Isabelle Formalization of Hedge-Constrained pre* and DPNs with Locks

Peter Lammich

January 30, 2009

Abstract

Dynamic Pushdown Networks (DPNs) are a model for concurrent programs with recursive procedures and thread creation. We formalize a true-concurrency semantics for DPNs. Executions of this semantics have a tree structure. We show the relation of our semantics to the original interleavings semantics. We then show how to compute predecessor sets of regular sets of configurations w.r.t. tree-regular constraints on the execution.

Acquisition histories have been introduced by Kahlon et al. to model-check parallel pushdown systems with well-nested locks , but without thread creation. We generalize acquisistion histories to be used with DPNs. For this purpose, our tree-based semantics can be naturally applied. Moreover, the generalized acquisition histories enable us to characterize the (tree-based) executions that have a schedule that is valid w.r.t. locks, thus obtaining an algorithm to compute lock-sensitive predecessor sets.

Contents

1 Introduction

Writing parallel programs has become popular in the last decade. However, writing correct parallel programs is notoriously difficult, as there are many possibilities for concurrency related bugs. These are hard to find and hard to reproduce due to the nondeterministic behaviour of the scheduler. Hence there is a strong need for formal methods to verify parallel programs and help find concurrency related bugs. A formal model for parallel programs, that has been studied in the last few years, are dynamic pushdown networks (DPNs) [2], a generalization of pushdown systems, where a rule may have the additional side effect of creating a new process, that is then executed in parallel. Analysis of DPNs is usually done w.r.t. to an interleaving semantics, where an execution is a sequence of rule applications. The interleaving semantics models the execution on a single processor, that performs one step at a time and may switch the currently executing process after every step. However, these interleaved executions do not have nice language theoretic properties, what makes them difficult to reason about. For example, it is undecidable whether there exists an execution with a given regular property. Moreover, executions of the interleaving semantics are not suited to track properties of specific processes, e.g. acquired locks.

In the first part of this formalization, we define a semantics that models an execution as a partially ordered set of steps, rather than a (totally ordered) sequence of steps. This partial ordering only reflects the ordering between steps of the same process and the causality due to process creation, i.e. steps of a created process must be executed after the step that created the process. However, it does not enforce any ordering between steps of processes running in parallel. The interleaved executions can be interpreted as topological sorts of the partial ordering. For executions of DPNs the partial ordering has a tree shape, where thread creation steps have at most two successors and pushdown steps have at most one successor. We formally define these executions as list of trees (called execution hedges).

The key concept of model-checking DPNs is to compute the set of predecessor configurations of a set of configurations. Configurations of DPNs are represented as words over control- and stack- symbols, and for a regular set of configurations, the set of predecessor configurations is regular as well and can be computed efficiently [2]. Predecessor computations can be used for various interesting analysis, like kill/gen analysis on bitvectors [2] and context-bounded model checking [1]. Our approach extends the predecessor computation by additionally allowing tree-regular constraints on the executions. The counterpart for the interleaving semantics, i.e. predecessor computations with (word-)regular constraints on the interleaved executions, is not effective.

In the second part of this formalization, we extend DPNs by adding mutual exclusion via well-nested locks. Locks are a commonly used synchronization primitive to manage shared resources between processes. A process may acquire and release a lock, and, at any time, each lock may be owned by at most one process. If a process wants to acquire a lock already owned by another process, it has to wait until the lock is released. We assume that locks are used in a well-nested fashion, i.e. a process has to release locks in the reversed order of acquisition. Note that in practice locks are commonly used in a well-nested fashion, e.g. the synchronized-blocks of Java guarantee well-nested lock usage. Also note that for non-well-nested locks, even simple reachability problems are undecidable [4]. Parallel pushdown processes with well-nested locks have been analyzed using acquisition histories [4, 3]. We generalize this technique to DPNs. Our generalization is non-trivial, as the original technique is defined for a model where only two parallel processes that both exist at the beginning of the execution need to be considered, while we have a model with unboundedly many processes that may be created at any point of the execution. The generalized acquisition histories allow us to characterize the executions, that are consistent w.r.t. lock usage, by a tree-regular set. Applying the results from the first part of this paper yields an algorithm for computing lock-sensitive predecessor sets with tree-regular constraints.

This formalization accompanies a paper that is currently in preparation. Thus the proofs in this work partially depend on unpublished results that are currently in the process of submission. The following are the most notable results proven in this formalization:

- We present a tree-based view on DPN executions, and an efficient predecessor computation with tree-regular constraints.
- We generalize the concept of acquisition histories to programs with process creation.
- We characterize lock-sensitive executions by tree-regular constraints, thus obtaining an algorithm for computing lock-sensitive predecessor sets.

However, this formalization also has its limits. In particular, it does not include:

- A formalization of operations on automata or tree automata, that would allow to generate executable code.
- A formalization of the saturation algorithm for computing predecessor sets of DPNs [2] — another prerequisite for generating executable code. We have an unpublished formalization of this saturation algorithm, that we will adapt to the latest version of Isabelle and publish in near future.
- Due to the first two limitations, we cannot give a formal proof that shows that our methods are, indeed, executable. However, we prove some lemmas that give strong evidence that our methods are effective and could be implemented in principle.

2 Labeled transition systems

theory *LTS*
imports *Main*
begin

Labeled transition systems (LTS) provide a model of a state transition system with named transitions.

2.1 Definitions

An LTS is modeled as a ternary relation between start configuration, transition label and end configuration

types $('c,'a)$ *LTS* $= ('c \times 'a \times 'c)$ *set*

Transitive reflexive closure

inductive-set
trcl :: $('c,'a)$ *LTS* $\Rightarrow$ $('c,'a$ *list*$)$ *LTS*
for *t*
where
empty[*simp*]: $(c,[],c) \in$ *trcl t*
| *cons*[*simp*]: $[\![$ $(c,a,c') \in t$; $(c',w,c'') \in$ *trcl t* $]\!] \Longrightarrow (c,a\#w,c'') \in$ *trcl t*

2.2 Basic properties of transitive reflexive closure

lemma *trcl-empty-cons*: $(c,[],c')\in$*trcl t* $\Longrightarrow (c=c')$
by (*auto elim*: *trcl.cases*)
lemma *trcl-empty-simp*[*simp*]: $(c,[],c')\in$*trcl t* $= (c=c')$
by (*auto elim*: *trcl.cases intro*: *trcl.intros*)

lemma *trcl-single*[*simp*]: $((c,[a],c') \in$ *trcl t*$) = ((c,a,c') \in t)$
by (*auto elim*: *trcl.cases*)
lemma *trcl-uncons*: $(c,a\#w,c')\in$*trcl t* $\Longrightarrow \exists$ *ch* . $(c,a,ch)\in t \wedge (ch,w,c') \in$ *trcl t*
by (*auto elim*: *trcl.cases*)
lemma *trcl-uncons-cases*: $[\![$
$(c,e\#w,c')\in$*trcl S*;
!!*ch*. $[\![$ $(c,e,ch)\in S$; $(ch,w,c')\in$*trcl S* $]\!] \Longrightarrow P$
$]\!] \Longrightarrow P$
by (*blast dest*: *trcl-uncons*)
lemma *trcl-one-elem*: $(c,e,c')\in t \Longrightarrow (c,[e],c')\in$*trcl t*
by *auto*

lemma *trcl-unconsE*[*cases set*, *case-names split*]: $[\![$
$(c,e\#w,c')\in$*trcl S*;
!!*ch*. $[\![(c,e,ch)\in S$; $(ch,w,c')\in$*trcl S*$]\!] \Longrightarrow P$
$]\!] \Longrightarrow P$
by (*blast dest*: *trcl-uncons*)
lemma *trcl-pair-unconsE*[*cases set*, *case-names split*]: $[\![$
$((s,c),e\#w,(s',c'))\in$*trcl S*;
!!*sh ch*. $[\![((s,c),e,(sh,ch))\in S$; $((sh,ch),w,(s',c'))\in$*trcl S*$]\!] \Longrightarrow P$
$]\!] \Longrightarrow P$
by (*fast dest*: *trcl-uncons*)

lemma *trcl-concat*: !! *c* . $[\![$ $(c,w1,c')\in$*trcl t*; $(c',w2,c'')\in$*trcl t* $]\!]$
$\Longrightarrow (c,w1@w2,c'')\in$*trcl t*
proof (*induct w1*)
case *Nil* **thus** *?case* **by** (*subgoal-tac c=c'*) *auto*
next

case (*Cons a w*) **thus** *?case* **by** (*auto dest*: *trcl-uncons*)
qed

lemma *trcl-unconcat*: !! *c* . (*c*,*w1*@*w2*,*c*′)∈*trcl t*
⟹ ∃ *ch* . (*c*,*w1*,*ch*)∈*trcl t* ∧ (*ch*,*w2*,*c*′)∈*trcl t*
proof (*induct w1*)
case *Nil* **hence** (*c*,[],*c*)∈*trcl t* ∧ (*c*,*w2*,*c*′)∈*trcl t* **by** *auto*
thus *?case* **by** *fast*
next
case (*Cons a w1*) **note** *IHP* = *this*
hence (*c*,*a*#(*w1*@*w2*),*c*′)∈*trcl t* **by** *simp*
with *trcl-uncons* **obtain** *chh* **where** (*c*,*a*,*chh*)∈*t* ∧ (*chh*,*w1*@*w2*,*c*′)∈*trcl t* **by** *fast*
moreover with *IHP* **obtain** *ch* **where** (*chh*,*w1*,*ch*)∈*trcl t* ∧ (*ch*,*w2*,*c*′)∈*trcl t* **by** *fast*
ultimately have (*c*,*a*#*w1*,*ch*)∈*trcl t* ∧ (*ch*,*w2*,*c*′)∈*trcl t* **by** *auto*
thus *?case* **by** *fast*
qed

2.2.1 Appending of elements to paths

lemma *trcl-rev-cons*: ⟦ (*c*,*w*,*ch*)∈*trcl T*; (*ch*,*e*,*c*′)∈*T* ⟧ ⟹ (*c*,*w*@[*e*],*c*′)∈*trcl T*
by (*auto dest*: *trcl-concat iff add*: *trcl-single*)
lemma *trcl-rev-uncons*: (*c*,*w*@[*e*],*c*′)∈*trcl T*
⟹ ∃ *ch*. (*c*,*w*,*ch*)∈*trcl T* ∧ (*ch*,*e*,*c*′)∈*T*
by (*force dest*: *trcl-unconcat*)
lemma *trcl-rev-uncons-cases*: ⟦
(*c*,*w*@[*e*],*c*′)∈*trcl T*;
!!*ch*. ⟦(*c*,*w*,*ch*)∈*trcl T*; (*ch*,*e*,*c*′)∈*T*⟧ ⟹ *P*
⟧ ⟹ *P*
by (*blast dest*: *trcl-rev-uncons*)

lemma *trcl-rev-induct*[*induct set*, *consumes 1*, *case-names empty snoc*]: !! *c*′. ⟦
(*c*,*w*,*c*′)∈*trcl S*;
!!*c*. *P c* [] *c*;
!!*c w c*′ *e c*′′. ⟦ (*c*,*w*,*c*′)∈*trcl S*; (*c*′,*e*,*c*′′)∈*S*; *P c w c*′ ⟧ ⟹ *P c* (*w*@[*e*]) *c*′′
⟧ ⟹ *P c w c*′
by (*induct w rule*: *rev-induct*) (*auto dest*: *trcl-rev-uncons*)
lemma *trcl-rev-cases*: !!*c c*′. ⟦
(*c*,*w*,*c*′)∈*trcl S*;
⟦*w*=[]; *c*=*c*′⟧⟹*P*;
!!*ch e wh*. ⟦ *w*=*wh*@[*e*]; (*c*,*wh*,*ch*)∈*trcl S*; (*ch*,*e*,*c*′)∈*S* ⟧⟹*P*
⟧ ⟹ *P*
by (*induct w rule*: *rev-induct*) (*simp*, *blast dest*: *trcl-rev-uncons*)

lemma *trcl-cons2*: ⟦ (*c*,*e*,*ch*)∈*T*; (*ch*,*f*,*c*′)∈*T* ⟧ ⟹ (*c*,[*e*,*f*],*c*′)∈*trcl T*
by *auto*

2.2.2 Transitivity reasoning setup

declare *trcl-cons2*[*trans*] — It's important that this is declared before *trcl-concat*, because we want *trcl-concat* to be tried first by the transitivity reasoner
declare *cons*[*trans*]
declare *trcl-concat*[*trans*]
declare *trcl-rev-cons*[*trans*]

