time S* **by** *auto*
qed

lemma *M-subset-slice*: $M \subseteq \textit{combined-slice cd od } M$
using *combined-slice.intros* **by** *blast*

Proof of Corollary 3.1 Note that since $S \subseteq \textit{backward-slice tscd } S$, the premise is equivalent to $\textit{backward-slice tscd } S = S$.

theorem *tscd-slice-noninterferent-time'*: **assumes** $\textit{backward-slice tscd } S \subseteq S$
shows *noninterferent-time S*
proof−
from *assms M-subset-slice* **have** $\textit{backward-slice tscd } S = S$ **by** *blast*
with *tscd-slice-noninterferent-time* **show** *?thesis* **by** *blast*
qed

4.3.3 Minimality of Timing Sensitive Control Dependence

lemma *is-input-prepend*: **assumes** *is-input i*
$x \in \textit{succs } n$
shows *is-input* $(i(n{:=}\textit{LCons } x\ (i\ n)))$
using *assms is-input-def* **by** *auto*

lemma *trace-time-obs-shift*: $\textit{trace-time-obs}'\ S\ (k+k')\ ns$
$= \textit{lmap}\ (\lambda(k,\ n).\ (k+k',\ n))\ (\textit{trace-time-obs}'\ S\ k\ ns)$
proof−
have *pred-f*: $(\lambda p.\ \textit{fst}\ (\textit{snd}\ p) \in S)\ o\ (\lambda(k,\ n).\ (k+k',\ n)) = (\lambda p.\ \textit{fst}\ (\textit{snd}\ p) \in S)$ **by** *auto*
have $\textit{iterates Suc}\ (k+k') = \textit{lmap}\ (\lambda k.\ k+k')\ (\textit{iterates Suc}\ k)$ **by** (*coinduction arbitrary*: *k*) *force*
with *lzip-lmap1*[*of* $\lambda k.\ k+k'$ *iterates Suc k ns*]
lfilter-lmap[*of* $\lambda p.\ \textit{fst}\ (\textit{snd}\ p) \in S\ \lambda(k,\ n).\ (k+k',\ n)$, *unfolded pred-f*]
show *?thesis* **by** *auto*
qed

lemma *trace-time-obs-equiv-LCons*:
assumes *trace-time-obs-equiv S* (*LCons* (*n*,*n1*) *ns1*) (*LCons* (*n*,*n2*) *ns2*)
shows *trace-time-obs-equiv S ns1 ns2*
proof−
let $?f = (\lambda(k,\ n).\ (k+(1{::}\textit{nat}),\ n))$
from *assms trace-time-obs-equiv-def*
have $\textit{trace-time-obs}'\ S\ 0\ (\textit{LCons}\ (n,n1)\ ns1) = \textit{trace-time-obs}'\ S\ 0\ (\textit{LCons}\ (n,n2)\ ns2)$ **by** *auto*
with *iterates.code*[*of Suc 0*] **have** $\textit{trace-time-obs}'\ S\ 1\ ns1 = \textit{trace-time-obs}'\ S\ 1\ ns2$

by (*cases* $n \in S$) *auto*
with *trace-time-obs-shift*[*of S 0 1*] *llist.inj-map-strong*[*of - - ?f ?f*]
have *trace-time-obs' S 0 ns1* = *trace-time-obs' S 0 ns2* **by** *auto*
with *trace-time-obs-equiv-def* **show** *?thesis* **by** *auto*
qed

Helper function to generate a valid input.

fun *arbitrary-input* :: *'node* ⇒ *'node llist*
where *arbitrary-input n* = (*if succs n* = {} *then LNil else repeat* (*SOME x. x* ∈ *succs n*))

lemma *arbitrary-input-succs-infinite*: *succs n* ≠ {} ⟹ ¬ *lfinite* (*arbitrary-input n*)
using *lfinite-iterates* **by** *auto*

lemma *arbitrary-input-in-succs*: *n'* ∈ *lset* (*arbitrary-input n*) ⟹ *n'* ∈ *succs n*
using *someI*[*of* λ*x. x* ∈ *succs n*] **by** (*cases succs n* = {}) *auto*

Given a maximal path, generates a valid input whose execution results in that path.

primcorec *max-path-to-input* :: *'node llist* ⇒ *'node* ⇒ *'node llist*
where *max-path-to-input ns n* =
(*case ldropWhile* (λ*n'. n'* ≠ *n*) *ns of*
LNil ⇒ *arbitrary-input n*
| *LCons n1 LNil* ⇒ *arbitrary-input n*
| *LCons n1* (*LCons n2 ns'*) ⇒ *LCons n2* (*max-path-to-input* (*LCons n2 ns'*) *n*))

lemma *max-path-to-input-cases*:
assumes *max-path-to-input ns n* = *ms*
ldropWhile (λ*n'. n'* ≠ *n*) *ns* = *LNil* ⟹ *ms* = *arbitrary-input n* ⟹ *P*
⋀*n1. ldropWhile* (λ*n'. n'* ≠ *n*) *ns* = *LCons n1 LNil* ⟹ *ms* = *arbitrary-input n* ⟹ *P*
⋀*n1 n2 ns'. ldropWhile* (λ*n'. n'* ≠ *n*) *ns* = *LCons n1* (*LCons n2 ns'*)
⟹ *ms* = *LCons n2* (*max-path-to-input* (*LCons n2 ns'*) *n*)
⟹ *P*
shows *P*
proof−
show *?thesis*
proof (*cases ldropWhile* (λ*n'. n'* ≠ *n*) *ns*)
case *LNil*
with *assms max-path-to-input.code* **show** *?thesis* **by** *auto*
next
case (*LCons m1 ms'*)
with *assms max-path-to-input.code* **show** *?thesis* **by** (*cases ms'*) *auto*
qed
qed

lemma *ldropWhile-LCons*:

assumes *ldropWhile P xs* = *LCons x xs'*
obtains *xs1* **where** *xs* = *lappend* (*llist-of xs1*) (*LCons x xs'*) ¬ *P x*
proof−
from *assms ldropWhile-eq-LNil-iff* **have** *ex-not-P*: ∃ *x*∈*lset xs*. ¬ *P x* **by** *fastforce*
with *lfinite-ltakeWhile*[*of P*] *lfinite-eq-range-llist-of* **obtain** *xs1*
where *ltakeWhile P xs* = *llist-of xs1* **by** *auto*
from *this*[*symmetric*] **have** *xs* = *lappend* (*llist-of xs1*) (*ldropWhile P xs*) **by** *auto*
with *assms lhd-ldropWhile*[*OF ex-not-P*] *that* **show** *?thesis* **by** *auto*
qed

