Abstract
In this contribution, we show how correctness proofs for intra- and interprocedural slicing can be used to prove that slicing is able to guarantee information flow noninterference. Moreover, we also illustrate how to lift the control flow graphs of the respective frameworks such that they fulfill the additional assumptions needed in the noninterference proofs. A detailed description of the intraprocedural proof and its interplay with the slicing framework can be found in [10].

1 Introduction
Information Flow Control (IFC) encompasses algorithms which determines if a given program leaks secret information to public entities. The major group are so called IFC type systems, where well-typed means that the respective program is secure. Several IFC type systems have been verified in proof assistants, e.g. see [1, 2, 5, 3, 7].

However, type systems have some drawbacks which can lead to false alarms. To overcome this problem, an IFC approach basing on slicing has been developed [4], which can significantly reduce the amount of false alarms. This contribution presents the first machine-checked proof that slicing is able to guarantee IFC noninterference. It bases on previously published machine-checked correctness proofs for slicing [8, 9]. Details for the intraprocedural case can be found in [10].

2 HRB Slicing guarantees IFC Noninterference

theory NonInterferenceInter
imports ../HRB−Slicing/StaticInter/FundamentalProperty
begin
2.1 Assumptions of this Approach

Classical IFC noninterference, a special case of a noninterference definition using partial equivalence relations (per) [6], partitions the variables (i.e. locations) into security levels. Usually, only levels for secret or high, written H, and public or low, written L, variables are used. Basically, a program that is noninterferent has to fulfill one basic property: executing the program in two different initial states that may differ in the values of their H-variables yields two final states that again only differ in the values of their H-variables; thus the values of the H-variables did not influence those of the L-variables.

Every per-based approach makes certain assumptions: (i) all H-variables are defined at the beginning of the program, (ii) all L-variables are observed (or used in our terms) at the end and (iii) every variable is either H or L. This security label is fixed for a variable and can not be altered during a program run. Thus, we have to extend the prerequisites of the slicing framework in [9] accordingly in a new locale:

locale NonInterferenceInterGraph =
  SDG sourcenode targetnode kind valid-edge Entry
  for sourcenode :: 'edge ⇒ 'node and targetnode :: 'edge ⇒ 'node
  and kind :: 'edge ⇒ ('var, 'val, 'ret, ' pname) edge-kind
  and valid-edge :: 'edge ⇒ bool
  and Entry :: ('node ('(Entry-')) ) and get-proc :: 'node ⇒ 'p-name
  and get-return-edges :: 'edge ⇒ 'edge set
  and proc :: ('p-name × 'var list × 'var list) list and Main :: 'p-name
  and Exit::'node ('(Exit-'))
  and Def :: 'node ⇒ 'var set and Use :: 'node ⇒ 'var set
  and ParamDefs :: 'node ⇒ 'var set and ParamUses :: 'node ⇒ 'var set list +
  fixes H :: 'var set
  fixes L :: 'var set
  fixes High :: 'node ('(High-'))
  fixes Low :: 'node ('(Low-'))
  assumes Entry-edge-Exit-or-High:
  [valid-edge a; sourcenode a = (Entry-) ]
  ⇒ targetnode a = (Exit-) ∨ targetnode a = (High-)
  and High-target-Entry-edge:
  ∃ a. valid-edge a ∧ sourcenode a = (Entry-) ∧ targetnode a = (High-) ∧
  kind a = (λs. True),
  and Entry-predecessor-of-High:
  [valid-edge a; targetnode a = (High-) ] ⇒ sourcenode a = (Entry-)
  and Exit-edge-Entry-or-Low: [valid-edge a; targetnode a = (Exit-) ]
  ⇒ sourcenode a = (Exit-) ∨ sourcenode a = (Low-)
  and Low-source-Exit-edge:
  ∃ a. valid-edge a ∧ sourcenode a = (Low-) ∧ targetnode a = (Exit-) ∧
  kind a = (λs. True),
  and Exit-successor-of-Low:
  [valid-edge a; sourcenode a = (Low-) ] ⇒ targetnode a = (Exit-)