2.2.3 Monotonicity

lemma *trcl-mono*: !!A B. $A \subseteq B \Longrightarrow$ *trcl* $A \subseteq$ *trcl* B
  apply (*clarsimp*)
  apply (*erule trcl.induct*)
  apply *auto*
done

lemma *trcl-inter-mono*: $x \in$*trcl* $(S \cap R) \Longrightarrow x \in$*trcl* S $x \in$*trcl* $(S \cap R) \Longrightarrow x \in$*trcl* R
proof −
  assume $x \in$*trcl* $(S \cap R)$
  with *trcl-mono*[*of* $S \cap R$ S] **show** $x \in$*trcl* S **by** *auto*
next
  assume $x \in$*trcl* $(S \cap R)$
  with *trcl-mono*[*of* $S \cap R$ R] **show** $x \in$*trcl* R **by** *auto*
qed

2.2.4 Special lemmas for reasoning about states that are pairs

lemmas *trcl-pair-induct* = *trcl.induct*[*of* (*xc1*,*xc2*) *xb* (*xa1*,*xa2*), *consumes 1*, *split-format* (*complete*), *case-names empty cons*]
lemmas *trcl-rev-pair-induct* = *trcl-rev-induct*[*of* (*xc1*,*xc2*) *xb* (*xa1*,*xa2*), *consumes 1*, *split-format* (*complete*), *case-names empty snoc*]

2.2.5 Invariants

lemma *trcl-prop-trans*[*cases set*, *consumes 1*, *case-names empty steps*]: $\llbracket$
  $(c,w,c') \in$*trcl* S;
  $\llbracket c=c'$; $w=[] \rrbracket \Longrightarrow P$;
  $\llbracket c \in$*Domain* S; $c' \in$*Range* (*Range* S)$\rrbracket \Longrightarrow P$
 $\rrbracket \Longrightarrow P$
  apply (*erule-tac trcl-rev-cases*)
  apply *auto*
  apply (*erule trcl.cases*)
  apply *auto*
  done

end

3 Dynamic Pushdown Networks

theory *DPN*
imports *Main common/LTS*
begin declare *predicate2I*[*HOL.rule del*, *Pure.rule del*]

3.1 Model Definition

A *Dynamic Pushdown Network* (DPN) [2] is a system of pushdown rules over states from $'Q$ and stack symbols from $'\Gamma$, where each pushdown rule may spawn additional processes. Rules are labeled by elements of type $'L$

datatype $('P,'\Gamma,'L)$ *pushdown-rule* =
  NOSPAWN $'P$ $'\Gamma$ $'L$ $'P$ $'\Gamma$ *list* (-,- $\hookrightarrow_{-}$ -,- *51*) |
  SPAWN $'P$ $'\Gamma$ $'L$ $'P$ $'\Gamma$ *list* $'P$ $'\Gamma$ *list* (-,- $\hookrightarrow_{-}$ -,- $\sharp$ -,- *51*)

notation *NOSPAWN* (-,- $\hookrightarrow_{-}$ -,- *51*)
notation *SPAWN* (-,- $\hookrightarrow_{-}$ -,- $\sharp$ -,- *51*)

types $('Q,'\Gamma,'L)$ *dpn* = $('Q,'\Gamma,'L)$ *pushdown-rule set*

We fix the finiteness assumption of the set of rules in a locale. Note that we do not assume the base types of states, stack symbols, or labels to be finite. However, the finiteness assumption of the set of rules implies that the sets of *used* control states, stack symbols, and labels are finite.

locale *DPN* =
  fixes Δ :: $('Q,'\Gamma,'L)$ *dpn*
  assumes *ruleset-finite*[*simp*, *intro*!]: *finite* Δ

end

4 Semantics

theory *Semantics*
imports *DPN RegSet-add*
begin

In this theory, we define an interleaving and a tree-based semantics of DPNs. We show the equivalence of the two semantics.

4.1 Interleaving Semantics

The interleaving semantics models the execution of a DPN on a single processor, that makes one step at a time, and may switch the currently executed process after each step. This is the original semantics of DPNs [2].

The interleaving semantics is formalized by means of a labeled transition system. A single process is modeled as a pair of its control state and its stack.

A configuration of the DPN is modeled as a list of processes. Note that we use lists of processes here, rather than multisets, to enable representation of configurations as regular sets, as required by the algorithms of [2].

types
$('Q,'\Gamma)$ *pconf* $= 'Q \times '\Gamma$ *list*
$('Q,'\Gamma)$ *conf* $= ('Q,'\Gamma)$ *pconf list*

The (single-) step relation *dpntr* of the interleavings semantics is defined as the least solution of the following constraints:

inductive-set *dpntr* :: $('Q,'\Gamma,'L)$ *dpn* $\Rightarrow$ $(('Q,'\Gamma)$ *conf* $\times$ $'L \times ('Q,'\Gamma)$ *conf*$)$ *set*
for Δ **where**
— A non-spawning step modifies a single pushdown process according to a non-spawning rule in the DPN:
dpntr-no-spawn:
$(p,\gamma \hookrightarrow_l p',w) \in \Delta \Longrightarrow$
$(c1@(p,\gamma\#r)\#c2,l,c1@(p',w@r)\#c2) \in$ *dpntr* Δ |
— A spawning step modifies a pushdown process according to a spawning rule in the DPN and adds the spawned process immediately before the spawning process:
dpntr-spawn:
$(p,\gamma \hookrightarrow_l ps,ws \sharp p',w) \in \Delta \Longrightarrow$
$(c1@(p,\gamma\#r)\#c2,l,c1@(ps,ws)\#(p',w@r)\#c2) \in$ *dpntr* Δ

We denote the reflexive, transitive closure of the single-step relation by *dpntrc*:

abbreviation *dpntrc M* == *trcl* (*dpntr M*)

4.2 Tree Semantics

Now we regard a true concurrency semantics, where an execution does not contain the interleaving between independent steps. When starting at a single process, we model such an execution as a tree, where each node corresponds to an applied step. A node corresponding to a non-spawning step has one successor, a node corresponding to a spawning step has two successors. We annotate the leafs of the tree by the configuration of the reached process.

When starting at a configuration consisting of (a list of) multiple processes, we model the execution as a list of multiple execution trees, one for each process.

datatype $('Q,'\Gamma,'L)$ *ex-tree* =
NLEAF $('Q,'\Gamma)$ *pconf* |
NNOSPAWN $'L$ $('Q,'\Gamma,'L)$ *ex-tree* |
NSPAWN $'L$ $('Q,'\Gamma,'L)$ *ex-tree* $('Q,'\Gamma,'L)$ *ex-tree*

types $('Q,'\Gamma,'L)$ *ex-hedge* = $('Q,'\Gamma,'L)$ *ex-tree list*

inductive *tsem*

:: $('Q,'\Gamma,'L)$ *dpn* ⇒ $('Q,'\Gamma)$ *pconf* ⇒ $('Q,'\Gamma,'L)$ *ex-tree* ⇒ $('Q,'\Gamma)$ *conf* ⇒ *bool*
for Δ **where**
tsem-leaf[*simp*, *intro*!]:
tsem Δ *pw* (*NLEAF pw*) [*pw*] |
tsem-nospawn:
$[\![(p,\gamma \hookrightarrow_l p',w)\in\Delta;$ *tsem* $\Delta\ (p',w@r)\ t\ c']\!] \Longrightarrow$
tsem $\Delta\ (p,\gamma\#r)$ (*NNOSPAWN l t*) c' |
tsem-spawn:
$[\![(p,\gamma \hookrightarrow_l ps,ws \sharp p',w)\in\Delta;$ *tsem* Δ (*ps*,*ws*) *ts cs*; *tsem* $\Delta\ (p',w@r)\ t\ c']\!] \Longrightarrow$
tsem $\Delta\ (p,\gamma\#r)$ (*NSPAWN l ts t*) $(cs@c')$

inductive *hsem*
:: $('Q,'\Gamma,'L)$ *dpn* ⇒ $('Q,'\Gamma)$ *conf* ⇒ $('Q,'\Gamma,'L)$ *ex-hedge* ⇒ $('Q,'\Gamma)$ *conf* ⇒ *bool*
for Δ **where**
hsem-empty[*simp*, *intro*!]: *hsem* Δ [] [] [] |
hsem-cons: $[\![$*tsem* $\Delta\ \pi\ t\ cf'$; *hsem* $\Delta\ c\ h\ c']\!] \Longrightarrow$ *hsem* $\Delta\ (\pi\#c)\ (t\#h)\ (cf'@c')$

In the following we show some basic facts about the *tsem*- and *hsem*-relations.

lemma *hsem-empty-h*[*simp*]:
hsem $\Delta\ c\ []\ c' \longleftrightarrow c=[] \wedge c'=[]$
by (*auto elim*: *hsem.cases intro*: *hsem.intros*)

lemma *hsem-length*: *hsem* $\Delta\ c\ h\ c' \Longrightarrow$ *length c* = *length h*
by (*induct rule*: *hsem.induct*) *auto*

The hedges and configurations of the hedge semantics can be concatenated.

lemmas *hsem-cons-single* = *hsem-cons*[**where** $cf'=[\pi']$, *simplified*, *standard*]

lemma *hsem-conc*: $[\![$*hsem* $\Delta\ c1\ h1\ c1'$; *hsem* $\Delta\ c2\ h2\ c2']\!] \Longrightarrow$
hsem $\Delta\ (c1@c2)\ (h1@h2)\ (c1'@c2')$
by (*induct c1 h1 c1' rule*: *hsem.induct*) (*auto intro*: *hsem-cons*)

lemmas *hsem-conc-lel* = *hsem-conc*[*OF* - *hsem-cons*]

lemmas *hsem-conc-leel* = *hsem-conc*[*OF* - *hsem-cons*[*OF* - *hsem-cons*]]

lemma *tsem-not-empty*[*simp*]: ¬ *tsem* $\Delta\ \pi\ t$ []
by (*induct t arbitrary*: π) (*auto elim*: *tsem.cases*)
lemma *hsem-empty-simps1*[*simp*]:
hsem $\Delta\ []\ h\ c' \longleftrightarrow (h=[] \wedge c'=[])$
hsem $\Delta\ c\ h\ [] \longleftrightarrow (c=[] \wedge h=[])$
by (*auto elim*: *hsem.cases*)

lemma *hsem-id*[*simp*, *intro*!]: *hsem* Δ *c* (*map NLEAF c*) *c*
by (*induct c*) (*auto intro*: *hsem-cons-single*)
lemmas *hsem-id'*[*simp*, *intro*!] = *hsem-id*[*of* - $\pi\#c$, *simplified*, *standard*]

Given a partition of the starting configuration, we can construct a corresponding partition of the hedge and the final configuration.