lemma *max-path-input*: **assumes** *max-path n ns*
shows *is-input* (*max-path-to-input ns*)
proof−
{
fix *m m'*
assume *m'* ∈ *lset* (*max-path-to-input ns m*)
with *lset-split* **obtain** *ns1 ns2*
where *max-path-to-input ns m* = *lappend* (*llist-of ns1*) (*LCons m' ns2*) **by**
metis
with *assms* **have** *m'* ∈ *succs m*
proof (*induction ns1 arbitrary*: *n ns*)
case (*Nil n ns*)
show *?thesis*
proof (*cases rule*: *max-path-to-input-cases*[*OF Nil*(*2*)])
case *1*
have *m'* ∈ *lset* (*LCons m' ns2*) **by** *auto*
with *arbitrary-input-in-succs 1* **show** *?thesis* **by** *auto*
next
case (*2 n1*)
have *m'* ∈ *lset* (*LCons m' ns2*) **by** *auto*
with *arbitrary-input-in-succs 2* **show** *?thesis* **by** *auto*
next
case (*3 n1 n2 ns'*)
from *ldropWhile-LCons*[*OF 3*(*1*)] **obtain** *ns1*
where *ns-split*: *ns* = *lappend* (*llist-of ns1*) (*LCons n1* (*LCons n2 ns'*))
n1 = *m* **by** *metis*
with *3 Nil max-path-split* **have** *max-path m* (*LCons m* (*LCons m' ns'*)) **by**
auto
from *this Nil max-path-hd* **show** *?thesis* **by** *cases auto*
qed
next
case (*Cons x ns1 n ns*)
show *?thesis*
proof (*cases rule*: *max-path-to-input-cases*[*OF Cons*(*3*)])
case *1*
have *m'* ∈ *lset* (*lappend* (*llist-of* (*x* # *ns1*)) (*LCons m' ns2*)) **by** *auto*
with *arbitrary-input-in-succs 1* **show** *?thesis* **by** *auto*
next
case (*2 n1*)

```
      have m′ ∈ lset (lappend (llist-of (x # ns1)) (LCons m′ ns2)) by auto
      with arbitrary-input-in-succs 2 show ?thesis by auto
    next
      case (3 n1 n2 ns′)
      from ldropWhile-LCons[OF 3(1)] obtain ns1′
        where ns-split: ns = lappend (llist-of ns1′) (LCons n1 (LCons n2 ns′))
by metis
      with lappend-llist-of-LCons
      have ns = lappend (llist-of (ns1′@[n1])) (LCons n2 ns′) by auto
      with 3 Cons max-path-split have max-path n2 (LCons n2 ns′) by auto
      with Cons 3 show ?thesis by auto
    qed
  qed
 }
 note set-succs = this
 {
  fix n
  assume succs n ≠ {}
  assume lfinite (max-path-to-input ns n)
  with lfinite-eq-range-llist-of obtain ns1
    where max-path-to-input ns n = llist-of ns1 by auto
  then have False
  proof (induction ns1 arbitrary: ns)
    case (Nil ns)
    from ‹succs n ≠ {}› iterates.code[of λx. x SOME x. x ∈ succs n]
    show ?thesis by (cases rule: max-path-to-input-cases[OF Nil]) auto
  next
    case (Cons n′ ns1)
    show ?thesis
    proof (cases rule: max-path-to-input-cases[OF Cons(2)])
      case 1
     with ‹succs n ≠ {}› arbitrary-input-succs-infinite lfinite-llist-of show ?thesis
by metis
    next
      case (2 n1)
     with ‹succs n ≠ {}› arbitrary-input-succs-infinite lfinite-llist-of show ?thesis
by metis
    next
      case (3 n1 n2 ns′)
      with Cons show ?thesis by auto
    qed
  qed
 }
 with set-succs is-input-def show ?thesis by metis
qed

lemma max-path-exec: assumes max-path n ns
          shows ns = lmap fst (exec n (max-path-to-input ns))
proof−
```

```
  from assms have llist-eq ns (lmap fst (exec n (max-path-to-input ns)))
  proof (coinduction arbitrary: n ns)
    case (llist-eq n ns)
    show ?case
    proof (cases succs n = {})
      case True
      with llist-eq max-path-no-succs have ns = LCons n LNil by auto
      from True exec.code have lmap fst (exec n (max-path-to-input ns)) = LCons
n LNil by auto
      with llist-eq-is-eq ‹ns = LCons n LNil› show ?thesis by auto
    next
      case False
      with llist-eq max-path-step obtain n′ ns′
        where ns-split: ns = LCons n ns′ max-path n′ ns′ by metis
      let ?i = max-path-to-input ns
      let ?i′ = ?i(n:=ltl (?i n))
      from ns-split max-path-LCons obtain ns′′ where ns′-split: ns′ = LCons n′
ns′′ by auto
      with ns-split have ldropWhile (λn′. n′ ≠ n) ns = LCons n (LCons n′ ns′′)
by auto
      with max-path-to-input.code[of ns n] ns′-split
      have input-n: ?i n = LCons n′ (max-path-to-input ns′ n) by auto
      {
        fix n2
        have ?i′ n2 = max-path-to-input ns′ n2
        proof (cases n2 = n)
          case True
          with input-n show ?thesis by auto
        next
          case False
          with ns-split max-path-to-input.code show ?thesis by auto
        qed
      }
      then have ?i′ = max-path-to-input ns′ by auto
      with input-n False exec.code[of n ?i]
      have lmap fst (exec n (max-path-to-input ns))
            = LCons n (lmap fst (exec n′ (max-path-to-input ns′))) by auto
      with ns-split show ?thesis by auto
    qed
  qed
  with llist-eq-is-eq show ?thesis by auto
qed

lemma at-pos-obs-lset: assumes at-pos k (lmap fst ns) m
  obtains m′ where (k,m,m′) ∈ lset (lzip (iterates Suc 0) ns)
proof−
  obtain k′ :: nat where k′ = 0 by simp
  from assms obtain m′ where (k+k′,m,m′) ∈ lset (lzip (iterates Suc k′) ns)
  proof (induction k arbitrary: k′ ns thesis)
```

case *0*
with *at-pos-def* **obtain** *n ns′* **where** *split*: *ns* = *LCons n ns′ fst n* = *m* **by** (*cases ns*) *auto*
then obtain *m′* **where** *n* = (*m*,*m′*) **by** (*cases n*) *simp*
with *0 iterates.code*[*of Suc k′*] *split* **show** *?case* **by** *auto*
next
case (*Suc k k′ ns thesis*)
with *at-pos-def* **obtain** *n ns′* **where** *split*: *ns* = *LCons n ns′* **by** (*cases ns*) *auto*
with *at-pos-succ Suc* **have** *at-pos k* (*lmap fst ns′*) *m* **by** *auto*
with *Suc*(*1*)[*of k′+1*] **obtain** *m′*
where (*k+k′+1*,*m*,*m′*) ∈ *lset* (*lzip* (*iterates Suc* (*k′+1*)) *ns′*) **by** *auto*
with *Suc iterates.code*[*of Suc k′*] *split* **show** *?case* **by** *auto*
qed
with ‹*k′* = *0*› *that* **show** *?thesis* **by** *auto*
qed

lemma *no-obs-after-k*: **assumes** (*k*,*m*,*m′*) ∈ *lset* (*lzip* (*iterates Suc k′*) *ns*)
k < *k′*
shows *False*
proof−
from *assms lset-split* **obtain** *ns1 ns2*
where *lzip* (*iterates Suc k′*) *ns* = *lappend* (*llist-of ns1*) (*LCons* (*k*,*m*,*m′*) *ns2*) **by** *metis*
with *assms*(*2*) **show** *?thesis*
proof (*induction ns1 arbitrary*: *ns k′*)
case *Nil*
with *iterates.code*[*of Suc k′*] **show** *?case* **by** (*cases ns*) *auto*
next
case (*Cons n ns1*)
with *iterates.code*[*of Suc k′*] *Cons*(*1*)[*of k′+1*] **show** *?case* **by** (*cases ns*) *auto*
qed
qed

lemma *lset-obs-at-pos*: **assumes** (*k*,*m*,*m′*) ∈ *lset* (*lzip* (*iterates Suc 0*) *ns*)
shows *at-pos k* (*lmap fst ns*) *m*
proof−
from *assms* **obtain** *k′* **where** (*k+k′*,*m*,*m′*) ∈ *lset* (*lzip* (*iterates Suc k′*) *ns*) *k′* = *0* **by** *auto*
from *this*(*1*) **show** *?thesis*
proof (*induction k arbitrary*: *k′ ns*)
case (*0 k′ ns*)
then obtain *n ns′* **where** *ns-split*: *ns* = *LCons n ns′* **by** (*cases ns*) *auto*
with *0 no-obs-after-k*[*of k′ m m′*] *iterates.code*[*of Suc k′*] *at-pos-def ns-split enat-0*
show *?case* **by** *auto*
next
case (*Suc k k′ ns*)
then obtain *n ns′* **where** *ns-split*: *ns* = *LCons n ns′* **by** (*cases ns*) *auto*

```
    with iterates.code[of Suc k'] Suc
    have (k+k'+1,m,m') ∈ lset (lzip (iterates Suc (k'+1)) ns') by auto
    with at-pos-succ Suc(1)[of k'+1] ns-split show ?case by auto
  qed
qed
```