2
and DefHigh: Def (-High-) = H
and UseHigh: Use (-High-) = H
and UseLow: Use (-Low-) = L
and HighLowDistinct: H \cap L = \{}
and HighLowUNIV: H \cup L = UNIV

begin

lemma Low-neq-Exit: assumes L \neq \{}
  shows (-Low-) \neq (-Exit-)
⟨proof⟩

lemma valid-node-High [simp]: valid-node (-High-)
⟨proof⟩

lemma valid-node-Low [simp]: valid-node (-Low-)
⟨proof⟩

lemma get-proc-Low:
  get-proc (-Low-) = Main
⟨proof⟩

lemma get-proc-High:
  get-proc (-High-) = Main
⟨proof⟩

lemma Entry-path-High-path:
  assumes (-Entry-) \rightarrow^{*} n \text{ and inner-node } n
  obtains a' as' where as = a'\#as' and (-High-) \rightarrow^{*} n
  and kind a' = (\lambda s. \text{True})\sqrt
⟨proof⟩

lemma Exit-path-Low-path:
  assumes n \rightarrow^{*} (-Exit-) \text{ and inner-node } n
  obtains a' as' where as = as[a']\#a' and n \rightarrow^{*} (-Low-)
  and kind a' = (\lambda s. \text{True})\sqrt
⟨proof⟩

lemma not-Low-High: V \notin L \implies V \in H
⟨proof⟩

lemma not-High-Low: V \notin H \implies V \in L
⟨proof⟩

3
2.2 Low Equivalence

In classical noninterference, an external observer can only see public values, in our case the $L$-variables. If two states agree in the values of all $L$-variables, these states are indistinguishable for him. Low equivalence groups those states in an equivalence class using the relation $\approx_L$:

**definition** lowEquivalence :: ('var → 'val) list ⇒ ('var → 'val) list ⇒ bool

(infixl $\approx_L$ 50)

where $s \approx_L s' \equiv \forall V \in L. \text{hd } s . V = \text{hd } s' . V$

The following lemmas connect low equivalent states with relevant variables as necessary in the correctness proof for slicing.

**lemma** relevant-vars-Entry:

assumes $V \in \text{rv } S \text{ (CFG-node (}\text{-Entry-}) \text{) and (}\text{-High-} ) \notin [\text{HRB-slice } S]_\text{CFG}$

shows $V \in L$

(proof)

**lemma** lowEquivalence-relevant-nodes-Entry:

assumes $s \approx_L s' \text{ and (}\text{-High-} ) \notin [\text{HRB-slice } S]_\text{CFG}$

shows $\forall V \in \text{rv } S \text{ (CFG-node (}\text{-Entry-}) \text{), } \text{hd } s . V = \text{hd } s' . V$

(proof)

2.3 The Correctness Proofs

In the following, we present two correctness proofs that slicing guarantees IFC noninterference. In both theorems, $\text{CFG-node (}\text{-High-} ) \notin \text{HRB-slice } S$, where $\text{CFG-node (}\text{-Low-} ) \in S$, makes sure that no high variable (which are all defined in (\text{-High-})) can influence a low variable (which are all used in (\text{-Low-})).

First, a theorem regarding (\text{-Entry-} $\rightarrow^{\ast}$ (\text{-Exit-}) paths in the control flow graph (CFG), which agree to a complete program execution:

**lemma** slpa-rv-Low-Use-Low:

assumes $\text{CFG-node (}\text{-Low-} ) \in S$

shows $\text{[same-level-path-aux cs as; upd-cs cs as = []; same-level-path-aux cs as']}$

\n
∀ c \in \text{set cs}. \text{valid-edge c; m } \rightarrow^{\ast} \text{ (}\text{-Low-} ); m' \rightarrow^{\ast} \text{ (}\text{-Low-} );$

∀ i < \text{length cs}. \forall V \in \text{rv } S \text{ (CFG-node (source-node (cs!i))}).$

$\text{fst (s!Suc i) } V = \text{fst (s'!Suc i) } V; \forall i < \text{Suc (length cs)}. \text{snd (s!i) = snd (s'?i)};$