lemma *hsem-split′*:
  ⟦*hsem* Δ (*c1*@*c2*) *h c′*⟧ ⟹ ∃ *h1 h2 c1′ c2′*.
    h=*h1*@*h2* ∧ *c′*=*c1′*@*c2′* ∧
    hsem Δ *c1 h1 c1′* ∧ *hsem* Δ *c2 h2 c2′*
proof (*induct c1 arbitrary*: *c2 h c′*)
  case *Nil* **hence** *h*=[]@*h* *c′*=[]@*c′* *hsem* Δ [] [] [] *hsem* Δ *c2 h c′*
    by (*auto intro*: *hsem.intros*)
  with *Nil* **show** *?case* **by** *blast*
next
  case (*Cons p c1*)
  from *Cons.prems*[*simplified*] **show** *?case*
  proof (*cases rule*: *hsem.cases*)
    case *hsem-empty* **hence** *False* **by** *simp* **thus** *?thesis* **..**
  next
    case (*hsem-cons px t ct′ c hx cx′*)
    hence *CC*: *h*=*t*#*hx* *tsem* Δ *p t ct′* *hsem* Δ (*c1*@*c2*) *hx cx′* *c′*=*ct′*@*cx′*
      by *simp-all*
    from *Cons.hyps*[*OF CC*(*3*)] **obtain** *h1 h2 c1′ c2′* **where**
      IHAPP: *hx*=*h1*@*h2* *cx′*=*c1′*@*c2′* *hsem* Δ *c1 h1 c1′* *hsem* Δ *c2 h2 c2′*

      by *blast*
    have *h*=(*t*#*h1*)@*h2* *c′*=(*ct′*@*c1′*)@*c2′* **using** *CC IHAPP*
      by *simp-all*
    with *hsem.intros*(*2*)[*OF CC*(*2*) *IHAPP*(*3*)] *IHAPP*(*4*) **show** *?thesis* **by** *blast*
  qed
qed

lemma *hsem-split*[*consumes 1*]: ⟦*hsem* Δ (*c1*@*c2*) *h c′*;
  !!*h1 h2 c1′ c2′*.
    ⟦*h*=*h1*@*h2*; *c′*=*c1′*@*c2′*; *hsem* Δ *c1 h1 c1′*; *hsem* Δ *c2 h2 c2′*⟧ ⟹ *P*
  ⟧ ⟹ *P*
  by (*blast dest*: *hsem-split′*)

lemma *hsem-single*:
  ⟦*hsem* Δ [π] *h c′*; !!*t*. ⟦ *h*=[*t*]; *tsem* Δ π *t c′* ⟧ ⟹ *P* ⟧ ⟹ *P*
  by (*auto intro*: *hsem.intros elim*!: *hsem.cases*)

lemma *hsem-split-single*[*consumes 1*]: ⟦*hsem* Δ (π#*c2*) *h c′*;
  !!*t1 h2 c1′ c2′*.
    ⟦*h*=*t1*#*h2*; *c′*=*c1′*@*c2′*; *tsem* Δ π *t1 c1′*; *hsem* Δ *c2 h2 c2′*⟧ ⟹ *P*
  ⟧ ⟹ *P*
  by (*fastsimp elim*: *hsem-split*[**where** *?c1.0*=[π], *simplified*] *hsem-single*)

lemma *hsem-lel*: ⟦ *hsem* Δ (*c1*@π#*c2*) *h c′*;
    !!*h1 t h2 c1′ ct′ c2′*. ⟦
      h=*h1*@*t*#*h2*; *c′*=*c1′*@*ct′*@*c2′*;
      hsem Δ *c1 h1 c1′*; *tsem* Δ π *t ct′*; *hsem* Δ *c2 h2 c2′*
    ⟧ ⟹ *P*

$]\!] \Longrightarrow P$
by (*fastsimp elim*: *hsem-split hsem-split-single*)

Given a partition of the hedge, we can construct a corresponding partition of the initial and final configuration.

lemma *hsem-split-h'*: $[\![$*hsem* Δ *c* (*h1*@*h2*) *c'*$]\!] \Longrightarrow$
$\exists$ *c1 c2 c1' c2'*. *c*=*c1*@*c2* $\wedge$ *c'*=*c1'*@*c2'* $\wedge$
hsem Δ *c1 h1 c1'* $\wedge$ *hsem* Δ *c2 h2 c2'*
proof (*induct h1 arbitrary*: *h2 c c'*)
case *Nil* **hence** *c*=[]@*c* *c'*=[]@*c'* *hsem* Δ [] [] [] *hsem* Δ *c h2 c'*
by (*auto intro*: *hsem.intros*)
with *Nil* **show** *?case* **by** *blast*
next
case (*Cons t h1*)
from *Cons.prems*[*simplified*] **show** *?case* **proof** (*cases rule*: *hsem.cases*)
case *hsem-empty* **hence** *False* **by** *simp* **thus** *?thesis* **..**
next
case (*hsem-cons p tx ct' cx hx cx'*)
hence *CC*: *c*=*p*#*cx* *tsem* Δ *p t ct'* *hsem* Δ *cx* (*h1*@*h2*) *cx'* *c'*=*ct'*@*cx'*

by *simp-all*
from *Cons.hyps*[*OF CC*(*3*)] **obtain** *c1 c2 c1' c2'* **where**
IHAPP: *cx*=*c1*@*c2* *cx'*=*c1'*@*c2'* *hsem* Δ *c1 h1 c1'* *hsem* Δ *c2 h2 c2'*
by *blast*
have *c*=(*p*#*c1*)@*c2* *c'*=(*ct'*@*c1'*)@*c2'* **using** *CC IHAPP* **by** *simp-all*
with *hsem.intros*(*2*)[*OF CC*(*2*), *OF IHAPP*(*3*)] *IHAPP*(*4*) **show** *?thesis*
by *blast*
qed
qed

lemma *hsem-split-h*:
$[\![$ *hsem* Δ *c* (*h1*@*h2*) *c'*;
!!*c1 c2 c1' c2'*.
$[\![$*c*=*c1*@*c2*; *c'*=*c1'*@*c2'*; *hsem* Δ *c1 h1 c1'*; *hsem* Δ *c2 h2 c2'*$]\!] \Longrightarrow P$
$]\!] \Longrightarrow P$
by (*blast dest*: *hsem-split-h'*)

lemma *hsem-single-h*:
$[\![$*hsem* Δ *c* [*t*] *c'*; !!*p*. $[\![$ *c*=[*p*]; *tsem* Δ *p t c'* $]\!] \Longrightarrow P$ $]\!] \Longrightarrow P$
by (*force intro*: *hsem.intros elim*!: *hsem.cases*)

lemmas *hsem-split-h-single* = *hsem-split-h*[**where** *?h1.0*=[*t*], *simplified*, *standard*]

lemma *hsem-lel-h*: $[\![$ *hsem* Δ *c* (*h1*@*t*#*h2*) *c'*;
!!*c1 p c2 c1' ct' c2'*. $[\![$
c=*c1*@*p*#*c2*; *c'*=*c1'*@*ct'*@*c2'*;
hsem Δ *c1 h1 c1'*; *tsem* Δ *p t ct'*; *hsem* Δ *c2 h2 c2'*
$]\!] \Longrightarrow P$

$\rrbracket \Longrightarrow P$
by (*fastsimp elim*!: *hsem-split-h hsem-split-h-single hsem-single-h*)

4.2.1 Scheduler

The scheduler maps execution hedges to compatible label sequences. This is done by eating up the given hedge from the roots to the leafs, until all non-leaf nodes have been consumed. From an ordering point of view, the hedge represents a partial ordering on the steps, and the scheduler maps this ordering to the set of all its topological sorts.

An execution hedge is called *final* if it solely consists of leaf nodes.

inductive *final-t* **where**
[*simp*, *intro*!]: *final-t* (*NLEAF pw*)

lemma [*simp*, *intro*!]:
¬*final-t* (*NNOSPAWN l t*)
¬*final-t* (*NSPAWN l ts t*)
by (*auto elim*: *final-t.cases*)

abbreviation *final* == *list-all final-t*

Final execution hedges contain no steps, hence they do not change the configuration.

lemma *final-tsem-nostep*: ⟦*final-t t*; *tsem* Δ *pw t c'*⟧ ⟹ *c'*=[*pw*]
by (*cases t*) (*auto elim*: *tsem.cases*)

lemma *final-hsem-nostep*: ⟦*final h*; *hsem* Δ *c h c'*⟧ ⟹ *c'*=*c*
apply (*rotate-tac*)
apply (*induct rule*: *hsem.induct*)
apply (*auto intro*: *final-tsem-nostep*)
done

As described above, the scheduler eats up the execution hedge from the roots to the leafs, until there are no inner nodes remaining, i.e. the hedge is final.

inductive *sched* :: (*'Q*,*'T*,*'L*) *ex-hedge* ⇒ *'L list* ⇒ *bool* **where**
sched-final: *final h* ⟹ *sched h* [] |
sched-nospawn:
sched (*h1*@*t*#*h2*) *w* ⟹ *sched* (*h1*@(*NNOSPAWN l t*)#*h2*) (*l*#*w*) |
sched-spawn:
sched (*h1*@*ts*#*t*#*h2*) *w* ⟹ *sched* (*h1*@(*NSPAWN l ts t*)#*h2*) (*l*#*w*)

inductive-set *sched-rel* :: ((*'Q*,*'T*,*'L*) *ex-hedge*, *'L*) *LTS* **where**
sched-rel-nospawn: ((*h1*@(*NNOSPAWN l t*)#*h2*),*l*,*h1*@*t*#*h2*)∈*sched-rel* |
sched-rel-spawn: ((*h1*@(*NSPAWN l ts t*)#*h2*),*l*,(*h1*@*ts*#*t*#*h2*))∈*sched-rel*

definition *sched' h ll* == (∃ *h'*. (*h*,*ll*,*h'*)∈*trcl sched-rel* ∧ *final h'*)

lemma *sched-alt1*: *sched h ll* ⟹ *sched′ h ll*
  by (*unfold sched′-def*, *induct rule*: *sched.induct*)
    (*auto intro*: *trcl.intros sched-rel.intros*)

lemma *sched-rel-alt2*: ⟦(*h*,*ll*,*h′*)∈*trcl sched-rel*; *final h′*⟧ ⟹ *sched h ll*
  by (*induct rule*: *trcl.induct*) (*auto intro*: *sched.intros elim*: *sched-rel.cases*)

lemma *sched-alt*: *sched′ h ll* ⟷ *sched h ll*
  by (*unfold sched′-def*, *auto intro*: *sched-alt1*[*unfolded sched′-def*] *sched-rel-alt2*)

We now show some basic facts about the scheduler.

lemma *sched-empty-seq*[*simp*]: *sched h* [] ⟷ *final h*
  by (*auto intro*: *sched-final elim*: *sched.cases*)

lemma *sched-empty-hedge*[*simp*]: *sched* [] *ll* ⟷ *ll*=[]
  by (*auto intro*: *sched-final elim*: *sched.cases*)

lemma *sched-empty-empty*[*simp*, *intro*!]: *sched* [] [] **by** (*auto intro*: *sched-final*)

lemma *sched-final-simp*[*simp*]: *final h* ⟹ *sched h c* ⟷ *c*=[]
  by (*auto elim*: *sched.cases*)

In the following few lemmas we derive an induction scheme that reasons about hedges in the way they are consumed by the scheduler

fun *sched-ind-size* **where**
  sched-ind-size (*NLEAF* π) = *0* |
  sched-ind-size (*NNOSPAWN l t*) = *Suc* (*sched-ind-size t*) |
  sched-ind-size (*NSPAWN l ts t*) = *Suc* (*sched-ind-size ts* + *sched-ind-size t*)

abbreviation *sched-ind-sizeh h* == *listsum* (*map sched-ind-size h*)

lemma *sched-ind-h-cases*[*consumes 1*, *case-names NOSPAWN SPAWN*]:
  ⟦ *sched-ind-sizeh h* > *0*;
    !!*h1 l t h2*. *h*=*h1*@(*NNOSPAWN l t*)#*h2* ⟹ *P*;
    !!*h1 ts t h2 l*. *h* = *h1*@(*NSPAWN l ts t*)#*h2* ⟹ *P*
  ⟧ ⟹ *P*
proof (*induct h*)
  case *Nil* **thus** *?case* **by** *auto*
next
  case (*Cons t h*)
  show *?case* **proof** (*cases t*)
    case (*NLEAF* π)
    with *Cons.prems*(*1*) **have** *I*: *0* < *sched-ind-sizeh h* **by** *simp*
    show *?thesis* **proof** (*rule Cons.hyps*[*OF I*])
      fix *h1 l tt h2*
      assume *h*=*h1* @ *NNOSPAWN l tt* # *h2*
      hence *t*#*h* = (*t*#*h1*) @ *NNOSPAWN l tt* # *h2* **by** *simp*
      with *Cons.prems*(*2*) **show** *?thesis* **by** *blast*

```
    next
      fix h1 ts tt h2 l
      assume h = h1 @ NSPAWN l ts tt # h2
      hence t#h = (t#h1) @ NSPAWN l ts tt # h2 by simp
      with Cons.prems(3) show ?thesis by blast
    qed
  next
    case (NNOSPAWN L tt)
    with Cons.prems(2)[of [], simplified] show ?thesis by auto
  next
    case (NSPAWN L ts tt)
    with Cons.prems(3)[of [], simplified] show ?thesis by auto
  qed
qed

lemma sched-ind-helper:
  ⟦ !!h. final h ⟹ P h;
    !!h1 t h2 l. P (h1@t#h2) ⟹ P (h1@(NNOSPAWN l t)#h2);
    !!h1 ts t h2 l. P (h1@ts#t#h2) ⟹ P (h1@(NSPAWN l ts t)#h2);
    sched-ind-sizeh h = k
  ⟧ ⟹ P h
proof (induct k arbitrary: h)
  case 0 note C=this from C(4) have final h
    apply (induct h)
    apply simp
    apply (case-tac a)
    apply auto
    done
  with C(1) show ?case by blast
next
  case (Suc k) hence S: sched-ind-sizeh h > 0 by simp
  thus ?case proof (cases rule: sched-ind-h-cases)
    case (NOSPAWN h1 l t h2)
    with Suc.prems(4) have I: sched-ind-sizeh (h1@t#h2) = k by simp
    with Suc.prems(1,2,3) NOSPAWN show ?thesis
      by (drule-tac Suc.hyps) blast+
  next
    case (SPAWN h1 ts t h2 l)
    with Suc.prems(4) have I: sched-ind-sizeh (h1@ts#t#h2) = k by simp
    with Suc.prems(1,2,3) SPAWN show ?thesis
      by (drule-tac Suc.hyps) blast+
  qed
qed

lemma sched-ind[case-names FINAL NOSPAWN SPAWN]:
  ⟦ !!h. final h ⟹ P h;
    !!h1 t h2 l. P (h1@t#h2) ⟹ P (h1@(NNOSPAWN l t)#h2);
    !!h1 ts t h2 l. P (h1@ts#t#h2) ⟹ P (h1@(NSPAWN l ts t)#h2)
  ⟧ ⟹ P h
```

using *sched-ind-helper* **by** *blast*

Every tree/hedge has at least one schedule. From an ordering point of view, this is because hedge-structures are acyclic, and thus have always at least one topological sort. However, using the inductive definition of the scheduler, the proof of this lemma is by straightforward induction.

lemma *exists-schedule*: $[\![!!ll.\ sched\ h\ ll \Longrightarrow P]\!] \Longrightarrow P$
by (*induct h rule*: *sched-ind*) (*auto intro*: *sched.intros*)

Next, we want to show that the true concurrency semantics corresponds to the interleaving semantics. For this purpose, we show that we have an execution with labeling sequence *ll* in the interleaving semantics if and only if there is an execution *h* in the true concurrency semantics that has *ll* in its set of schedules.