Proof of Theorem 3.4 (Minimality of Time-Sensitive Control Dependence). In this version, the trace showing the violation of the non-interference criterion might start at any node of the graph.

```
theorem tscd-minimal: assumes ¬ (S' ⊇ backward-slice tscd M) (is ¬ (- ⊇ ?S))
                              M ⊆ S'
                      shows ¬ noninterferent-time S'
proof−
  from assms obtain n' where n' ∈ ?S n' ∉ S' by auto
   from this assms obtain n m where nm-gen: n ∉ S' m ∈ S' tscd n m by
induction auto
 with tscd-def obtain k x1 x2 where x-gen: x1 ∈ succs n ¬ on-max-paths-pos-k-first
x1 k m
                                       x2 ∈ succs n on-max-paths-pos-k-first x2 k m
    by auto
  with succs-valid have valid: valid-node n valid-node x2 by auto
  from on-max-paths-pos-k-first-def x-gen obtain ns
    where ns-gen: max-path x1 ns ¬ at-pos-first k ns m by auto
  with max-path-input max-path-exec obtain i
    where i-gen: is-input i ns = lmap fst (exec x1 i) by metis
  from i-gen is-input-max-path valid have max-path x2 (lmap fst (exec x2 i)) by
auto
  with at-pos-def at-pos-first-def x-gen on-max-paths-pos-k-first-def
  have at-pos-x2: at-pos k (lmap fst (exec x2 i)) m
              ∀ k'<k. ¬ at-pos k' (lmap fst (exec x2 i)) m by auto
  from ns-gen not-at-pos-first-to-at-pos have ¬ at-pos k ns m ∨ (∃ k'<k. at-pos k'
ns m) by auto
  then have trace-time-obs S' (exec x1 i) ≠ trace-time-obs S' (exec x2 i)
  proof
    assume ¬ at-pos k ns m
    from ‹m ∈ S'› at-pos-obs-lset[OF at-pos-x2(1)] obtain m'
      where m'-gen: (k,m,m') ∈ lset (trace-time-obs S' (exec x2 i)) by auto
    from lset-obs-at-pos[of k m m'] ‹¬ at-pos k ns m› ‹m ∈ S'› i-gen
    have (k,m,m') ∉ lset (trace-time-obs S' (exec x1 i)) by auto
    with m'-gen show ?thesis by metis
  next
    assume ∃ k'<k. at-pos k' ns m
    then obtain k' where at-pos k' ns m k' < k by auto
    with ‹m ∈ S'› at-pos-obs-lset[of k'] i-gen obtain m'
      where m'-gen: (k',m,m') ∈ lset (trace-time-obs S' (exec x1 i)) by auto
    from lset-obs-at-pos[of k' m m'] at-pos-x2 ‹m ∈ S'› ‹k' < k›
    have (k',m,m') ∉ lset (trace-time-obs S' (exec x2 i)) by auto
    with m'-gen show ?thesis by metis
  qed
```

with *trace-time-obs-equiv-def*
have *obs-not-equiv*: ¬ *trace-time-obs-equiv S′* (*exec x1 i*) (*exec x2 i*) **by** *auto*
let *?i1* = *i*(*n*:=*LCons x1* (*i n*))
let *?i2* = *i*(*n*:=*LCons x2* (*i n*))
from *input-obs-equiv-def nm-gen i-gen is-input-prepend x-gen*
have *inputs*: *input-obs-equiv S′ ?i1 ?i2 is-input ?i1 is-input ?i2* **by** *auto*
from *x-gen exec.code* **have** *exec n ?i1* = *LCons* (*n*, *Some x1*) (*exec x1 i*)
exec n ?i2 = *LCons* (*n*, *Some x2*) (*exec x2 i*) **by** *auto*
with *obs-not-equiv trace-time-obs-equiv-LCons*
have ¬ *trace-time-obs-equiv S′* (*exec n ?i1*) (*exec n ?i2*) **by** *metis*
with *valid inputs noninterferent-time-def* **show** *?thesis* **by** *blast*
qed

Proof of Theorem 3.4 (Minimality of Time-Sensitive Control Dependence). In this version, the trace showing the violation of the non-interference criterion has to start at the entry node. Here, we need to assume that every node is reachable from the entry node.

theorem *tscd-minimal-entry-node*:
assumes ¬ ($S′ \supseteq$ *backward-slice tscd Os*) (**is** ¬ (- $\supseteq$ *?S*))
Os $\subseteq$ *S′*
$\bigwedge n$. *valid-node n* $\Longrightarrow$ $\exists$ *ns*. *is-path* (*-Entry-*) *ns n*
obtains *i1 i2* **where** *is-input i1 is-input i2 input-obs-equiv S′ i1 i2*
¬ *trace-time-obs-equiv S′* (*exec* (*-Entry-*) *i1*) (*exec* (*-Entry-*) *i2*)
proof−
from *assms tscd-minimal noninterferent-time-def* **obtain** *i1 i2 n*
where *i-n-gen*: *valid-node n is-input i1 is-input i2 input-obs-equiv S′ i1 i2*
¬ *trace-time-obs-equiv S′* (*exec n i1*) (*exec n i2*) **by** *metis*
with *assms* **obtain** *ns* **where** *is-path* (*-Entry-*) *ns n* **by** *auto*
with *that i-n-gen* **show** *?thesis*
proof (*induction ns arbitrary*: *n i1 i2 rule*: *rev-induct*)
case (*snoc n′ ns′ n i1 i2*)
let *?i1′* = *i1*(*n′*:=*LCons n* (*i1 n′*))
let *?i2′* = *i2*(*n′*:=*LCons n* (*i2 n′*))
from *snoc*(*8*) *is-path-snoc succs-valid*
have *split*: *n* ∈ *succs n′ valid-node n′ is-path* (*-Entry-*) *ns′ n′* **by** *metis+*
with *snoc*(*4*,*5*) *is-input-prepend* **have** *is-input*: *is-input ?i1′ is-input ?i2′* **by** *auto*
from *snoc*(*6*) *input-obs-equiv-def* **have** *input-equiv*: *input-obs-equiv S′ ?i1′ ?i2′* **by** *auto*
from *split exec.code* **have** *exec n′ ?i1′* = *LCons* (*n′*, *Some n*) (*exec n i1*)
exec n′ ?i2′ = *LCons* (*n′*, *Some n*) (*exec n i2*) **by** *auto*
with *trace-time-obs-equiv-LCons snoc*(*7*)
have ¬ *trace-time-obs-equiv S′* (*exec n′ ?i1′*) (*exec n′ ?i2′*) **by** *metis*
with *snoc split is-input input-equiv* **show** *?case* **by** *blast*
qed *auto*
qed

5 Proofs for the Algorithm section

5.1 Postdominance Frontiers

Definition 5.2, part 1. *spdom* = *1*−⊑-Postdominance.

abbreviation *spdom pdrel n m* == $\exists$ *m*′≠*m*. *pdrel n m*′ ∧ *pdrel m*′ *m*

Definition 5.2, part 2.

abbreviation *ipdom pdrel n* == {*m*. *spdom pdrel n m* ∧ (∀ *m*′. *spdom pdrel n m*′ ⟶ *pdrel m m*′)}

Definition 5.3.

abbreviation *pdf pdrel m* == {*n*. ¬ *spdom pdrel n m* ∧ (∃ *x*∈*succs n*. *pdrel x m*)}

```
lemma on-max-paths-step: assumes on-max-paths n m
                                 n ≠ m
                                 x ∈ succs n
                         shows on-max-paths x m
proof−
  {
    fix ns
    assume max-path x ns
    with assms max-path.intros on-max-paths-def have m ∈ lset ns by fastforce
  }
  with on-max-paths-def show ?thesis by blast
qed

lemma on-sink-paths-step: assumes on-sink-paths n m
                                  n ≠ m
                                  x ∈ succs n
                          shows on-sink-paths x m
proof−
  {
    fix ns
    assume sink-path x ns
    with assms succs-path path-sink-path-append on-sink-paths-def have m ∈ lset
ns by fastforce
  }
  with on-sink-paths-def show ?thesis by auto
qed
```

Ntscd part of Lemma 5.1

```
theorem ntscd-on-max-paths-frontier:
  assumes n ≠ m
  shows n ∈ pdf on-max-paths m ⟷ ntscd n m
proof
  assume n ∈ pdf on-max-paths m
  with assms on-max-paths-refl ntscd-cond-succ show ntscd n m by fast
```

next
  assume *ntscd n m*
  with *ntscd-def* **obtain** *x1 x2* **where** *x1* ∈ *succs n x2* ∈ *succs n*
    on-max-paths x1 m ¬ *on-max-paths x2 m* **by** *auto*
   with *on-max-paths-step assms on-max-paths-trans* **show** *n* ∈ *pdf on-max-paths m* **by** *fast*
qed

lemma *nticd-cond-succ*: **assumes** *finite* (*Collect valid-node*)
  ¬ *on-sink-paths p n*
  x ∈ *succs p*
  on-sink-paths x n
  shows *nticd p n*
proof−
  from *assms on-sink-ext-paths-equiv on-ext-paths-def* **obtain** *ns n′*
   where *ext*: *is-path p ns n′* ∀ *ns′ n″*. *is-path n′ ns′ n″* ⟶ *n* ∉ *set* (*ns*@*ns′*@[*n″*]) **by** *metis*
  have ∃ *x2*∈*succs p*. ¬ *on-ext-paths x2 n*
  proof (*cases ns*)
    case *Nil*
    from *assms on-sink-ext-paths-equiv on-ext-paths-ex succs-valid* **obtain** *ns′*
      where *is-path x ns′ n* **by** *metis*
    with *Nil assms succs-path-extend ext* **show** *?thesis* **by** *fastforce*
  next
    case (*Cons p′ ns2*)
    with *ext is-path-Cons* **obtain** *x2*
      where *x2-gen*: *p′* = *p x2* ∈ *succs p is-path x2 ns2 n′* **by** *blast*
    from *ext Cons* **have** ∀ *ns′ n″*. *is-path n′ ns′ n″* ⟶ *n* ∉ *set* (*ns2*@*ns′*@[*n″*]) **by** *auto*
    with *x2-gen on-ext-paths-def* **show** *?thesis* **by** *metis*
  qed
  with *assms on-sink-ext-paths-equiv nticd-def* **show** *?thesis* **by** *auto*
qed