∀ V \in \text{rv } S \text{ (CFG-node m). state-val } s . V = \text{state-val } s' . V;$

$\text{preds (slice-kinds S as) s; preds (slice-kinds S as') s';}$

$\text{length s = Suc (length cs); length s' = Suc (length cs)}$

$$\implies \forall V \in \text{Use (}\text{-Low-}). \text{state-val } (\text{transfers(slice-kinds S as) s}) . V = \text{state-val } (\text{transfers(slice-kinds S as') s'}) . V$$

(proof)
lemma \textit{rv-Low-Use-Low}:
\begin{align*}
\text{assumes } & m \xrightarrow{\cdot} \cdot (\text{-Low-}) \text{ and } m \xrightarrow{\cdot'} \cdot (\text{-Low-}) \text{ and get-proc } m = \text{Main} \\
\text{and } & \forall V \in \text{rv} \ (\text{CFG-node } m), \ cf V = cf' V \\
\text{and } & \text{preds (slice-kinds } S \text{ as)} [(cf, \text{undefined})] \\
\text{and } & \text{preds (slice-kinds } S \text{ as'}) [(cf', \text{undefined})] \\
\text{and } & \text{CFG-node (\text{-Low-}) } \in S \\
\text{shows } & V \in \text{Use (\text{-Low-}).} \\
\text{state-val (transfers (slice-kinds } S \text{ as)} [(cf,\text{undefined})]) V = \\
\text{state-val (transfers (slice-kinds } S \text{ as'}) [(cf',\text{undefined})]) V
\end{align*}

\langle \text{proof} \rangle

lemma \textit{nonInterference-path-to-Low}:
\begin{align*}
\text{assumes } & [cf] \approx_L [cf'] \text{ and (\text{-High-}) } \notin [\text{HRB-slice } S]_{\text{CFG}} \\
\text{and } & \text{CFG-node (\text{-Low-}) } \in S \\
\text{and (\text{-Entry-}) } & \xrightarrow{\cdot} \cdot (\text{-Low-}) \text{ and preds (kinds as)} [(cf, \text{undefined})] \\
\text{and (\text{-Entry-}) } & \xrightarrow{\cdot'} \cdot (\text{-Low-}) \text{ and preds (kinds as')} [(cf', \text{undefined})] \\
\text{shows } & \text{map fst (transfers (kinds as)} [(cf,\text{undefined})]) \approx_L \\
& \text{map fst (transfers (kinds as')} [(cf',\text{undefined})])
\end{align*}

\langle \text{proof} \rangle

\textbf{end}

The second theorem assumes that we have a operational semantics, whose evaluations are written \langle c,s \rangle \Rightarrow \langle c',s' \rangle and which conforms to the CFG. The correctness theorem then states that if no high variable influenced a low variable and the initial states were low equivalent, the resulting states are again low equivalent:

\textbf{locale} \textit{NonInterferenceInter} =
\begin{align*}
\text{NonInterferenceInter} & \text{Graph sourcenode targetnode kind valid-edge Entry} \\
& \text{get-proc get-return-edges procs Main Exizt Def Use ParamDefs ParamUses} \\
& H L \text{ High Low} + \\
& \text{SemanticsProperty sourcenode targetnode kind valid-edge Entry get-proc} \\
& \text{get-return-edges procs Main Exizt Def Use ParamDefs ParamUses sem identifies} \\
& \text{for sourcenode :: \text{’edge } \Rightarrow \text{’node and targetnode :: \text{’edge } \Rightarrow \text{’node} } \\
& \text{and kind :: \text{’edge } \Rightarrow \text{’(var, val,’ret,’pname) edge-kind} } \\
& \text{and valid-edge :: \text{’edge } \Rightarrow \text{bool} }
\end{align*}
and Entry :: 'node ('Entry') and get-proc :: 'node ⇒ 'pname
and get-return-edges :: 'edge ⇒ 'edge set
and proc :: ('pname × 'var list × 'var list) list and Main :: 'pname
and Entry :: node ('Entry')
and Def :: 'node ⇒ 'var set
and Use :: 'node ⇒ 'var set
and ParamDefs :: 'node ⇒ 'var list and ParamUses :: 'node ⇒ 'var list set
and sem :: 'com ⇒ (var ⇒ val) list ⇒ 'com ⇒ (var ⇒ val) list ⇒ bool
and identifies :: 'node ⇒ 'com ⇒ bool (¬ - [51,0] 80)
and H :: 'var set and L :: 'var set
and High :: 'node ('High') and Low :: 'node ('Low') +
fixes final :: 'com ⇒ bool
assumes final-edge-Low: [final c; n ≡ c] =⇒∃ a. valid-edge a ∧ sourcenode a = n ∧ targetnode a = (Low-) ∧ kind a = id
begin