The next two lemmas show the two directions of this claim.

lemma *sched-correct1*: $(c,ll,c')\in dpntrc\ \Delta \Longrightarrow \exists h.\ hsem\ \Delta\ c\ h\ c' \wedge sched\ h\ ll$
proof (*induct rule*: *trcl.induct*)
case (*empty c*) **thus** *?case* **by** (*induct c*) (*auto intro*: *hsem-cons-single*)
next
case (*cons c l ch ll c'*)
from *cons.hyps*(*3*) **obtain** *h* **where** *IHAPP*: $hsem\ \Delta\ ch\ h\ c'$ $sched\ h\ ll$ **by** *blast*
from *cons.hyps*(*1*) **show** *?case*
proof (*cases*)
case (*dpntr-no-spawn p* γ *la p' w c1 r c2*)
hence
C-simp[*simp*]: $c = c1\ @\ (p,\ \gamma\ \#\ r)\ \#\ c2$ $ch = c1\ @\ (p',\ w\ @\ r)\ \#\ c2$ **and**

C: $(p,\gamma \hookrightarrow_l p',w) \in \Delta$
by *auto*
from *hsem-lel*[*OF IHAPP*(*1*)[*simplified*]] **obtain** *h1 t h2 c1' ct' c2'* **where**
[*simp*]: $h = h1\ @\ t\ \#\ h2$ $c' = c1'\ @\ ct'\ @\ c2'$ **and**
HSPLIT: $hsem\ \Delta\ c1\ h1\ c1'$ $tsem\ \Delta\ (p',\ w\ @\ r)\ t\ ct'$ $hsem\ \Delta\ c2\ h2\ c2'$
.
from *tsem-nospawn*[*OF C HSPLIT*(*2*)] **have**
ST: $tsem\ \Delta\ (p,\gamma\#r)\ (NNOSPAWN\ l\ t)\ ct'$ **.**
from *hsem-conc-lel*[*OF HSPLIT*(*1*) *ST HSPLIT*(*3*)] **have**
$hsem\ \Delta\ c\ (h1\ @\ NNOSPAWN\ l\ t\ \#\ h2)\ c'$
by *simp*
moreover from *sched-nospawn*[*OF IHAPP*(*2*)[*simplified*]] **have**
$sched\ (h1\ @\ NNOSPAWN\ l\ t\ \#\ h2)\ (l\#ll)$ **.**
ultimately show *?thesis* **by** *blast*
next
case (*dpntr-spawn p* γ *la ps ws p' w c1 r c2*)
hence
[*simp*]: $c = c1\ @\ (p,\ \gamma\ \#\ r)\ \#\ c2$
$ch = c1\ @\ (ps,\ ws)\ \#\ (p',\ w\ @\ r)\ \#\ c2$ **and**
C: $(p,\gamma \hookrightarrow_l ps,ws \sharp p',w) \in \Delta$

by *auto*
from *IHAPP*(*1*)[*simplified*] **obtain** *h1 ts t h2 c1' cs' ct' c2'* **where**
[*simp*]: *h* = *h1* @ *ts* # *t* # *h2* *c'* = *c1'* @ *cs'* @ *ct'* @ *c2'* **and**
HSPLIT: *hsem* Δ *c1 h1 c1'* *tsem* Δ (*ps*,*ws*) *ts cs'*
tsem Δ (*p'*, *w* @ *r*) *t ct'* *hsem* Δ *c2 h2 c2'*
by (*fastsimp elim*: *hsem-split hsem-split-single*)
from *tsem-spawn*[*OF C HSPLIT*(*2*,*3*)] **have**
ST: *tsem* Δ (*p*,γ#*r*) (*NSPAWN l ts t*) (*cs'*@*ct'*) **.**
from *hsem-conc-lel*[*OF HSPLIT*(*1*) *ST HSPLIT*(*4*)] **have**
hsem Δ *c* (*h1* @ *NSPAWN l ts t* # *h2*) *c'* **by** *simp*
moreover from *sched-spawn*[*OF IHAPP*(*2*)[*simplified*]] **have**
sched (*h1* @ *NSPAWN l ts t* # *h2*) (*l*#*ll*) **.**
ultimately show *?thesis* **by** *blast*
qed
qed

lemma *sched-correct2*: ⟦ *sched h ll*; *hsem* Δ *c h c'* ⟧ ⟹ (*c*,*ll*,*c'*)∈*dpntrc* Δ
proof (*induct h ll arbitrary*: *c c' rule*: *sched.induct*)
case (*sched-final h c c'*) **thus** *?case* **by** (*auto dest*: *final-hsem-nostep*)
next
case (*sched-nospawn h1 t h2 ll l c c'*)
from *hsem-lel-h*[*OF sched-nospawn.prems*] **obtain** *c1 p*γ*r c2 c1' ct' c2'* **where**
[*simp*]: *c* = *c1* @ *p*γ*r* # *c2* *c'* = *c1'* @ *ct'* @ *c2'* **and**
SPLIT: *hsem* Δ *c1 h1 c1'*
tsem Δ *p*γ*r* (*NNOSPAWN l t*) *ct'*
hsem Δ *c2 h2 c2'*
.
from *SPLIT*(*2*) **obtain** *p* γ *r p' w* **where**
[*simp*]: *p*γ*r*=(*p*,γ#*r*) **and**
ST: (*p*,γ ↪$_l$ *p'*,*w*)∈Δ *tsem* Δ (*p'*,*w*@*r*) *t ct'*
by (*erule-tac tsem.cases*) *fastsimp*+
from *dpntr-no-spawn*[*OF ST*(*1*)] **have** (*c*,*l*,*c1* @ (*p'*, *w* @ *r*) # *c2*)∈*dpntr* Δ
by *auto*
also from *sched-nospawn.hyps*(*2*)[*OF hsem-conc-lel*[*OF SPLIT*(*1*) *ST*(*2*) *SPLIT*(*3*)]]
have
SST: (*c1* @ (*p'*, *w* @ *r*) # *c2*, *ll*, *c1'* @ *ct'* @ *c2'*) ∈ *dpntrc* Δ **.**
finally show *?case* **by** *auto*
next
case (*sched-spawn h1 ts t h2 ll l c c'*)
from *hsem-lel-h*[*OF sched-spawn.prems*] **obtain** *c1 p*γ*r c2 c1' ct' c2'* **where**
[*simp*]: *c* = *c1* @ *p*γ*r* # *c2* *c'* = *c1'* @ *ct'* @ *c2'* **and**
SPLIT: *hsem* Δ *c1 h1 c1'*
tsem Δ *p*γ*r* (*NSPAWN l ts t*) *ct'*
hsem Δ *c2 h2 c2'*
.
from *SPLIT*(*2*) **obtain** *p* γ *r ps ws p' w cts' ctt'* **where**
[*simp*]: *p*γ*r*=(*p*,γ#*r*) *ct'*=*cts'*@*ctt'* **and**
ST: (*p*,γ ↪$_l$ *ps*,*ws* ♯ *p'*,*w*)∈Δ *tsem* Δ (*ps*,*ws*) *ts cts'*

```
      tsem Δ (p′,w@r) t ctt′
    by (erule-tac tsem.cases) fastsimp+
  from dpntr-spawn[OF ST(1)] have
    (c,l,c1 @ (ps,ws) # (p′, w @ r) # c2)∈dpntr Δ
    by auto
  also from sched-spawn.hyps(2)[OF hsem-conc-leel[OF SPLIT(1) ST(2,3) SPLIT(3)]]
have
    SST: (c1 @ (ps,ws) # (p′, w @ r) # c2, ll, c′) ∈ dpntrc Δ
    by simp
  finally show ?case by auto
qed
```

Finally, we formulate the correspondance between the interleaving and the true concurrency semantics as a single equivalence:

```
theorem sched-correct: (c,ll,c′)∈dpntrc Δ ⟷ (∃h. hsem Δ c h c′ ∧ sched h ll)
  by (auto intro: sched-correct1 sched-correct2)
```

As any hedge has at least one schedule, we always get an interleaving execution from a hedge execution:

```
lemma obtain-schedule:
  ⟦ hsem Δ c h c′;
    !!ll. ⟦(c,ll,c′)∈dpntrc Δ; sched h ll ⟧ ⟹ P
  ⟧ ⟹ P
  apply (rule-tac h=h in exists-schedule)
  apply (metis sched-correct)
  done
```

5 Predecessor Sets

Following [2], we define the set of immediate predecessors *pre* Δ *C* and predecessors *pre** Δ *C* of a set of configurations *C*. The set of immediate predecessors contains those configurations from that we can reach (a configuration in) *C* with exactly one step. The set of predecessors contains those configurations from that we can reach *C* with an arbitrary number of steps, including no steps at all (i.e. *pre** is reflexive).

Computing predecessor sets is the key to model checking and analysis of DPNs, see [2] for details.

```
definition pre Δ C′ == { c . ∃l c′. c′∈C′ ∧ (c,l,c′) ∈ dpntr Δ }
definition pre-star (pre*) where
  pre* Δ C′ == { c . ∃ll c′. c′∈C′ ∧ (c,ll,c′) ∈ dpntrc Δ }
```

5.1 Hedge-Constrained Predecessor Sets

For a set of configurations C' and a set of execution hedges H, we define the *hedge-constrained predecessor set* of C' w.r.t. H as the set of those configurations from that we can reach C' with an execution hedge in H.

definition *prehc* Δ *H* C' == { *c* . $\exists$ *h* c'. *h*$\in$*H* $\wedge$ $c'$$\in$$C'$ $\wedge$ *hsem* Δ *c h* c' }

lemma *prehcI*: $[\![$*h*$\in$*H*; $c'$$\in$$C'$; *hsem* Δ *c h* $c'$$]\!]$ $\Longrightarrow$ *c*$\in$*prehc* Δ *H* C'
by (*unfold prehc-def*) *auto*

lemma *prehcE*:
$[\![$*c*$\in$*prehc* Δ *H* C'; !!*h* c'. $[\![$ *h*$\in$*H*; $c'$$\in$$C'$; *hsem* Δ *c h* c' $]\!]$ $\Longrightarrow$ *P* $]\!]$ $\Longrightarrow$ *P*
by (*unfold prehc-def*) *auto*

The hedge-constrained predecessor set is monotonic in the constraint

lemma *prehc-mono*: *H*$\subseteq$$H'$ $\Longrightarrow$ *prehc* Δ *H* C' $\subseteq$ *prehc* Δ H' C'
by (*auto simp add*: *prehc-def*)

The hedge-constrained predecessor set without constraints is the same as the original predecessor set.

lemma *prehc-triv-is-pre-star*: *prehc* Δ *UNIV* C' = pre^* Δ C'
apply (*unfold prehc-def pre-star-def*)
apply *auto*
apply (*rule-tac h*=*h* **in** *exists-schedule*)
apply (*metis sched-correct*)
apply (*metis sched-correct*)
done