Nticd part of Lemma 5.1

theorem *nticd-on-max-paths-frontier*:
  assumes *finite* (*Collect valid-node*)
    $n \neq m$
  shows *n* ∈ *pdf on-sink-paths m* ⟷ *nticd n m*
proof
  assume *n* ∈ *pdf on-sink-paths m*
  with *assms on-sink-paths-refl nticd-cond-succ* **show** *nticd n m* **by** *fast*
next
  assume *nticd n m*
  with *nticd-def* **obtain** *x1 x2* **where** *x1* ∈ *succs n x2* ∈ *succs n*
    on-sink-paths x1 m ¬ *on-sink-paths x2 m* **by** *auto*
   with *on-sink-paths-step assms on-sink-paths-trans* **show** *n* ∈ *pdf on-sink-paths m* **by** *fast*
qed

Definition 5.5, part 1.

abbreviation *closedG pdrel* == ∀ *n x m*. *x* ∈ *succs n* ∧ *pdrel n m* ∧ *n* ≠ *m* ⟶ *pdrel x m*

Definition 5.5, part 2.

abbreviation *noJoin pdrel* == ∀ *n m1 m2 m12*. (*m12* ∈ *ipdom pdrel m1* ∧ *m12* ∈ *ipdom pdrel m2*
∧ *pdrel n m1* ∧ *pdrel n m2* ∧ *m1* ≠ *m2* ∧ *valid-node n*)
⟶ *m1* ∈ *ipdom pdrel m2* ∨ *m2* ∈ *ipdom pdrel m1*

Part of Lemma 5.2: $\sqsubseteq_{MAX}$ is closed under $\rightarrow_G$.

theorem *on-max-paths-closedG*: *closedG on-max-paths*
using *on-max-paths-step* **by** *auto*

Part of Lemma 5.2: $\sqsubseteq_{SINK}$ is closed under $\rightarrow_G$.

theorem *on-sink-paths-closedG*: *closedG on-sink-paths*
using *on-sink-paths-step* **by** *auto*

abbreviation *linearizable pdrel* == ∀ *n m1 m2*. *valid-node n* ∧ *pdrel n m1* ∧ *pdrel n m2*
⟶ *pdrel m1 m2* ∨ *pdrel m2 m1*

"linearize" lemma to be instantiated with $\sqsubseteq_{MAX}$ and $\sqsubseteq_{SINK}$.

lemma *on-all-paths-linearize*: **assumes** *closedG P*
⋀*n m*. *P n m* ⟹ *valid-node n* ⟹ ∃ *ns*. *is-path n ns m*
shows *linearizable P*
proof−
{
fix *n m1 m2*
assume *assms2*: *valid-node n P n m1 P n m2*
with *assms* **obtain** *ns* **where** *is-path n ns m2* **by** *metis*
with *assms assms2* **have** *P m1 m2* ∨ *P m2 m1*
proof (*induction ns arbitrary*: *n*)
case (*Cons a ns n*)
with *is-path-Cons Cons* **show** *?case* **by** *blast*
qed *auto*
}
then show *?thesis* **by** *auto*
qed

lemma *linearizable-noJoin*: **assumes** *linearizable P*
⋀*n m1 m2*. *P n m1* ⟹ *P m1 m2* ⟹ *P n m2*
⋀*n*. *P n n*
shows *noJoin P*
proof−

{
fix *n m1 m2 m12*
assume *assms2*: *m12* ∈ *ipdom P m1 m12* ∈ *ipdom P m2 P n m1 P n m2 m1* ≠ *m2 valid-node n*
with *assms* **have** *P m1 m2* ∨ *P m2 m1* **by** *blast*
with *assms2* **obtain** *m1′ m2′*
where *m′-gens*: *m12* ∈ *ipdom P m1′ m12* ∈ *ipdom P m2′ P n m1′ P n m2′ m1′* ≠ *m2′ P m1′ m2′ m1′* ∈ {*m1*, *m2*} *m2′* ∈ {*m1*, *m2*}
by *blast*
{
fix *m′*
assume *spdom P m1′ m′*
with *m′-gens assms* **have** *P m12 m′* ∧ *P m2′ m12* **by** *blast*
with *assms* **have** *P m2′ m′* **by** *blast*
}
with *assms m′-gens*(*5*,*6*) **have** *m2′* ∈ *ipdom P m1′* **by** *blast*
with *m′-gens* **have** *m1* ∈ *ipdom P m2* ∨ *m2* ∈ *ipdom P m1* **by** *auto*
}
then show *?thesis* **by** *blast*
qed

"linearize" lemma for $\sqsubseteq_{MAX}$.

lemma *on-max-paths-linearize*: *linearizable on-max-paths*
using *on-all-paths-linearize on-max-paths-step on-max-paths-ex-path* **by** *blast*

Part of Lemma 5.2: $\sqsubseteq_{MAX}$ lacks joins.

theorem *on-max-path-noJoin*: *noJoin on-max-paths*
using *on-max-paths-refl on-max-paths-trans linearizable-noJoin*[*OF on-max-paths-linearize*]
by *blast*

"linearize" lemma for $\sqsubseteq_{SINK}$.

lemma *on-sink-paths-linearize*: **assumes** *finite* (*Collect valid-node*)
shows *linearizable on-sink-paths*
proof−
from *assms on-ext-paths-ex on-sink-ext-paths-equiv*
have $\bigwedge$*n m*. *on-sink-paths n m* ⟹ *valid-node n* ⟹ ∃ *ns*. *is-path n ns m* **by** *blast*
with *assms on-all-paths-linearize on-sink-paths-step* **show** *?thesis* **by** *blast*
qed

Part of Lemma 5.2: $\sqsubseteq_{SINK}$ lacks joins.

theorem *on-sink-path-noJoin*: **assumes** *finite* (*Collect valid-node*)
shows *noJoin on-sink-paths*
proof−
from *assms on-sink-paths-linearize* **have** *linearizable on-sink-paths* **by** *simp*
from *on-sink-paths-refl on-sink-paths-trans*[*OF assms*] *linearizable-noJoin*[*OF this*]
show *?thesis* **by** *blast*
qed

5.2 Transitive Reductions and Pseudo-forests

Theorems for the properties of the transitive, reflexive reductions (see Observation 5.1).

We will not give a full mechanized proof here due to the complexity of formalizing transitive, reflexive reductions.

We will however prove lemmas here and give a pen-and-paper proof on why they imply those properties.

For $\sqsubseteq \in \{\sqsubseteq_{MAX}, \sqsubseteq_{SINK}\}$, we will need linearizable: $m1 \sqsubseteq n \Longrightarrow m2 \sqsubseteq n \Longrightarrow m2 \sqsubseteq m1 \vee m1 \sqsubseteq m2$ and scc: $n \neq m1 \Longrightarrow m1 \sqsubseteq n \Longrightarrow m2 \sqsubseteq n \Longrightarrow n \sqsubseteq m1 \Longrightarrow n \sqsubseteq m2$. The linearizable part has already been proved in the previous section, the scc part will be proved in this section.

Now, assume we have $m1 < n$ and $m2 < n$. (with $<$ being the corresponding transitive, reflexive reduction of $\sqsubseteq$ (*)). Then from (*) we have $m1 \sqsubseteq n$ and $m2 \sqsubseteq n$. With "linearize", we have $m2 \sqsubseteq m1 \vee m1 \sqsubseteq m2$ (w.l.o.g. let $m2 \sqsubseteq m1$ be true). This means we have (via (*), $m1 \sqsubseteq n$ and $m2 \sqsubseteq m1$) a path in the "<-graph" from n to $m2$. But since $m2 < n$ and (*), this path must contain the $m2 < n$ edge. But then $n \sqsubseteq m1$, and "scc" gives us $n \sqsubseteq m2$ (note $m1 < n$ and (*) gives us $n \neq m1$). Thus, n, $m1$ and $m2$ belong to the same SCC of the "¡-graph". In any transitive, reflexive reduction, the nodes of an SCC in the original graph form a cycle without other edges between them (Theorem 2 of "The Transitive Reduction of a Directed Graph" by Aho, Alfred and R. Garey, M and Ullman, Jeffrey (doi 10.1137/0201008)). But then $m1 = m2$.