The following theorem needs the explicit edge from (High-) to n. An approach using a init predicate for initial statements, being reachable from (High-) via a (λs. True) edge, does not work as the same statement could be identified by several nodes, some initial, some not. E.g., in the program

while (True) Skip;;Skip

two nodes identify this initial statement: the initial node and the node within the loop (because of loop unrolling).

theorem nonInterference:
assumes [cf_1] ≈ L [cf_2] and (High-) /∈ [HRB-slice S] CFG
and CFG-node (Low-) ∈ S
and valid-edge a and sourcenode a = (High-) and targetnode a = n
and kind a = (λs. True) and n ≡ c and final c'
and ⟨c,[cf_1]⟩ ⇒ ⟨c',s_1⟩ and ⟨c,[cf_2]⟩ ⇒ ⟨c',s_2⟩
shows s_1 ≈ L s_2
⟨proof⟩

end

end

3 Framework Graph Lifting for Noninterference

theory LiftingInter
imports NonInterferenceInter begin

In this section, we show how a valid CFG from the slicing framework in [8] can be lifted to fulfil all properties of the NonInterferenceIntraGraph locale. Basically, we redefine the hitherto existing Entry and Exit nodes as new High and Low nodes, and introduce two new nodes NewEntry and NewExit. Then, we have to lift all functions to operate on this new graph.
3.1 Liftings

3.1.1 The datatypes

datatype 'node LDCFG-node = Node 'node
| NewEntry
| NewExit

type-synonym ('edge,'node,'var,'val,'ret,'pname) LDCFG-edge = 'node LDCFG-node × (('var,'val,'ret,'pname) edge-kind) × 'node LDCFG-node

3.1.2 Lifting basic definitions using 'edge and 'node

inductive lift-valid-edge :: ('edge ⇒ bool) ⇒ ('edge ⇒ 'node) ⇒ ('edge ⇒ 'node) ⇒ bool
for valid-edge::'edge ⇒ bool and src::'edge ⇒ 'node and trg::'edge ⇒ 'node
and knd::'edge ⇒ ('var,'val,'ret,'pname) edge-kind and E::'node and X::'node

where lve-edge:
| valid-edge a; src a ≠ E ∨ trg a ≠ X;
| e = (Node (src a),knd a,Node (trg a))
⇒ lift-valid-edge valid-edge src trg knd E X e

| lve-Entry-edge:
| e = (NewEntry,(λs. True)√,Node E)
⇒ lift-valid-edge valid-edge src trg knd E X e

| lve-Exit-edge:
| e = (Node X,(λs. True)√,NewExit)
⇒ lift-valid-edge valid-edge src trg knd E X e

| lve-Entry-Exit-edge:
| e = (NewEntry,(λs. False)√,NewExit)
⇒ lift-valid-edge valid-edge src trg knd E X e

lemma [simp]:¬ lift-valid-edge valid-edge src trg knd E X (Node E,et,Node X)
(proof)