The hedge-constrained predecessor set is always a subset of the unconstrained predecessor set.

lemma *prehc-subset-pre-star*: *prehc* Δ *H* C' $\subseteq$ pre^* Δ C'
apply (*unfold prehc-def pre-star-def*)
apply *auto*
apply (*rule-tac h*=*h* **in** *exists-schedule*)
apply (*metis sched-correct*)
done

We can use a hedge-constraint to express immediate predecessor sets.

definition *Hpre* :: ('*P*,'Γ,'*L*) *ex-hedge set* **where**
Hpre == { *hl1*@*t*#*hl2* | *hl1 t hl2 lab ts* t'.
final hl1 $\wedge$ *final hl2* $\wedge$ *final-t ts* $\wedge$ *final-t* t' $\wedge$
(*t*=*NNOSPAWN lab* t' $\vee$ *t*=*NSPAWN lab ts* t') }

lemma *HpreI-nospawn*:
$[\![$*final h1*; *final h2*; *final-t* $t'$$]\!]$ $\Longrightarrow$ *h1*@*NNOSPAWN lab* t'#*h2* $\in$ *Hpre*
by (*unfold Hpre-def*) *blast*

lemma *HpreI-spawn*:
$[\![$*final h1*; *final h2*; *final-t ts*; *final-t* $t'$$]\!]$ $\Longrightarrow$ *h1*@*NSPAWN lab ts* t'#*h2* $\in$ *Hpre*
by (*unfold Hpre-def*) *blast*

lemmas *HpreI* = *HpreI-nospawn HpreI-spawn*

lemma *HpreE*[*cases set*, *consumes 1*, *case-names nospawn spawn*]:

```
  ⟦ h∈Hpre;
    !!h1 lab t' h2. ⟦
      h=h1@NNOSPAWN lab t'#h2; final h1; final h2; final-t t'
    ⟧ ⟹ P;
    !!h1 lab ts t' h2. ⟦
      h=h1@NSPAWN lab ts t'#h2;
      final h1; final h2; final-t ts; final-t t'
    ⟧ ⟹ P
  ⟧ ⟹ P
  by (unfold Hpre-def) blast
```

In order to show that *Hpre* is correct, we first show that it exactly admits the schedules of length one.

```
lemma Hpre-length1: ⟦h∈Hpre; sched h ll⟧ ⟹ length ll = 1
proof (erule HpreE)
  case (goal1 h1 lab t' h2) note C=this — nospawn
  note [simp] = C(2−)
  from C(1) obtain l ll' where ll=l#ll'   sched (h1@t'#h2) ll'
    by (erule-tac sched.cases) (auto dest!: prop-matchD[where P=final-t])
  moreover have final (h1@t'#h2) by auto
  ultimately show ?case by auto
next
  case (goal2 h1 lab ts t' h2) note C=this — spawn
  note [simp] = C(2−)
  from C(1) obtain l ll' where ll=l#ll'   sched (h1@ts#t'#h2) ll'
    by (erule-tac sched.cases) (auto dest!: prop-matchD[where P=final-t])
  moreover have final (h1@ts#t'#h2) by auto
  ultimately show ?case by auto
qed

lemma Hpre-length2: ⟦sched h ll; length ll = 1⟧ ⟹ h∈Hpre
  by (erule sched.cases) (auto intro: HpreI)

theorem Hpre-length: sched h ll ⟹ h∈Hpre ⟷ length ll = 1
  using Hpre-length1 Hpre-length2 by blast
```

It is then straightforward to show that *prehc* Δ *Hpre* = *pre* Δ

```
lemma Hpre-correct1: c∈prehc Δ Hpre C' ⟹ c∈pre Δ C'
  apply (unfold prehc-def)
  apply auto
  apply (rule-tac h=h in exists-schedule)
  apply (simp only: Hpre-length)
  apply (drule (1) sched-correct2)
  apply (case-tac ll)
  apply simp
  apply simp
  apply (auto simp add: pre-def)
  done
```

lemma *Hpre-correct2*: $c \in pre\ \Delta\ C' \Longrightarrow c \in prehc\ \Delta\ Hpre\ C'$
  apply (*unfold pre-def*)
  apply *auto*
  apply (*drule iffD2*[*OF trcl-single*])
  apply (*drule sched-correct1*)
  apply *auto*
  apply (*drule Hpre-length2*)
  apply (*auto simp add*: *prehc-def*)
  done

theorem *Hpre-correct*: $prehc\ \Delta\ Hpre = pre\ \Delta$
  using *Hpre-correct1 Hpre-correct2* **by** (*blast intro*: *ext*)

end

6 DPN Semantics on Lists

theory *ListSemantics*
imports *Semantics*
begin

The interleaving semantics works on configurations that are lists of process configurations.

However, in [2] a DPN configuration is represented as a sequence of control and stack symbols. Each process starts with a control symbol, followed by its stack symbols. The configuration is simply a concatenation of processes. This representation allows the notion of a regular set of configurations as a set of configurations accepted by a FSM.

In this theory, we adopt this representation of configurations, define a semantics directly over this representation, and show that this representation is isomorphic to ours for sequences starting with a control symbol. Note that sequences starting with a stack symbol have no meaningful interpretation, as each process's configuration has to start with a control symbol.

6.1 Definitions

We separate stack and control symbols using a datatype with two constructors:

datatype ($'Q$,$'\Gamma$) *cl-item* = *CTRL* $'Q$ | *STACK* $'\Gamma$
types ($'Q$,$'\Gamma$) *cl* = ($'Q$,$'\Gamma$) *cl-item list*

The mapping from configurations to list-based configurations is straightforward:

fun *pc2cl* :: ($'Q$,$'\Gamma$) *pconf* ⇒ ($'Q$,$'\Gamma$) *cl* **where**
  pc2cl (*p*,*w*) = *CTRL p* # *map STACK w*

definition *c2cl* :: ('*Q*,'T) *conf* ⇒ ('*Q*,'T) *cl* **where**
c2cl c == *concat* (*map pc2cl c*)

abbreviation *c2cl-abbrv* :: ('*Q*,'T) *conf* ⇒ ('*Q*,'T) *cl*
— This abbreviation is just for convenience
where
c2cl-abbrv c == *concat* (*map pc2cl c*)

Valid single-process configurations are those that start with a control symbol followed by a list of stack symbols:

definition *pclvalid* == {*CTRL p*#*map STACK w* | *p w*. *True*}

Valid configurations are those that start with a control symbol:

definition *clvalid* == {[]} ∪ {*CTRL p*#*c* | *p c*. *True*}

We also define the step relation directly on list representation of configurations:

inductive-set *cltr* :: ('*Q*,'T,'*L*) *dpn* ⇒ (('*Q*,'T) *cl* × '*L* × ('*Q*,'T) *cl*) *set*
for Δ **where**
cltr-no-spawn:
⟦ $(p,\gamma \hookrightarrow_l p',w)\in\Delta$ ⟧ ⟹
(*c1*@[*CTRL p*, *STACK* γ]@*c2*,
l,
c1@*CTRL p'*#(*map STACK w*)@*c2*
) ∈ *cltr* Δ |
cltr-spawn:
⟦ $(p,\gamma \hookrightarrow_l ps,ws \sharp p',w)\in\Delta$ ⟧ ⟹
(*c1*@[*CTRL p*, *STACK* γ]@*c2*,
l,
c1@*CTRL ps*#(*map STACK ws*)@*CTRL p'*#(*map STACK w*)@*c2*
) ∈ *cltr* Δ

6.2 Theorems

lemma *inj-STACK*[*simp*, *intro*!]: *inj STACK* **by** (*rule injI*) *auto*

6.2.1 Representation of Single Processes

lemma *pc2cl-not-empty*[*simp*]: *pc2cl* π ≠ [] **by** (*cases* π) *auto*

lemma *pc2cl-inj*[*simp*, *intro*!]: *inj pc2cl*
apply (*rule injI*)
apply (*case-tac x*, *case-tac y*)
apply *simp*
done

lemmas *pc2cl-inj-simp*[*simp*] = *inj-eq*[*OF pc2cl-inj*]

lemma *pc2cl-valid*[*intro*!,*simp*]: *pc2cl* π ∈ *pclvalid*

by (*cases* π) (*auto simp add*: *pclvalid-def*)

lemma *pc2cl-surj*: $[\![\pi l \in pclvalid; !!\pi. \pi l = pc2cl\ \pi \Longrightarrow P]\!] \Longrightarrow P$
apply (*unfold pclvalid-def*)
apply (*cases* πl)
apply *simp*
apply *fastsimp*
done

6.2.2 Representation of Configurations

We start with a bunch of simplification rules and other auxilliary lemmas:

lemma *stack-no-ctrl1*[*simp*]:
map STACK w $\neq$ *c1@CTRL p#c2*
by (*auto elim*!: *map-eq-concE*)

lemmas *stack-no-ctrl2*[*simp*] = *stack-no-ctrl1*[*symmetric*]

lemma *map-stack-ne-cCc1* [*simp*]:
map STACK w $\neq$ *c@CTRL s#c'*
apply (*induct w arbitrary*: *c s c'*)
apply *auto*
apply (*case-tac c*)
apply *auto*
done
lemmas *map-stack-ne-cCc2*[*simp*] = *map-stack-ne-cCc1*[*symmetric*]

lemmas *map-stack-ne-add-simps*[*simp*] =
map-stack-ne-cCc1[**where** *c*=[], *simplified*]
map-stack-ne-cCc1[**where** *c*=[*a*], *simplified*, *standard*]

lemma *map-STACK-eq-map-STACK-simp*[*simp*]:
map STACK w @CTRL p # cl = map STACK w' @ CTRL p' # cl' $\longleftrightarrow$
w'=w $\wedge$ *p'=p* $\wedge$ *cl'=cl*
apply (*induct w arbitrary*: *w'*)
apply (*case-tac w'*)
apply *auto*[*2*]
apply (*case-tac w'*)
apply *auto*
done

lemma *map-stack-ne-pc2cl*[*simp*]:
map STACK w $\neq$ *c@pc2cl* π*@c'*
c@pc2cl π*@c'* $\neq$ *map STACK w*
by (*cases* π, *auto*)+

lemmas *map-stack-ne-pc2cl-add-simps*[*simp*] =

map-stack-ne-pc2cl[**where** *c*=[], *simplified*]

lemma *map-STACK-eq-map-STACK-add-simps*[*simp*]:
  map STACK w @ *CTRL p*#*cl* = *map STACK w'*@*pc2cl* π'@*cl'* ⟷
    w=*w'* ∧ *p*=*fst* π' ∧ *cl*=*map STACK* (*snd* π')@*cl'*
  map STACK w'@*pc2cl* π'@*cl'* = *map STACK w* @ *CTRL p*#*cl* ⟷
    w=*w'* ∧ *p*=*fst* π' ∧ *cl*=*map STACK* (*snd* π')@*cl'*
  by (*cases* π', *auto*)+

lemma *c2cl-simps*[*simp*]:
  c2cl [] = []
  c2cl (π#*c*) = *pc2cl* π @ *c2cl c*
  c2cl (*c1*@*c2*) = *c2cl c1* @ *c2cl c2*
  by (*unfold c2cl-def*) *auto*

lemma *c2cl-empty*[*simp*]:
  c2cl c = [] ⟷ *c*=[]
  [] = *c2cl c* ⟷ *c*=[]
  by (*cases c*, *auto*)+

lemma *c2cl-start-with-ctrl*[*simp*]:
  c2cl c ≠ *STACK* γ#*cl*
  STACK γ#*cl* ≠ *c2cl c*
  by (*cases c*, *auto*)+

lemma *c2cl-start-with-ctrl-map*:
  w≠[] ⟹ *c2cl c* ≠ *map STACK w*
  w≠[] ⟹ *map STACK w* ≠ *c2cl c*
  by (*cases w*, *auto*)+

lemma *map-stack-c2cl-eq-simps*[*simp*]:
  map STACK w @ *c2cl c* = *map STACK w'* @ *c2cl c'* ⟷ *w*=*w'* ∧ *c2cl c*=*c2cl*
c'
  apply (*rule iffI*)
  defer
  apply *simp*
  apply (*induct w arbitrary*: *w'*)
  apply (*case-tac w'*)
  apply *auto*
  apply (*case-tac w'*)
  apply *auto*
  apply (*case-tac w'*)
  apply *auto*
  done