"scc" lemma to be instantiated with $\sqsubseteq_{MAX}$ and $\sqsubseteq_{SINK}$.

lemma *on-all-paths-scc*: **assumes** *closedG P*
$\bigwedge n\ m.\ P\ n\ m \Longrightarrow \textit{valid-node}\ n \Longrightarrow \exists\, ns.\ \textit{is-path}\ n\ ns\ m$
$\bigwedge n\ m1\ m2.\ P\ n\ m1 \Longrightarrow P\ m1\ m2 \Longrightarrow P\ n\ m2$
$\bigwedge n.\ P\ n\ n$
valid-node n $n \neq m1$ $P\ n\ m1$ $P\ n\ m2$ $P\ m1\ n$
shows $P\ m2\ n$

proof−
from *assms* **obtain** *ns* **where** *path*: *is-path n ns m2* **by** *metis*
show *?thesis*
proof (*cases ns*)
case *Nil*
with *path assms*(*4*) **show** *?thesis* **by** *simp*
next
case *Cons*
with *path is-path-Cons* **have** $n \in \textit{set}\ ns$ **by** *auto*
with *split-list-last* **obtain** *ns1 ns2* **where** *ns-split*: $ns = ns1 @ n \# ns2$ $n \notin \textit{set}\ ns2$ **by** *metis*
with *path is-path-split* **have** *is-path n* ($n\#ns2$) *m2* **by** *blast*
with *is-path-Cons* **obtain** *x* **where** *x-gen*: $x \in \textit{succs}\ n$ *is-path x ns2 m2* **by** *blast*

with *assms* **have** *P x n* **by** *blast*
with *x-gen*(*2*) *ns-split*(*2*) **show** *?thesis*
proof (*induction ns2 arbitrary*: *x*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons a ns2 x*)
with *is-path-Cons* **obtain** *y* **where** $a = x$ $y \in$ *succs x is-path y ns2 m2* **by** *blast*
with *assms*(*1*) *Cons* **show** *?case* **by** *auto*
qed
qed
qed

"scc" lemma for $\sqsubseteq_{MAX}$.

lemma *on-max-paths-scc*: **assumes** *valid-node n*
$n \neq m1$
on-max-paths n m1
on-max-paths n m2
on-max-paths m1 n
shows *on-max-paths m2 n*
using *assms on-all-paths-scc*[*of on-max-paths n m1 m2*] *on-max-paths-step on-max-paths-ex-path on-max-paths-refl on-max-paths-trans* **by** *blast*

"scc" lemma for $\sqsubseteq_{SINK}$.

lemma *on-sink-paths-scc*: **assumes** *finite* (*Collect valid-node*)
valid-node n
$n \neq m1$
on-sink-paths n m1
on-sink-paths n m2
on-sink-paths m1 n
shows *on-sink-paths m2 n*
proof−
from *assms on-ext-paths-ex on-sink-ext-paths-equiv*
have $\bigwedge n\ m.$ *on-sink-paths n m* $\Longrightarrow$ *valid-node n* $\Longrightarrow \exists\, ns.$ *is-path n ns m* **by** *blast*
with *assms on-all-paths-scc*[*of on-sink-paths n m1 m2*] *on-sink-paths-step on-sink-paths-refl on-sink-paths-trans* **show** *?thesis* **by** *blast*
qed

5.3 Transitivity results

5.3.1 Reducible Graphs

To define reducibility, we need an additional assumption that every node is reachable from the entry node.

context
assumes *Entry-path*: $\bigwedge n.$ *valid-node n* $\Longrightarrow \exists\, ns.$ *is-path* (*-Entry-*) *ns n*

assumes *reducible*: $\bigwedge n\ ns.\ is\text{-}path\ n\ ns\ n \wedge ns \neq []$
$\longrightarrow (\exists\, m \in set\ ns.\ \forall\, m' \in set\ ns.\ \forall\, ns'.\ is\text{-}path\ (\text{-}Entry\text{-})\ ns'\ m'$
$\longrightarrow m \in set\ (ns'@[m']))$
begin

Definition of Weak Order Dependency. Not used in any results given in the article, but an important definition to make proofs about reducible graphs easier.

definition *wod* :: *'node* ⇒ *'node* ⇒ *'node* ⇒ *bool*
where *wod n m1 m2* == *m1* ≠ *m2*
∧ (∃ *ms1*. *is-path n ms1 m1* ∧ *m2* ∉ *set ms1*)
∧ (∃ *ms2*. *is-path n ms2 m2* ∧ *m1* ∉ *set ms2*)
∧ (∃ *x* ∈ *succs n*. *on-max-paths-prev x m1 m2* ∨ *on-max-paths-prev x m2 m1*)

lemma *on-max-path-prev-non-step-wod*: **assumes** *on-max-paths n m1*
x ∈ *succs n*
on-max-paths-prev x m1 m2
¬ *on-max-paths-prev n m1 m2*
n ≠ *m2*
m1 ≠ *m2*
shows *wod n m1 m2*
proof−
from *assms succs-valid on-max-paths-prev-split* **obtain** *ns11*
where *ns1-split*: *is-path x ns11 m1 m2* ∉ *set ns11* **by** *metis*
with *succs-path-extend assms* **have** *path1*: *is-path n* (*n#ns11*) *m1* **by** *blast*
from *assms on-max-paths-not-prev* **obtain** *ns2* **where** *is-path n ns2 m2 m1* ∉ *set ns2* **by** *metis*
with *path1 ns1-split assms wod-def* **show** *?thesis* **by** *auto*
qed

lemma *paths-order-ntscd-tranclp*: **assumes** *is-path p pns n*
m ∉ *set pns*
is-path p pms m
n ∉ *set pms*
x ∈ *succs p*
n ≠ *m*
on-max-paths-prev x n m
shows $ntscd^{**}\ p\ m \vee ntscd^{**}\ p\ n$
proof (*clarify*)
assume ¬ $ntscd^{**}\ p\ n$
from *max-path-ext assms succs-valid* **have** *max-ext-x*: *max-path x* (*ext-max-path x*) **by** *auto*
from *assms on-max-paths-prev-split succs-valid* **obtain** *xns*
where *xns-gen*: *is-path x xns n n* ∉ *set xns m* ∉ *set xns* **by** *metis*
from *path-first*[*OF assms*(*1*)] **obtain** *ns ns'*
where *pn-path*: *is-path p ns n pns* = *ns@ns'* **by** *blast*
with *assms* **have** *m* ∉ *set ns* **by** *auto*

from *path-first*[*OF assms*(*3*)] **obtain** *ms ms'*
where *pm-path*: *is-path p ms m m* ∉ *set ms pms* = *ms*@*ms'* **by** *auto*
with *assms* **have** *n* ∉ *set ms* **by** *auto*
have *ms* ≠ []
proof
assume *ms* = []
with *path-empty-conv pm-path* **have** *p* = *m* **by** *auto*
with *path-empty-conv assms pn-path* **have** *ns* ≠ [] **by** *auto*
with ‹*m* ∉ *set ns*› *path-cons-conv*[*of - p*] ‹*p* = *m*› *pn-path* **show** *False* **by** (*cases ns*) *auto*
qed
from *assms on-max-paths-def on-max-paths-prev-def* **have** *on-max-paths x n* **by** *auto*
with *assms ntscd-cond-succ* ‹¬ *ntscd*** *p n*› **have** *max-paths*: *on-max-paths p n* **by** *auto*
from *is-path-valid-node*[*OF pm-path*(*1*)] *max-path-ext*
have *max-ext-m*: *max-path m* (*ext-max-path m*) **by** *auto*
with *pm-path max-path-append* **have** *max-path p* (*lappend* (*llist-of ms*) (*ext-max-path m*)) **by** *auto*
with ‹*n* ∉ *set ms*› *max-paths on-max-paths-def* **have** *n* ∈ *lset* (*ext-max-path m*) **by** *auto*
from *lset-split*[*OF this*] **obtain** *ens1 ens2*
where *ext-max-path m* = *lappend* (*llist-of ens1*) (*LCons n ens2*) **by** *auto*
with *max-ext-m max-path-split* **have** *path-mns*: ∃ *mns*. *is-path m mns n* **by** *simp blast*
show *ntscd*** *p m*
proof (*cases* ∃ *nms*. *is-path n nms m*)
case *False*
{
fix *m'*
assume *m'-gen*: *m'* ∈ *set* (*m*# *rev ms*) *m'* ≠ *p on-max-paths p m'*
with *on-max-paths-step assms* **have** *on-max-paths x m'* **by** *auto*
with *max-ext-x on-max-paths-def* **have** *m'* ∈ *lset* (*ext-max-path x*) **by** *auto*
with *max-path-split-elem max-ext-x* **obtain** *ms1'* **where** *path-xm'*: *is-path x ms1' m'* **by** *metis*
obtain *ms3'* **where** *is-path m' ms3' m*
proof (*cases m*=*m'*)
case *True*
with *path0 is-path-valid-node*[*OF path-xm'*] *that*[*of* []] **show** *?thesis* **by** *auto*
next
case *False*
with *m'-gen* **have** *m'* ∈ *set ms* **by** *auto*
with *path-split-elem pm-path*(*1*) *that* **show** *?thesis* **by** *blast*
qed
with *path-xm' path-append* **have** *is-path x* (*ms1'*@*ms3'*) *m* **by** *auto*
with *on-max-paths-prev-ccontr*[*OF assms*(*7*,*6*) *this*] **have** *n* ∈ *set* (*ms1'*@*ms3'*) **by** *auto*
with *path-split-elem* ‹*is-path x* (*ms1'*@*ms3'*) *m*› *False* **have** *False* **by** *blast*
}