fun lift-get-proc :: ('node ⇒ 'pname) ⇒ 'pname ⇒ 'node LDCFG-node ⇒ 'pname
where lift-get-proc get-proc Main (Node n) = get-proc n
| lift-get-proc get-proc Main NewEntry = Main
| lift-get-proc get-proc Main NewExit = Main
\textbf{inductive-set} lift-get-return-edges :: ('edge \Rightarrow 'edge set) \Rightarrow ('edge \Rightarrow bool) \Rightarrow ('edge \Rightarrow 'node) \Rightarrow ('edge \Rightarrow 'node) \Rightarrow ('edge \Rightarrow ('var,'val,'ret,'pname) edge-kind) \\
\Rightarrow ('edge,'node,'var,'val,'ret,'pname) LDCFG-edge \\
\Rightarrow ('edge,'node,'var,'val,'ret,'pname) LDCFG-edge set \\
for get-return-edges :: 'edge \Rightarrow 'edge set and valid-edge :: 'edge \Rightarrow bool \\
\textbf{and} \ src::'edge \Rightarrow 'node \ and \ try::'edge \Rightarrow 'node \\
\textbf{and} \ knnd::'edge \Rightarrow ('var,'val,'ret,'pname) edge-kind \\
\textbf{and} \ e::('edge,'node,'var,'val,'ret,'pname) LDCFG-edge \\
\textbf{where} lift-get-return-edgesI: \\
\llbracket e = (Node (src a),knnd a,Node (try a)); valid-edge a; a' \in \text{get-return-edges} a; \\
e' = (Node (src a'),knnd a',Node (try a')) \rrbracket \\
\implies e' \in \text{lift-get-return-edges get-return-edges valid-edge src trg knnd e} \\

3.1.3 Lifting the Def and Use sets \\

\textbf{inductive-set} lift-Def-set :: ('node \Rightarrow 'var set) \Rightarrow 'node \Rightarrow 'node \Rightarrow 'var set \Rightarrow 'var set \Rightarrow ('node LDCFG-node \times 'var) set \\
for Def::('node \Rightarrow 'var set) and E::'node and X::'node \\
\textbf{and} \ H::'var set and L::'var set \\
\textbf{where} lift-Def-node: \\
V \in \text{Def} n \implies (Node n,V) \in \text{lift-Def-set} Def E X H L \\
| lift-Def-High: \\
V \in H \implies (Node E,V) \in \text{lift-Def-set} Def E X H L \\

\textbf{abbreviation} lift-Def :: ('node \Rightarrow 'var set) \Rightarrow 'node \Rightarrow 'node \Rightarrow 'var set \Rightarrow 'var set \Rightarrow 'node LDCFG-node \Rightarrow 'var set \\
\textbf{where} lift-Def Def E X H L n \equiv \{ V. (n,V) \in \text{lift-Def-set} Def E X H L \} \\

\textbf{inductive-set} lift-Use-set :: ('node \Rightarrow 'var set) \Rightarrow 'node \Rightarrow 'node \Rightarrow 'var set \Rightarrow 'var set \Rightarrow ('node LDCFG-node \times 'var) set \\
for Use::'node \Rightarrow 'var set and E::'node and X::'node \\
\textbf{and} \ H::'var set and L::'var set \\
\textbf{where} \ lift-Use-node: \\
V \in \text{Use} n \implies (Node n,V) \in \text{lift-Use-set} Use E X H L \\
\ \ | lift-Use-High: \\
V \in H \implies (Node E,V) \in \text{lift-Use-set} Use E X H L \\
\ \ | lift-Use-Low: \\
V \in L \implies (Node X,V) \in \text{lift-Use-set} Use E X H L
abbreviation lift-Use :: ('node ⇒ 'var set) ⇒ 'node ⇒ 'node ⇒ 'var set ⇒ 'var set ⇒ 'node LDCFG-node ⇒ 'var set
where lift-Use Use E X H L n ≡ \{ V. (n, V) ∈ lift-Use-set Use E X H L\}

fun lift-ParamUses :: ('node ⇒ 'var set list) ⇒ 'node LDCFG-node ⇒ 'var set list
where lift-ParamUses ParamUses (Node n) = ParamUses n
| lift-ParamUses ParamUses NewEntry = []
| lift-ParamUses ParamUses NewExit = []

fun lift-ParamDefs :: ('node ⇒ 'var list) ⇒ 'node LDCFG-node ⇒ 'var list
where lift-ParamDefs ParamDefs (Node n) = ParamDefs n
| lift-ParamDefs ParamDefs NewEntry = []
| lift-ParamDefs ParamDefs NewExit = []