lemma *c2cl-s-cl-eqE*:

```
  ⟦ STACK γ # cl = map STACK w @ c2cl c;
    !!wr. ⟦ w=γ#wr; cl = map STACK wr @ c2cl c ⟧ ⟹ P
  ⟧ ⟹ P
  by (cases w) auto

lemma c2cl-first-processE:
  ⟦ c2cl c = CTRL p#cl2;
    !!w c2 cl2'. ⟦ c=(p,w)#c2; cl2=(map STACK w)@cl2'; c2cl c2=cl2' ⟧ ⟹ P
  ⟧ ⟹ P
  apply (cases c)
  apply simp
  apply simp
  apply (case-tac a)
  apply simp
  apply blast
  done

lemma c2cl-find-process1:
  ⟦ c2cl c = cl1@CTRL p#cl2;
    !!c1 w c2. ⟦ c=c1@(p,w)#c2; cl2=(map STACK w)@c2cl c2;
                 cl1=c2cl c1
               ⟧ ⟹ P
  ⟧ ⟹ P
proof (induct cl1 arbitrary: c P rule: length-compl-induct)
  case Nil thus ?case by (force elim!: c2cl-first-processE)
next
  case (Cons e cl1') show ?case proof (cases e)
    case (STACK γ) with Cons.prems(1) have False by simp thus ?thesis ..
  next
    case (CTRL p')[simp]
    from Cons.prems(1) have E: c2cl c = CTRL p' # (cl1'@CTRL p#cl2) by
simp
    from c2cl-first-processE[OF E] obtain w c2 cl2' where
      [simp]: c = (p', w) # c2 and
        S: cl1' @ CTRL p # cl2 = map STACK w @ cl2'    c2cl c2 = cl2'
      .
    obtain cl1'2 where [simp]: cl1'=map STACK w @ cl1'2
    proof -
      from S(1) have take (length w) (cl1'@CTRL p#cl2) = map STACK w by
auto
      hence map STACK w = take (length w) cl1'
        by (cases length w - length cl1') auto
      hence cl1'=map STACK w @ drop (length w) cl1' by auto
      thus ?thesis using that by blast
    qed
    with S have
      P: c2cl c2=cl1'2@CTRL p#cl2 and
      LEN: length cl1'2 ≤ length cl1' by auto
    from Cons.hyps[OF LEN P] obtain c1x wx c2x where
```

```
    IHAPP: c2 = c1x@(p,wx)#c2x
        cl2=map STACK wx @ c2cl c2x and
    [simp]: cl1'2 = c2cl c1x
    by metis
  hence 1: c=((p',w)#c1x)@(p,wx)#c2x by auto
  show ?thesis by (rule Cons.prems(2)[OF 1 IHAPP(2)]) auto
 qed
qed
```

Then we show that our representation mapping is injective and surjective on valid configurations.

```
lemma c2cl-inj[simp, intro!]: inj c2cl
 apply (rule injI)
proof -
 case (goal1 c c')
 thus ?case proof (induct c arbitrary: c')
  case Nil thus ?case by auto
 next
  case (Cons π c)
  thus ?case
   apply (cases c')
   apply simp
   apply simp
   apply (cases π)
   apply (case-tac a)
   apply auto
   done
 qed
qed

lemmas c2cl-inj-simps[simp] = inj-eq[OF c2cl-inj]
lemmas c2cl-img-Int[simp] = image-Int[OF c2cl-inj]

lemma c2cl-valid[simp,intro!]: c2cl c ∈ clvalid
 by (cases c) (auto simp add: clvalid-def)

lemma c2cl-surj: ⟦cl∈clvalid; !!c. cl=c2cl c ⟹ P⟧ ⟹ P
 apply (unfold clvalid-def)
 apply auto
proof -
 case goal1 thus ?case proof (induct c arbitrary: p)
  case Nil from Nil[of [(p,[])]] show ?case by auto
 next
  case (Cons s c) show ?case
   apply (cases s)
   apply (rule-tac p=Q in Cons.hyps)
   apply (rule-tac c=(p,[])#c in Cons.prems)
   apply simp
   apply (rule-tac p=p in Cons.hyps)
```

```
    apply (case-tac c)
    apply simp
    apply (case-tac a)
    apply simp
    apply (rule-tac c=(p,Γ#b)#list in Cons.prems)
    apply simp
    done
  qed
qed
```

6.2.3 Step Relation on List-Configurations

```
lemma cltr-pres-valid: (cl,l,cl')∈cltr Δ ⟹ cl∈clvalid ⟷ cl'∈clvalid
  apply (erule cltr.cases)
  apply (auto simp add: clvalid-def)
  apply (case-tac c1)
  apply auto
  apply (case-tac c1)
  apply auto
  apply (case-tac c1)
  apply auto
  apply (case-tac c1)
  apply auto
  done

lemma dpntr-is-cltr: ⟦(c,l,c')∈dpntr Δ⟧ ⟹ (c2cl c,l,c2cl c')∈cltr Δ
  apply (erule dpntr.cases)
  apply (unfold c2cl-def)
  apply (auto)
  apply (drule-tac ?c2.0=map STACK r@c2cl-abbrv c2 in cltr-no-spawn)
  apply simp
  apply (drule-tac ?c2.0=map STACK r@c2cl-abbrv c2 in cltr-spawn)
  apply simp
done

lemma cltr-is-dpntr: ⟦ (c2cl c,l,c2cl c')∈cltr Δ ⟧ ⟹ (c,l,c')∈dpntr Δ
  apply (erule cltr.cases)
  apply auto
  apply (erule c2cl-find-process1)
  apply (erule c2cl-find-process1)
  apply auto
  apply (erule c2cl-s-cl-eqE)
  apply (auto simp del: map-append append-assoc
          simp add: map-append[symmetric] append-assoc[symmetric]
          intro: dpntr-no-spawn)
  apply (erule c2cl-find-process1)
  apply (erule c2cl-find-process1)
  apply auto
  apply (erule c2cl-s-cl-eqE)
```

apply *auto*
apply (*case-tac c2b*)
apply *simp*
apply (*case-tac a*)
apply (*auto simp del*: *map-append append-assoc*
simp add: *map-append*[*symmetric*] *append-assoc*[*symmetric*]
intro: *dpntr-spawn*)
done

The following theorem formulates the equivalence of the original semantics and the list-based semantics.

theorem *cltr-eq-dpntr*: $(c2cl\ c,l,c2cl\ c')\in cltr\ \Delta \longleftrightarrow (c,l,c')\in dpntr\ \Delta$
by (*metis cltr-is-dpntr dpntr-is-cltr*)

The next two lemmas ease the derivation of executions of the original semantics from executions of the list-based semantics.

lemma *cltr2dpntr-fwd*:
$[\![$ $(c2cl\ c,l,cl')\in cltr\ \Delta$;
$!!c'.\ [\![cl'=c2cl\ c';\ (c,l,c')\in dpntr\ \Delta]\!] \Longrightarrow P$
$]\!] \Longrightarrow P$
proof −
assume
A: $(c2cl\ c,l,cl')\in cltr\ \Delta$ **and**
C: $!!c'.\ [\![cl'=c2cl\ c';\ (c,l,c')\in dpntr\ \Delta]\!] \Longrightarrow P$
from *cltr-pres-valid*[*OF A*] **have** *V*: $cl'\in clvalid$ **by** *auto*
from *c2cl-surj*[*OF V*] **obtain** c' **where** [*simp*]: $cl'=c2cl\ c'$ **.**
from *A* **show** *?thesis* **by** (*auto intro*: *C simp add*: *cltr-is-dpntr*)
qed

lemma *cltr2dpntr-bwd*:
$[\![$ $(cl,l,c2cl\ c')\in cltr\ \Delta$;
$!!c.\ [\![cl=c2cl\ c;\ (c,l,c')\in dpntr\ \Delta]\!] \Longrightarrow P$
$]\!] \Longrightarrow P$
proof −
assume
A: $(cl,l,c2cl\ c')\in cltr\ \Delta$ **and**
C: $!!c.\ [\![cl=c2cl\ c;\ (c,l,c')\in dpntr\ \Delta]\!] \Longrightarrow P$
from *cltr-pres-valid*[*OF A*] **have** *V*: $cl\in clvalid$ **by** *auto*
from *c2cl-surj*[*OF V*] **obtain** c **where** [*simp*]: $cl=c2cl\ c$ **.**
from *A* **show** *?thesis* **by** (*auto intro*: *C simp add*: *cltr-is-dpntr*)
qed

Finally, we give some lemmas to directly reason about the transitive closure of the step relation:

lemma *cltr-is-dpntrc*:
$(c2cl\ c,l,c2cl\ c')\in trcl\ (cltr\ \Delta) \Longrightarrow (c,l,c')\in dpntrc\ \Delta$
by (*induct l arbitrary*: *c*) (*auto elim*!: *trcl-unconsE cltr2dpntr-fwd*)

lemma *dpntrc-is-cltr*:

$(c,l,c')\in dpntrc\ \Delta \Longrightarrow (c2cl\ c,l,c2cl\ c')\in trcl\ (cltr\ \Delta)$
by (*induct rule*: *trcl.induct*) (*auto dest*: *dpntr-is-cltr*)

theorem *cltr-eq-dpntrc*:
$(c2cl\ c,l,c2cl\ c')\in trcl\ (cltr\ \Delta) \longleftrightarrow (c,l,c')\in dpntrc\ \Delta$
apply *safe*
apply (*induct l arbitrary*: *c*)
apply (*auto elim*!: *trcl-unconsE cltr2dpntr-fwd*)
apply (*induct rule*: *trcl.induct*)
apply (*auto dest*: *dpntr-is-cltr*)
done

lemma *cltrc-pres-valid*:
$(cl,w,cl')\in trcl\ (cltr\ \Delta) \Longrightarrow cl\in clvalid \longleftrightarrow cl'\in clvalid$
by (*induct rule*: *trcl.induct*) (*auto simp add*: *cltr-pres-valid*)

lemma *cltr2dpntrc-fwd*:
$[\![\ (c2cl\ c,l,cl')\in trcl\ (cltr\ \Delta);$
$!!c'.\ [\![cl'=c2cl\ c';\ (c,l,c')\in dpntrc\ \Delta]\!] \Longrightarrow P$
$]\!] \Longrightarrow P$
proof −
assume
A: $(c2cl\ c,l,cl')\in trcl\ (cltr\ \Delta)$ **and**
C: $!!c'.\ [\![cl'=c2cl\ c';\ (c,l,c')\in dpntrc\ \Delta]\!] \Longrightarrow P$
from *cltrc-pres-valid*[*OF A*] **have** V: $cl'\in clvalid$ **by** *auto*
from *c2cl-surj*[*OF V*] **obtain** c' **where** [*simp*]: $cl'=c2cl\ c'$ **.**
from *A* **show** *?thesis* **by** (*auto intro*: *C simp add*: *cltr-is-dpntrc*)
qed

lemma *cltr2dpntrc-bwd*:
$[\![\ (cl,l,c2cl\ c')\in trcl\ (cltr\ \Delta);$
$!!c.\ [\![cl=c2cl\ c;\ (c,l,c')\in dpntrc\ \Delta]\!] \Longrightarrow P$
$]\!] \Longrightarrow P$
proof −
assume
A: $(cl,l,c2cl\ c')\in trcl\ (cltr\ \Delta)$ **and**
C: $!!c.\ [\![cl=c2cl\ c;\ (c,l,c')\in dpntrc\ \Delta]\!] \Longrightarrow P$
from *cltrc-pres-valid*[*OF A*] **have** V: $cl\in clvalid$ **by** *auto*
from *c2cl-surj*[*OF V*] **obtain** c **where** [*simp*]: $cl=c2cl\ c$ **.**
from *A* **show** *?thesis* **by** (*auto intro*: *C simp add*: *cltr-is-dpntrc*)
qed

6.3 Predecessor Sets on List-Semantics

We also define predecessor sets for the list-semantics:

definition *precl* (pre_{cl}) **where**
$pre_{cl}\ \Delta\ C' == \{\ c\ .\ \exists\, l\ c'.\ c'\in C' \wedge (c,l,c') \in cltr\ \Delta\ \}$
definition *precl-star* ($pre^*{}_{cl}$) **where**

$pre^*{}_{cl}\ \Delta\ C' == \{\ c\ .\ \exists\, ll\ c'.\ c' \in C' \wedge (c,ll,c') \in trcl\ (cltr\ \Delta)\ \}$

And show that they are equivalent to their counterparts defined over the original semantics:

lemma *precl-is-pre*: $pre_{cl}\ \Delta\ (c2cl`C) = c2cl`(pre\ \Delta\ C)$
 apply (*unfold precl-def pre-def*)
 apply (*auto elim*!: *cltr2dpntr-bwd intro*: *dpntr-is-cltr*)
 done

lemma *precl-star-is-pre-star*: $pre^*{}_{cl}\ \Delta\ (c2cl`C) = c2cl`(pre^*\ \Delta\ C)$
 apply (*unfold precl-star-def pre-star-def*)
 apply (*auto elim*!: *cltr2dpntrc-bwd intro*: *dpntrc-is-cltr*)
 done

end

7 Automata for Execution Hedges

theory *HedgeAutomata*
imports *Main Semantics*
begin

In this section we define hedge automata that accept execution hedges.