    with ntscd-rtranclpI[OF pm-path(1)] show ?thesis by auto
  next
    case True
    with assms path-end-unique obtain nms
    where cycle1: is-path n (n#nms) m n ∉ set nms m ∉ set nms by blast
    from path-end-unique path-mns assms obtain mns
    where cycle2: is-path m (m#mns) n m ∉ set mns n ∉ set mns by blast
    let ?cs = n#nms@m#mns
    from path-append[OF cycle1(1) cycle2(1)] have is-path n ?cs n by auto
    with reducible[of n ?cs] obtain d where dom: d ∈ set ?cs
      ∀m′∈set ?cs. ∀ns. is-path (-Entry-) ns m′ ⟶ d ∈ set (ns @ [m′]) by auto
    from Entry-path assms obtain ps where entry-p-path: is-path (-Entry-) ps p
by auto
    have dom-path: d ∈ set (ps@[p])
    proof (rule ccontr)
      assume d ∉ set (ps@[p])
      from pm-path entry-p-path path-append succs-path-extend assms xns-gen(1)
      have is-path (-Entry-) (ps@ms) m is-path (-Entry-) (ps@p#xns) n by auto
      with dom ‹d ∉ set (ps@[p])› have d ∈ set (ms@[m]) d ∈ set (xns@[n]) by
auto
      with ‹m ∉ set xns› ‹n ∉ set ms› assms(6) have d-elem: d ∈ set ms d ∈ set
xns by auto
      with path-split-elem xns-gen obtain ns1 ns2
      where xns-d-split: xns = ns1@d#ns2 is-path x ns1 d by blast
      from d-elem path-split-elem pm-path obtain ms1 ms2
      where ms = ms1@d#ms2 is-path d (d#ms2) m by blast
      with xns-d-split ‹n ∉ set xns› ‹n ∉ set ms› path-append
      have is-path x (ns1@d#ms2) m n ∉ set (ns1@d#ms2) by auto
      from on-max-paths-prev-ccontr[OF assms(7,6) this] show False .
    qed
    obtain dps where dps-gen: is-path d dps p
    proof (cases d ∈ set ps)
      case True
      with path-split-elem entry-p-path that show ?thesis by blast
    next
      case False
      with dom-path assms path0[of - p] that[of []] show ?thesis by auto
    qed
    obtain c cs where c-gen: is-path c cs p c ∈ set ?cs ∀c′∈set (tl cs). c′ ∉ set
?cs
    proof (cases dps)
      case Nil
      with dom that[OF dps-gen] show ?thesis by auto
    next
      case (Cons d′ dps′)
      with path-cons-conv[of - d] dps-gen dom have ∃c∈set dps. c ∈ set ?cs by
auto
      from split-list-last-propE[OF this] obtain cs1 c cs2
      where cs-gen: dps = cs1@c#cs2 c ∈ set ?cs ∀c′∈set cs2. c′ ∉ set ?cs by

auto
  with *is-path-split*[*OF dps-gen*[*unfolded this*(*1*)]] *that* **show** *?thesis* **by** *auto*
 qed
 with *path-cons-conv*[*of - c*] **have** *n-set-cs*: $n \neq c \Longrightarrow n \notin set\ cs$ **by** (*cases cs*) *auto*
 {
  fix *pps*
  assume *pcs-gen*: $n \notin set\ pps$ *is-path p pps p* $pps \neq []$
  with *pcs-gen cycle-max-path-neq-nil* **have** *max-path p* (*cycle pps*) **by** *auto*
  with *max-paths on-max-paths-def cycle-lset*[*of pps*] *pcs-gen* **have** *False* **by** *auto*
 }
 note *cycle-ccontr* = *this*
 show *?thesis*
 proof (*cases* $c \in set\ (m\#mns)$)
  case *True*
  with *path-split-elem cycle2* **obtain** *mcs cns*
  where *mns-split*: *is-path m mcs c* $m\#mns = mcs@c\#cns$ **by** *blast*
  have *False*
  proof (*rule cycle-ccontr*)
   from *mns-split path-append pm-path c-gen* **show** *is-path p* (*ms@mcs@cs*) *p* **by** *auto*
   from *assms cycle2 True* **have** $n \notin set\ (m\#mns)$ **by** *auto*
   with *mns-split* ‹$ms \neq []$› *n-set-cs* ‹$n \notin set\ ms$›
   show $ms@mcs@cs \neq []$ $n \notin set\ (ms@mcs@cs)$ **by** *auto*
  qed
  thus *?thesis* **..**
 next
  case *False*
  with *c-gen* **have** $c \in set\ (n\#nms)$ **by** *simp*
  with *path-split-elem cycle1* **obtain** *ncs cms*
  where *nms-split*: *is-path n ncs c* $n\#nms = ncs@c\#cms$ **by** *blast*
  {
   fix m'
   assume m'*-gen*: $m' \in set\ (m\#rev\ ms)$ $m' \neq p$ *on-max-paths p* m'
   with *on-max-paths-step assms* **have** *on-max-paths x* m' **by** *auto*
   from m'*-gen assms* ‹$n \notin set\ ms$› **have** $m' \neq n$ **by** *auto*
   obtain mms' pms'
   where *ms-split*: *is-path* m' mms' *m* $n \notin set\ mms'$ *is-path p* pms' m' $n \notin set\ pms'$
   proof (*cases* $m=m'$)
    case *True*
    with *path0 is-path-valid-node*[*OF assms*(*3*)] *that*[*of* []] *pm-path* ‹$n \notin set\ ms$›
    show *?thesis* **by** *auto*
   next
    case *False*
    with m'*-gen* **have** $m' \in set\ ms$ **by** *auto*
    with *path-split-elem pm-path*(*1*) **obtain** *ms1 ms2*