3.2 The lifting lemmas

3.2.1 Lifting the CFG locales

abbreviation src :: ('edge,'node,'var,'val,'ret,'pname) LDCFG-edge ⇒ 'node LDCFG-node
where src a ≡ fst a

abbreviation trg :: ('edge,'node,'var,'val,'ret,'pname) LDCFG-edge ⇒ 'node LDCFG-node
where trg a ≡ snd(snd a)

abbreviation knd :: ('edge,'node,'var,'val,'ret,'pname) LDCFG-edge ⇒ ('var,'val,'ret,'pname) edge-kind
where knd a ≡ fst(snd a)

lemma lift-CFG:
assumes wf:CFGExit-wf sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit Def Use ParamDefs ParamUses
and pd:Postdomination sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit
shows CFG src trg knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry
(lift-get-proc get-proc Main)
(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind)
procs Main
⟨proof⟩

lemma lift-CFG-wf:
assumes wf:CFGExit-wf sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit Def Use ParamDefs ParamUses
and pd:Postdomination sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit
shows CFG-wf src trg knd

9
lemma lift-CFGExit:
assumes \( \text{wf:CFGExit-wf} \) sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit Def Use ParamDefs ParamUses
and \( \text{pd:Postdomination} \) sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit
shows \( \text{CFGExit} \) src trg knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry
(lift-get-proc get-proc Main)
(procs Main (lift-Def Def Entry Exit H L) (lift-Use Use Entry Exit H L)
(lift-ParamDefs ParamDefs) (lift-ParamUses ParamUses))
\( \langle \text{proof} \rangle \)

3.2.2 Lifting the SDG

lemma lift-Postdomination:
assumes \( \text{wf:CFGExit-wf} \) sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit Def Use ParamDefs ParamUses
and \( \text{pd:Postdomination} \) sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit
and \( \text{inner:CFGExit.inner-node} \) sourcenode targetnode kind valid-edge Entry Exit nx
shows Postdomination src trg knd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry
(lift-get-proc get-proc Main)
(procs Main NewExit)
\( \langle \text{proof} \rangle \)
lemma lift-SDG:
assumes SDG:SDG sourcenode targetnode kind valid-edge Entry get-proc
get-return-edges procs Main Exit Def Use ParamDefs ParamUses
and inner:CFGExit.inner-node sourcenode targetnode valid-edge Entry Exit nx
shows SDG src try kind
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry
(lift-get-proc get-proc Main)
(lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind)
procs Main NewExit (lift-Def Def Entry Exit H L) (lift-Use Use Entry Exit H L)
(lift-ParamDefs ParamDefs) (lift-ParamUses ParamUses)
⟨proof⟩
end

3.2.3 Low-deterministic security via the lifted graph

lemma Lift-NonInterferenceGraph:
fixes valid-edge and sourcenode and targetnode and kind and Entry and Exit
and get-proc and get-return-edges and procs and Main
and Def and Use and ParamDefs and ParamUses and H and L
defines lve: lve ≡ lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
and lget-proc: lget-proc ≡ lift-get-proc get-proc Main
and lget-return-edges: lget-return-edges ≡
lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind
and lDef: lDef ≡ lift-Def Def Entry Exit H L
and lUse: lUse ≡ lift-Use Use Entry Exit H L
and lParamDefs: lParamDefs ≡ lift-ParamDefs ParamDefs
and lParamUses: lParamUses ≡ lift-ParamUses ParamUses
assumes SDG:SDG sourcenode targetnode kind valid-edge Entry get-proc
get-return-edges procs Main Exit Def Use ParamDefs ParamUses
and inner:CFGExit.inner-node sourcenode targetnode valid-edge Entry Exit nx
and H \cap L = \{\} and H \cup L = UNIV
shows NonInterferenceInterGraph src try kind lve NewEntry lget-proc
lget-return-edges procs Main NewExit lDef lUse lParamDefs lParamUses H L
(Node Entry) (Node Exit)
⟨proof⟩
end

References