A hedge automaton consists of a set of states, an regular *initial language* of state sequences and a set of transitions. Transitions are either leaf transitions that label a leaf node with a state if the configuration at the leaf node is contained in some (regular) language, or non-spawning or spawning transitions, that label a spawning or non-spawning node respectively with a state depending on the states of the successor nodes.

In this formalization, we model the initial language and the regular languages at the leafs just at sets. However, if we want an executable representation, we need to model real automata there. This is planned to be done in the future.

datatype *('S,'P,'T,'L) ha-rule* =
 HAR-LEAF 'S 'P 'T list set |
 HAR-NOSPAWN 'S 'L 'S |
 HAR-SPAWN 'S 'L 'S 'S

types *('S,'P,'T,'L) ha* = *'S list set* × *('S,'P,'T,'L) ha-rule set*

In order to model acceptance of a hedge, we define a relation between trees and states with which we can label those trees. We then extend this relation to hedges.

inductive *lab*

:: $('S,'P,'\Gamma,'L)$ *ha-rule set* ⇒ $('P,'\Gamma,'L)$ *ex-tree* ⇒ *'S* ⇒ *bool*
for *H* **where**
lab-leaf:
⟦ *HAR-LEAF s p W* ∈ *H*; *w*∈*W* ⟧ ⟹ *lab H* (*NLEAF* (*p*,*w*)) *s* |
lab-nospawn:
⟦ *HAR-NOSPAWN s l s'* ∈ *H*; *lab H t s'* ⟧ ⟹ *lab H* (*NNOSPAWN l t*) *s* |
lab-spawn:
⟦ *HAR-SPAWN s l ss s'* ∈ *H*; *lab H ts ss*; *lab H t s'* ⟧ ⟹
lab H (*NSPAWN l ts t*) *s*

inductive *labh* :: $('S,'P,'\Gamma,'L)$ *ha-rule set* ⇒ $('P,'\Gamma,'L)$ *ex-hedge* ⇒ *'S list* ⇒ *bool*

for *H* **where**
labh-empty[*simp*, *intro*!]: *labh H* [] [] |
labh-cons: ⟦ *lab H t s*; *labh H h σ*⟧ ⟹ *labh H* (*t*#*h*) (*s*#*σ*)

lemma *labh-empty*[*simp*]:
labh H [] *σ* ⟷ *σ*=[]
labh H h [] ⟷ *h*=[]
by (*auto elim*: *labh.cases*)

lemma *labh-length*: *labh H h σ* ⟹ *length h* = *length σ*
by (*induct rule*: *labh.induct*) *auto*

The language of a hedge automaton consists of those hedges whose roots can be labeled with a state sequence in the initial language.

definition *langh* :: $('S,'P,'\Gamma,'L)$ *ha* ⇒ $('P,'\Gamma,'L)$ *ex-hedge set* **where**
langh HA == { *h* . ∃*σ*∈*fst HA*. *labh* (*snd HA*) *h σ* }

end

8 Computation of Hedge-Constrained Predecessor Sets

theory *CrossProd*
imports *ListSemantics HedgeAutomata*
begin

In this section we show how to compute predecessor sets with regular hedge constraints. The computation is done by reduction to the computation of the unconstrained predecessor set. The reduction uses a cross-product like approach, computing a product-DPN of the original DPN and the hedge automaton, and a product regular set of the original regular set and the hedge-automaton's leaf rules.

This theory uses a list-based representation of DPN-configurations, where the type of a configuration is a list of control- and stack-symbols. This type is less structured than the original type of configurations, that is lists of pairs of control symbol and stack. However, it admits handling configurations as words, and sets of configurations as (regular) languages.

This theory does not use a formalization of regular languages, nor does it generate executable code. Instead, regular sets are modeled as sets. The effectiveness proofs show representations that only contain operations well-known to preserve regularity. However, an implementation of those operations is not formalized.

The cross-product DPN simulates the rules of the hedge-automaton via its transitions, the current state of the hedge automaton is stored in the DPN's state:

inductive-set
  xdpn :: $('P,'\Gamma,'L)$ *dpn* ⇒ $('S,'P,'\Gamma,'L)$ *ha-rule set* ⇒ $('P\times'S,'\Gamma,'L)$ *dpn*
  for Δ H **where**
  xdpn-nospawn:
    $[\![(p,\gamma \hookrightarrow_l p',w)\in\Delta;\ HAR\text{-}NOSPAWN\ s\ l\ s'\in H]\!] \Longrightarrow$
      $((p,s),\gamma \hookrightarrow_l (p',s'),w) \in xdpn\ \Delta\ H$ |
  xdpn-spawn:
    $[\![(p,\gamma \hookrightarrow_l ps,ws \sharp\ p',w)\in\Delta;\ HAR\text{-}SPAWN\ s\ l\ ss\ s'\in H]\!] \Longrightarrow$
      $((p,s),\gamma \hookrightarrow_l (ps,ss),ws \sharp\ (p',s'),w)\in xdpn\ \Delta\ H$

The *xdpn-nospawn*-rule adds a transition rule to the cross-product DPN for each original non-spawning transition rule and hedge automaton rule that could be used to label the node generated by this transition rule. Analogously, the *xdpn-spawn*-rule adds a transition rule to the cross-product DPN for spawning rules.

We now define operators to map configurations of the cross-product DPN to configurations of the original DPN and sequences of states of the hedge automaton.

abbreviation
  proj-c1 :: $('P\times'S,'\Gamma)$ *conf* ⇒ $('P,'\Gamma)$ *conf* **where**
  proj-c1 *cx* == *map* $(\lambda((p,s),w).\ (p,w))$ *cx*
abbreviation
  proj-c2 :: $('P\times'S,'\Gamma)$ *conf* ⇒ *'S list* **where**
  proj-c2 *cx* == *map* $(\lambda((p,s),w).\ s)$ *cx*

We also have to define a mapping for execution hedges, because the labeling of the leafs is different:

fun *proj-t1* :: $('P\times'S,'\Gamma,'L)$ *ex-tree* ⇒ $('P,'\Gamma,'L)$ *ex-tree* **where**
  proj-t1 (*NLEAF* $((p,s),w)$) = *NLEAF* (p,w) |
  proj-t1 (*NNOSPAWN* *l* *t*) = *NNOSPAWN* *l* (*proj-t1* *t*) |
  proj-t1 (*NSPAWN* *l* *ts* *t*) = *NSPAWN* *l* (*proj-t1* *ts*) (*proj-t1* *t*)

Next we define how to transform the target set, that contains the configurations of that we want to compute the predecessors.

The new target set contains the configurations of the original target set with all labelings that may be done by leaf-rules of the hedge automaton:

— Process labeled by a leaf-rule:
abbreviation
xdpnCLP H == { ((*p*,*s*),*w*). ∃ *W*. *HAR-LEAF s p W* ∈ *H* ∧ *w*∈*W* }

— Configuration labeled by leaf-rules:
abbreviation
xdpnCL H == { *cx* . (∀((*p*,*s*),*w*)∈*set cx*. ((*p*,*s*),*w*) ∈ *xdpnCLP H*) }

— New target set:
definition
xdpnC C H == { *cx* . *proj-c1 cx* ∈ *C* } ∩ *xdpnCL H*

Finally we define how to transform the computed predecessor set in order to get a set of configurations of the original DPN. This phase consists of two operations: First, we have to restrict the configurations to those that are accepted by the hedge automaton's initial language, and then we have to project away the hedge-automaton's states to get a configuration of the original DPN. In the following definition, these two steps are combined:

definition
projH :: *'S list set* ⇒ (*'P*×*'S*,*'Γ*) *conf set* ⇒ (*'P*,*'Γ*) *conf set* **where**
projH H0 Cx == { *proj-c1 cx* | *cx*. *cx*∈*Cx* ∧ *proj-c2 cx* ∈ *H0* }

8.1 Correctness of Reduction

In this section we show that our reduction is correct, i.e. that we really get the hedge-constrained predecessor set by computing the predecessor set of the cross-product DPN and a transformed target set, and then applying the *projH*-operator to the result.

We first need to introduce a combination operator that combines an original DPN's configuration and a list of hedge automaton states to a cross-product DPN's configuration.

abbreviation *cxs c* σ == *zipf* (λ(*p*,*w*) *s*. ((*p*,*s*),*w*)) *c* σ

lemma *proj-cxs1*[*simp*]: *length c* = *length* σ ⟹ *proj-c1* (*cxs c* σ) = *c*
by (*induct rule*: *list-induct2*) *auto*

lemma *proj-cxs2*[*simp*]: *length c* = *length* σ ⟹ *proj-c2* (*cxs c* σ) = σ
by (*induct rule*: *list-induct2*) *auto*

lemma *cxs-proj*[*simp*]: *cxs* (*proj-c1 cx*) (*proj-c2 cx*) = *cx*
by (*induct cx*) *auto*

lemma *xdpnc-proj*: *cx* ∈ *xdpnC C H* ⟹ *proj-c1 cx* ∈ *C*
by (*unfold xdpnC-def*) *auto*

We now prove the two directions of our main goal. Each direction requires 2 lemmas, the first one for a single tree and the second one for a hedge.

```
lemmas tsem-induct-x =
  tsem.induct[ where ?x1.0 = ((p,s),w), split-format (complete),
             consumes 1, case-names tsem-leaf tsem-nospawn tsem-spawn
           ]

lemmas tsem-induct-p =
  tsem.induct[ where ?x1.0 = (p,w), split-format (complete),
             consumes 1, case-names tsem-leaf tsem-nospawn tsem-spawn
           ]

lemma xdpn-correct1-t:
  ⟦tsem (xdpn Δ H) ((p,s),w) t c'; c'∈xdpnCL H⟧ ⟹
    tsem Δ (p,w) (proj-t1 t) (proj-c1 c') ∧ lab H (proj-t1 t) s
proof (induct arbitrary: C rule: tsem-induct-x)
  case (tsem-leaf p s w) thus ?case by (auto intro: lab.intros)
next
  case (tsem-nospawn p s γ l p' s' w r t c' ) thus ?case
    by (auto elim: xdpn.cases intro: lab.intros tsem.intros)
next
  case (tsem-spawn p s γ l ps ss ws p' s' w ts cs r t c') thus ?case
    by (auto elim: xdpn.cases intro: lab.intros tsem.intros)
qed

lemma xdpn-correct1:
  ⟦ hsem (xdpn Δ H) c h c'; c'∈xdpnCL H ⟧ ⟹
     hsem Δ (proj-c1 c) (map proj-t1 h) (proj-c1 c') ∧
     labh H (map proj-t1 h) (proj-c2 c)
proof (induct arbitrary: C' rule: hsem.induct)
  case hsem-empty thus ?case by auto
next
  case (hsem-cons π t cf' c h c')
  obtain p s w where [simp]: π=((p,s),w) by (cases π) auto
  from hsem-cons.prems have CLHS: cf'∈xdpnCL H    c'∈xdpnCL H by auto
  from xdpn-correct1-t[OF hsem-cons.hyps(1)[simplified] CLHS(1)]
       hsem-cons.hyps(3)[OF CLHS(2)]
  show ?case by (auto intro: labh.intros hsem.intros)
qed

lemma xdpn-correct2-t:
  ⟦tsem Δ (p,w) t c'; lab H t s⟧ ⟹
     ∃ tx cx'. tsem (xdpn Δ H) ((p,s),w) tx cx' ∧
             cx'∈xdpnCL H ∧ proj-t1 tx = t ∧
             proj-c1 cx' = c'
proof (induct arbitrary: s rule: tsem-induct-p)
  case (tsem-leaf p w s) thus ?case
    apply (rule-tac x=NLEAF ((p,s),w) in exI)
    apply (rule-tac x=[((p,s),w)] in exI)
    by (auto elim: lab.cases)
next
```