where *ms* = *ms1*@*m'*#*ms2* *is-path m'* (*m'*#*ms2*) *m is-path p ms1 m'* **by** *blast*
with *that* ‹*n* ∉ *set ms*› **show** *?thesis* **by** *auto*
qed
from *xns-gen nms-split c-gen succs-path*[*OF assms*(*5*)] *path-append*
have *is-path x* (*xns*@*ncs*@*cs*@[*p*]) *x* **by** *auto*
with *cycle-max-path-neq-nil* **have** *max-path x* (*cycle* (*xns*@*ncs*@*cs*@[*p*])) **by** *auto*
with ‹*on-max-paths x m'*› *on-max-paths-def cycle-lset*[*of xns*@*ncs*@*cs*@[*p*]]
have *m'* ∈ *set* (*xns*@*ncs*@*cs*@[*p*]) **by** *auto*
have *False*
proof (*cases m'* ∈ *set xns*)
case *True*
with *path-split-elem xns-gen* **obtain** *xms' xms''*
where *is-path x xms' m' xns* = *xms'*@*m'*#*xms''* **by** *blast*
with *path-append ms-split xns-gen*
have *is-path x* (*xms'*@*mms'*) *m n* ∉ *set* (*xms'*@*mms'*) **by** *auto*
with *on-max-paths-prev-ccontr*[*OF assms*(*7*,*6*)] **show** *?thesis* **by** *blast*
next
case *False*
with ‹*m'* ∈ *set* (*xns*@*ncs*@*cs*@[*p*])› *m'-gen* **have** *m'* ∈ *set* (*ncs*@*cs*) **by** *auto*
then obtain *n' nps'* **where** *nps'-gen*: *ncs*@*cs* = *n'*#*nps'* **by** (*cases ncs*@*cs*) *auto*
with *path-append*[*OF nms-split*(*1*) *c-gen*(*1*)] **have** *is-path n* (*n'*#*nps'*) *p* **by** *auto*
with *nps'-gen path-cons-conv*[*of edge-rel n n'*] *edge-rel-def succs-valid* **obtain** *n2*
where *nps'-path*: *n*=*n' is-path n2 nps' p* **by** *blast*
with ‹*m'* ∈ *set* (*ncs*@*cs*)› *nps'-gen* ‹*m'*≠*n*› **have** *m'* ∈ *set nps'* **by** *auto*
with *path-split-elem nps'-path* **obtain** *nps1 nps2*
where *nps'-split*: *nps'* = *nps1*@*m'*#*nps2* *is-path m'* (*m'*#*nps2*) *p* **by** *blast*
have *n* ∉ *set nps'*
proof (*cases ncs*)
case *Nil*
with *nps'-gen c-gen*(*3*) **show** *?thesis* **by** *auto*
next
case (*Cons a list*)
with *nms-split*(*2*) *cycle1*(*2*) *nps'-gen n-set-cs* **show** *?thesis* **by** *force*
qed
with ‹*nps'* = *nps1*@*m'*#*nps2*› **have** *n-not-elem*: *n* ∉ *set* (*m'*#*nps'*) **by** *auto*
show *?thesis*
proof (*rule cycle-ccontr*)
from *n-not-elem nps'-split ms-split path-append*
show *n* ∉ *set* (*pms'*@*m'*#*nps2*) *is-path p* (*pms'*@*m'*#*nps2*) *p pms'*@*m'*#*nps2* ≠ [] **by** *auto*
qed
qed

```
      }
      with ntscd-rtranclpI[OF pm-path(1)] show ?thesis by auto
    qed
  qed
qed

lemma reducible-wod-imp-ntscd-tranclp: assumes wod n m1 m2
                                       shows ntscd** n m1 ∨ ntscd** n m2
proof−
  from assms wod-def obtain ms1 ms2
  where order-paths: is-path n ms1 m1 m2 ∉ set ms1 is-path n ms2 m2 m1 ∉ set
ms2 by auto
  from assms wod-def obtain x
  where m1 ≠ m2 x ∈ succs n on-max-paths-prev x m1 m2 ∨ on-max-paths-prev
x m2 m1 by auto
  with paths-order-ntscd-tranclp order-paths show ?thesis by blast
qed

lemma ntscd-not-on-max-paths: assumes ntscd n m
                                       n ≠ m
                                 shows ¬ on-max-paths n m
  using assms ntscd-def on-max-paths-step by blast

lemma ntscd-rtrancl-not-on-max-paths: assumes ntscd** n m
                                               n ≠ m
                                        shows ¬ on-max-paths n m
proof
  assume on-max-paths n m
  with assms show False
  proof (induction rule: converse-rtranclp-induct)
    case (step x y)
    show ?case
    proof (cases y = m)
      case True
      with step ntscd-not-on-max-paths show ?thesis by auto
    next
      case False
      with step have ¬ on-max-paths y m x ≠ y by auto
      with on-max-paths-def obtain ns where ns-gen: max-path y ns m ∉ lset ns
by auto
      from step ntscd-def obtain x1 where x1-gen: on-max-paths x1 y x1 ∈ succs
x by auto
      with on-max-paths-ex-path succs-valid path-first obtain ns1
        where ns1-gen: is-path x1 ns1 y y ∉ set ns1 by metis
      with succs-path-extend x1-gen max-path-append ns-gen
      have max-path x (lappend (llist-of (x#ns1)) ns) by blast
      with step on-max-paths-def ns1-gen ns-gen have m ∈ set ns1 by auto
      with ns1-gen path-split-elem obtain ns1' ns1''
        where ns1-split: is-path x1 ns1' m ns1 = ns1'@m#ns1'' by metis
```

from *step ntscd-def* **obtain** *x2* **where** *x2* ∈ *succs x* ¬ *on-max-paths x2 y* **by** *auto*
with *on-max-paths-def* **obtain** *ns2*
where *ns2-gen*: *max-path x2 ns2 y* ∉ *lset ns2* **by** *auto*
with *max-path.intros(2)* ‹*x2* ∈ *succs x*› *step(4,5)* *on-max-paths-def*
have *m* ∈ *lset ns2* **by** *fastforce*
with *ns2-gen max-path-split-elem* **obtain** *ns2′ ns2″*
where *ns2-split*: *max-path m (LCons m ns2″)*
ns2 = lappend (llist-of ns2′) (LCons m ns2″) **by** *metis*
with *ns1-split ns1-gen ns2-gen max-path-append*
have *max-path x1 (lappend (llist-of ns1′) (LCons m ns2″))*
y ∉ *lset (lappend (llist-of ns1′) (LCons m ns2″))* **by** *auto*
with *x1-gen on-max-paths-def* **show** *?thesis* **by** *auto*
qed
qed *simp*
qed

lemma *reducible-on-max-paths-order*: **assumes** *on-max-paths n m1*
on-max-paths n m2
m1 ≠ *m2*
shows *on-max-paths-prev n m1 m2* ∨ *on-max-paths-prev n m2 m1*
proof (*cases valid-node n*)
case *True*
with *max-path-ext* **obtain** *ns* **where** *max-path n ns* **by** *auto*
with *assms on-max-paths-def max-path-split-elem* **obtain** *ns1*
where *is-path n ns1 m1* **by** *metis*
with *assms* **show** *?thesis*
proof (*induction ns1 arbitrary*: *n*)
case *Nil*
with *path-empty-conv on-max-paths-prev-trivial* **show** *?case* **by** *auto*
next
case (*Cons n′ ns1 n*)
show *?case*
proof (*cases n = m2* ∨ *n = m1*)
case *False*
from *Cons is-path-Cons* **obtain** *x*
where *x-gen*: *x* ∈ *succs n is-path x ns1 m1 n = n′* **by** *metis*
with *on-max-paths-step False Cons*
have *max-paths*: *on-max-paths x m1 on-max-paths x m2* **by** *metis+*
with *Cons x-gen* **have** *x-prev*: *on-max-paths-prev x m1 m2* ∨ *on-max-paths-prev x m2 m1* **by** *auto*
from *Cons ntscd-rtrancl-not-on-max-paths False* **have** ¬ *ntscd** n m1* ¬ *ntscd** n m2* **by** *auto*
with *reducible-wod-imp-ntscd-tranclp* **have** ¬ *wod n m1 m2* ¬ *wod n m2 m1* **by** *auto*
with *on-max-path-prev-non-step-wod x-prev Cons x-gen False* **show** *?thesis* **by** *blast*
qed (*auto simp add*: *on-max-paths-prev-trivial*)

qed
qed (*auto simp add*: *on-max-paths-prev-def max-path-valid-node*)

Proof of Theorem 5.1. The assumption of a reducible graph is given by the context, so it is an implicit assumption of this theorem.

theorem *reducible-on-max-paths-first-pos-trans*: **assumes** *on-max-paths-pos-first x y*
on-max-paths-pos-first y z
shows *on-max-paths-pos-first x z*
proof (*cases valid-node x* $\wedge$ *y* $\neq$ *z*)
case *non-trivial*: *True*
from *assms on-max-paths-pos-first-def* **obtain** *k1 k2*
where *k-gen*: *on-max-paths-pos-k-first x k1 y on-max-paths-pos-k-first y k2 z* **by** *auto*
from *on-max-paths-pos-k-implies-on-max-paths on-max-paths-trans k-gen*
have *on-max-paths*: *on-max-paths x y on-max-paths y z on-max-paths x z* **by** *blast+*
show *?thesis*
proof (*cases on-max-paths-prev x y z*)
case *True*
{
fix *ns*
assume *max-path*: *max-path x ns*
with *on-max-paths on-max-paths-def lset-at-pos-first* **obtain** *k*
where *z-pos*: *at-pos-first k ns z* **by** *blast*
from *max-path k-gen on-max-paths-pos-k-first-def* **have** *at-pos-first k1 ns y* **by** *auto*
with *k-gen max-path on-max-paths-pos-first-chain z-pos on-max-paths-prev-at-pos-first True*
have *at-pos-first* (*k1+k2*) *ns z* **by** *fastforce*
}
with *on-max-paths-pos-first-def on-max-paths-pos-k-first-def* **show** *?thesis* **by** *auto*
next
case *False*
with *on-max-paths reducible-on-max-paths-order non-trivial*
have *z-prev-y*: *on-max-paths-prev x z y* **by** *auto*
from *on-max-paths max-path-ext non-trivial* **obtain** *ns* **where** *max-path*: *max-path x ns* **by** *auto*
with *on-max-paths on-max-paths-def lset-at-pos-first lset-at-pos-first* **obtain** *k*
where *z-pos*: *at-pos-first k ns z* **by** *blast*
from *max-path k-gen on-max-paths-pos-k-first-def* **have** *at-pos-first k1 ns y* **by** *auto*
with *k-gen max-path z-pos on-max-paths-prev-at-pos-first z-prev-y non-trivial*
have *less1*: *k* < *k1* **by** *fastforce*
with *on-max-paths-pos-k-first-diff k-gen z-pos max-path*
have *z-y*: *on-max-paths-pos-k-first z* (*k1*−*k*) *y* **by** *auto*
from *on-max-paths-prev-split z-prev-y non-trivial max-path-valid-node*
have *valid-node z* **by** *metis*

{
fix *ns2*
assume *max-path2*: *max-path x ns2*
with *on-max-paths on-max-paths-def lset-at-pos-first lset-at-pos-first* **obtain** *k′*
where *z-pos2*: *at-pos-first k′ ns2 z* **by** *blast*
from *max-path2 k-gen on-max-paths-pos-k-first-def* **have** *at-pos-first k1 ns2 y* **by** *auto*
with *k-gen max-path2 z-pos2 on-max-paths-prev-at-pos-first z-prev-y non-trivial*
have *less2*: $k' < k1$ **by** *fastforce*
with *on-max-paths-pos-k-first-diff k-gen z-pos2 max-path2*
have *on-max-paths-pos-k-first z* $(k1-k')$ *y* **by** *auto*
with *z-y on-max-paths-pos-k-first-k-unique* ‹*valid-node z*› **have** $k1-k' = k1-k$ **by** *auto*
with *less1 less2* **have** $k' = k$ **by** *auto*
with *less1 less2 z-pos2* **have** *at-pos-first k ns2 z* **by** *auto*
}
with *on-max-paths-pos-first-def on-max-paths-pos-k-first-def* **show** *?thesis* **by** *auto*
qed
next
case *False*
with *assms on-max-paths-pos-first-def on-max-paths-pos-k-first-def max-path-valid-node*
show *?thesis* **by** *auto*
qed

end

end

5.3.2 Graphs with unique exit node

The assumption that there is a unique exit node reachable from all other nodes is given by the Postdomination locale.

context *Postdomination*
begin

lemma *unique-exit-on-max-paths-first-pos-k-trans*: **assumes** *on-max-paths-pos-k-first x k1 y*
on-max-paths-pos-k-first y k2 z
shows *on-max-paths-pos-k-first x* $(k1+k2)$ *z*
proof (*cases valid-node x*)
case *x-valid*: *True*
{
fix *ns*
assume *max-path x ns*
with *assms x-valid* **have** *at-pos-first* $(k1+k2)$ *ns z*
proof (*induction k1 arbitrary*: *x ns*)

case *0*
with *on-max-paths-pos-k-first-0* **have** $x = y$ **by** *auto*
with *0 on-max-paths-pos-k-first-def* **show** *?case* **by** *auto*
next
case (*Suc k1 x ns*)
then show *?case*
proof (*cases* $x = z$)
case *True*
with *Suc on-max-paths-pos-k-first-refl on-max-paths-pos-k-first-k-unique*
have $z \neq y$ **by** *blast*
with *Suc True on-max-paths-pos-k-first-end-node Exit-succs* **have** $z \neq$ (*-Exit-*) **by** *auto*
{
fix *ns'*
assume *is-path y ns'* (*-Exit-*)
with *Exit-succs max-path-end* **have** *max-path y* (*llist-of* (*ns'*@[(*-Exit-*)])) **by** *auto*
with *Suc on-max-paths-pos-k-first-def at-pos-first-def in-lset-conv-lnth*
have $z \in$ *lset* (*llist-of* (*ns'*@[(*-Exit-*)])) **by** *metis*
with ‹$z \neq$ (*-Exit-*)› **have** $z \in$ *set ns'* **by** *simp*
}
note *exit-path-z* = *this*
with *path0* **have** $y \neq$ (*-Exit-*) **by** *fastforce*
from *Suc on-max-paths-pos-k-first-def at-pos-first-def*
have *at-pos-first* (*Suc k1*) *ns y* **by** *auto*
with *at-pos-first-def in-lset-conv-lnth* **have** $y \in$ *lset ns* **by** *metis*
with *Suc max-path-split-elem max-path-valid-node* **have** *valid-node y* **by** *metis*
with *Exit-is-path* **obtain** *ns2* **where** *ns2-gen*: *is-path y ns2* (*-Exit-*) **by** *auto*
with *exit-path-z* **have** *ns2* $\neq$ [] **by** *fastforce*
with *path-last ns2-gen* **obtain** *ns3*
where *ns3-gen*: *is-path y* (*y#ns3*) (*-Exit-*) $y \notin$ *set ns3* **by** *metis*
with ‹$z \neq y$› *exit-path-z split-list* **obtain** *ns4 ns5*
where *ns3* = *ns4*@*z#ns5* **by** *fastforce*
with *ns3-gen is-path-split*[*of - y#ns4*]
have *ns3-split*: *is-path z* (*z#ns5*) (*-Exit-*) $y \notin$ *set* (*z#ns5*) **by** *auto*
with *Exit-succs max-path-end*[*of - z#ns5*]
have *max-path z* (*llist-of* (*z#ns5*@[(*-Exit-*)])) **by** *auto*
with *True Suc on-max-paths-pos-k-first-def at-pos-first-def in-lset-conv-lnth*
have $y \in$ *lset* (*llist-of* (*z#ns5*@[(*-Exit-*)])) **by** *metis*
with *ns3-split* ‹$y \neq$ (*-Exit-*)› **show** *?thesis* **by** *auto*
next
case *False*
from *Suc on-max-paths-pos-k-first-end-node* **have** *succs x* $\neq$ {} **by** *blast*
with *Suc max-path-step* **obtain** *x' ns'*
where *step*: *ns* = *LCons x ns' max-path x' ns' x'* $\in$ *succs x* **by** *metis*
with *on-max-paths-pos-k-first-Suc Suc*(*2*) **have** *on-max-paths-pos-k-first x' k1 y* **by** *force*

```
      with step Suc succs-valid have at-pos-first (k1 + k2) ns' z by fastforce
      with at-pos-first-step step False show ?thesis by auto
    qed
  qed
 }
 with on-max-paths-pos-k-first-def show ?thesis by auto
qed (auto simp add: on-max-paths-pos-k-first-def max-path-valid-node)
```

Proof of Theorem 5.2. The assumption of a unique exit node is given by the locale context, so it is an implicit assumption of this theorem.

```
theorem unique-exit-on-max-paths-first-pos-trans: assumes on-max-paths-pos-first x y
                                    on-max-paths-pos-first y z
                                  shows on-max-paths-pos-first x z
 using assms on-max-paths-pos-first-def unique-exit-on-max-paths-first-pos-k-trans
by metis

end
```

5.4 Timing Sensitive Postdominance Frontiers

```
context CFG
begin
```

Definition 5.7, redefinition of *1*−⊑-Postdominance.

```
abbreviation spdom' pdrel n m == pdrel n m ∧ (∃ m'≠m. pdrel n m' ∧ pdrel m' m)
```

Redefinition of the Postdominance Frontier, which uses the redefined *1*−⊑-Postdominance from Definition 5.7.

```
abbreviation pdf' pdrel m == {n. ¬ spdom' pdrel n m ∧ (∃ x∈succs n. pdrel x m)}
```

Proof of Theorem 5.3.

```
theorem tscd-on-max-paths-pos-first-frontier:
 assumes n ≠ m
 shows n ∈ pdf' on-max-paths-pos-first m ⟷ tscd n m
proof
 assume pdf: n ∈ pdf' on-max-paths-pos-first m
  with assms on-max-paths-pos-first-refl have ¬ on-max-paths-pos-first n m by auto
 with pdf tscd-cond-succ show tscd n m by auto
next
 assume tscd n m
 with tscd-def obtain k x1 x2 where succs: x1 ∈ succs n x2 ∈ succs n
   on-max-paths-pos-k-first x1 k m ¬ on-max-paths-pos-k-first x2 k m by auto
 with on-max-paths-pos-first-def on-max-paths-pos-k-first-step assms
      on-max-paths-pos-k-first-k-unique succs-valid have ¬ on-max-paths-pos-first n m  by metis
```

with *succs assms on-max-paths-pos-first-def* **show** $n \in pdf'$ *on-max-paths-pos-first* m **by** *auto*
qed

end

end