case (*tsem-nospawn p γ l p′ w r t c′ s*)
from *tsem-nospawn.prems* **obtain** *s′* **where**
HRULE: *HAR-NOSPAWN s l s′∈H* *lab H t s′*
by (*auto elim*: *lab.cases*)
from *tsem-nospawn.hyps*(*3*)[*OF HRULE*(*2*)] **obtain** *tx cx′* **where**
IHAPP: *tsem* (*xdpn* Δ *H*) ((*p′*, *s′*), *w* @ *r*) *tx cx′*
cx′ ∈ *xdpnCL H* *proj-t1 tx* = *t* *proj-c1 cx′* = *c′*
by *blast*
from *tsem.intros*(*2*)[*OF xdpn-nospawn*[*OF tsem-nospawn.hyps*(*1*) *HRULE*(*1*)] *IHAPP*(*1*)]
have *tsem* (*xdpn* Δ *H*) ((*p*, *s*), γ # *r*) (*NNOSPAWN l tx*) *cx′* **.**
thus *?case* **using** *IHAPP*(*2*,*3*,*4*) **by** *fastsimp*
next
case (*tsem-spawn p γ l ps ws p′ w ts cs r t c′ s*)
from *tsem-spawn.prems* **obtain** *ss s′* **where**
HRULE: *HAR-SPAWN s l ss s′∈H* *lab H ts ss* *lab H t s′*
by (*auto elim*: *lab.cases*)
from *tsem-spawn.hyps*(*3*)[*OF HRULE*(*2*)] *tsem-spawn.hyps*(*5*)[*OF HRULE*(*3*)]

obtain *txs cxs tx cx′* **where**
IHAPPS: *tsem* (*xdpn* Δ *H*) ((*ps*, *ss*), *ws*) *txs cxs*
cxs ∈ *xdpnCL H* *proj-t1 txs* = *ts* *proj-c1 cxs* = *cs* **and**
IHAPP: *tsem* (*xdpn* Δ *H*) ((*p′*, *s′*), *w* @ *r*) *tx cx′* *cx′* ∈ *xdpnCL H*
proj-t1 tx = *t* *proj-c1 cx′* = *c′*
by *blast*
from *tsem.intros*(*3*)[*OF xdpn-spawn*[*OF tsem-spawn.hyps*(*1*) *HRULE*(*1*)]
IHAPPS(*1*) *IHAPP*(*1*)]
have *tsem* (*xdpn* Δ *H*) ((*p*, *s*), γ # *r*) (*NSPAWN l txs tx*) (*cxs* @ *cx′*) **.**
thus *?case* **using** *IHAPPS*(*2*,*3*,*4*) *IHAPP*(*2*,*3*,*4*) **by** *fastsimp*
qed

lemma *xdpn-correct2*:
⟦ *hsem* Δ *c h c′*; *labh H h σ* ⟧ ⟹
∃ *hx cx′*. *hsem* (*xdpn* Δ *H*) (*cxs c σ*) *hx cx′* ∧
cx′∈*xdpnCL H* ∧
(*map proj-t1 hx*) = *h* ∧
proj-c1 cx′ = *c′*
proof (*induct arbitrary*: σ *rule*: *hsem.induct*)
case *hsem-empty* **thus** *?case* **by** (*auto*)
next
case (*hsem-cons* π *t cf′ c h c′* σ)
from *hsem-cons.prems* **obtain** *s σs* **where**
[*simp*]: σ=*s*#σ*s* **and**
LS: *lab H t s* *labh H h σs*
by (*fastsimp elim*: *labh.cases*)
from *hsem-cons.hyps*(*3*)[*OF LS*(*2*)] **obtain** *hx cx′* **where**
IHAPP: *hsem* (*xdpn* Δ *H*) (*cxs c σs*) *hx cx′*
cx′ ∈ *xdpnCL H*

map proj-t1 hx = h
proj-c1 cx' = c'
by *blast*
moreover obtain *p w* **where** [*simp*]: *π=(p,w)* **by** (*cases π*) *auto*
from *xdpn-correct2-t*[*OF hsem-cons.hyps*(*1*)[*simplified*] *LS*(*1*)]
obtain *tx cfx'* **where**
tsem (*xdpn Δ H*) ((*p*, *s*), *w*) *tx cfx'*
cfx' ∈ *xdpnCL H*
proj-t1 tx = t
proj-c1 cfx' = cf'
by *blast*
ultimately show *?case*
apply (*rule-tac x=tx#hx* **in** *exI*)
apply (*rule-tac x=cfx'@cx'* **in** *exI*)
by (*auto intro*: *hsem.intros*)
qed

Finally we use the lemmas proven above to show our main goal, i.e. a representation of the hedge-constrained predecessor set w.r.t. the language of a hedge automaton by means of the sequential *pre**-operator and the cross-product construction.

theorem *xdpn-correct*:
prehc Δ (*langh* (*H0*,*H*)) *C' = projH H0* (*pre** (*xdpn Δ H*) (*xdpnC C' H*))
proof (*intro equalityI subsetI*)
fix *c*
assume *A*: *c* ∈ *prehc Δ* (*langh* (*H0*, *H*)) *C'*
then obtain *c' h* **where**
D: *c'∈C'* *hsem Δ c h c'* *h∈langh* (*H0*,*H*)
by (*unfold prehc-def*) *auto*
then obtain *σ* **where** *DD*: *σ∈H0* *labh H h σ* **by** (*unfold langh-def*) *auto*

— We need the following later in order to reason about the (underdefined) *cxs*-operator:
from *hsem-length*[*OF D*(*2*)] *labh-length*[*OF DD*(*2*)] **have**
[*simp*]: *length c = length σ*
by *simp*
from *xdpn-correct2*[*OF D*(*2*) *DD*(*2*)] **obtain** *hx cx'* **where**
M: *hsem* (*xdpn Δ H*) (*cxs c σ*) *hx cx'*
cx' ∈ *xdpnCL H*
map proj-t1 hx = h
proj-c1 cx' = c'
by *blast*
from *M*(*2*,*4*) *D*(*1*) **have** *cx'∈xdpnC C' H* **by** (*unfold xdpnC-def*) *auto*
hence *cxs c σ* ∈ *pre** (*xdpn Δ H*) (*xdpnC C' H*)
by (*rule-tac obtain-schedule*[*OF M*(*1*)]) (*auto simp add*: *pre-star-def*)
with *DD*(*1*) **show** *c* ∈ *projH H0* (*pre** (*xdpn Δ H*) (*xdpnC C' H*))
apply (*unfold projH-def*)
apply *auto*
apply (*rule-tac x=cxs c σ* **in** *exI*)

apply *auto*
done
next
fix *c*
assume *A*: $c \in projH\ H0\ (pre^*\ (xdpn\ \Delta\ H)\ (xdpnC\ C'\ H))$
then obtain *cx* **where**
D: $c{=}proj\text{-}c1\ cx \quad proj\text{-}c2\ cx \in H0 \quad cx{\in}pre^*\ (xdpn\ \Delta\ H)\ (xdpnC\ C'\ H)$
by (*unfold projH-def*) *auto*
then obtain *ll cx'* **where**
DD: $cx'{\in}(xdpnC\ C'\ H) \quad (cx,\ ll,\ cx'){\in}dpntrc\ (xdpn\ \Delta\ H)$
by (*unfold pre-star-def*) *auto*
then obtain *hx* **where** *DDH*: $hsem\ (xdpn\ \Delta\ H)\ cx\ hx\ cx'$
by (*auto simp add*: *sched-correct*)
from *DD*(*1*) **have** *CL*: $cx'{\in}xdpnCL\ H \quad proj\text{-}c1\ cx' \in C'$
by (*unfold xdpnC-def*) *auto*
from *xdpn-correct1*[*OF DDH CL*(*1*)] **have**
M: $hsem\ \Delta\ (proj\text{-}c1\ cx)\ (map\ proj\text{-}t1\ hx)\ (proj\text{-}c1\ cx')$
$labh\ H\ (map\ proj\text{-}t1\ hx)\ (proj\text{-}c2\ cx)$
by *auto*
from *D*(*2*) *M*(*2*) **have** $(map\ proj\text{-}t1\ hx){\in}langh\ (H0,H)$
by (*unfold langh-def*) *auto*
with *M*(*1*) *D*(*1*) *CL*(*2*) **show** $c \in prehc\ \Delta\ (langh\ (H0,\ H))\ C'$
by (*unfold prehc-def*) *auto*
qed

8.2 Effectiveness of Reduction

In this section we give indication that the cross-product construction is computable for regular target sets.

The new set of rules *xdpn* can be computed if the set of dpn rules and the set of hedge automaton transitions are finite, as the definition of *xdpn* is not recursive and each LHS depends on only one element of each set. However, as said above, we do not provide executable code here.

In [2], a configuration is represented as a sequence of control and stack symbols, each process starting with a control symbol followed by its stack. For sequences that start with a control symbol, this representation is isomorphic to our representation (cf. Section 6.2.3). As regular sets of configurations are best defined on this list-based semantics, we also show the effectiveness of our construction on the list-based semantics.

This section, especially the proofs of the Theorems, are rather technical. The theorems itself show how to compute the new target configuration and the projection from the computed predecessor set using only operations well-known to preserve regularity (in this case intersection, union, concatenation, star, and substitution) as well as some sets that are obviously regular. However, no formal proof of regularity or effectiveness is given.

8.2.1 Definitions

This function defines the projection operator from the extended to the original configuration:

fun *fp-cl1* **where**
fp-cl1 (*CTRL* (*p*,*s*)) = *CTRL p* |
fp-cl1 (*STACK* γ) = *STACK* γ

This function maps a hedge-automaton state to the regular set of all process configurations labeled with that state. Note that the sets {[*CTRL* (*p*, *s*)] |*p*. *True*} and {[*STACK* γ] |γ. *True*} are obviously regular.

definition *fp-inv-subst2* **where**
fp-inv-subst2 s = *conc* { [*CTRL* (*p*,*s*)] | *p*. *True* } (*star* {[*STACK* γ] | γ. *True*})

The projection operator can be written using substitution, projection (a special form of substitution), and intersection.

The intuitive idea is, that *subst fp-inv-subst2 H0* is the set of all configurations with a hedge-automaton labeling sequence that is accepted by *H0*.

definition *projH-cl* :: *'S list set* ⇒ (*'Q*×*'S*,'T) *cl set* ⇒ (*'Q*,'T) *cl set* **where**
projH-cl H0 Clx = *lang-proj fp-cl1* (*subst fp-inv-subst2 H0* ∩ (*Clx*))

The derivation of the new target set is done by first characterizing all sets of cross-product configurations whose leafs are labeled correctly according to the leaf rules of the hedge automaton. Note that there are only finitely many leaf-rules, hence the union below is over a finite set. Moreover, the language *W* at a leaf rule is regular by default, the operation *map STACK* ' - is a projection and the operation *op* # (*CTRL* (*p*,*s*)) ' - is a concatenation. Hence all the operations below are effective.

definition *xdpnCL-cl* :: (*'S*,*'P*,'T,*'L*) *ha-rule set* ⇒ (*'P*×*'S*,'T) *cl set* **where**
xdpnCL-cl H = *star* (⋃{ *op* # (*CTRL* (*p*,*s*)) ' (*map STACK* ' *W*) |
s p W. *HAR-LEAF s p W* ∈ *H* }
)

Having characterized all configurations that are correctly labeled, one gets the new target set by intersecting them with all configurations that correspond to the old target set:

definition *xdpnC-cl*
:: (*'P*,'T) *cl set* ⇒ (*'S*,*'P*,'T,*'L*) *ha-rule set* ⇒ (*'P*×*'S*,'T) *cl set*
where
xdpnC-cl Cl H = *lang-inv-proj fp-cl1 Cl* ∩ *xdpnCL-cl H*

In order to compute *prehc* Δ (*langh* (*H0*, *H*)) C', we map C' to its corresponding regular set of list-based configurations *c2cl* ' C' and apply the list-based operations for cross-product, predecessor set and projection on it:

definition *prehc-cl*

:: ('Q,'T,'L) *dpn* ⇒ ('S,'Q,'T,'L) *ha* ⇒ ('Q,'T) *cl set* ⇒ ('Q,'T) *cl set*
where
prehc-cl Δ *HA Cl'* =
projH-cl (*fst HA*) ($pre^*{}_{cl}$ (*xdpn* Δ (*snd HA*)) (*xdpnC-cl Cl'* (*snd HA*)))

8.2.2 Theorems

lemma *fp-cl1-map-stack-id*[*simp*]: *map fp-cl1* (*map STACK w*) = *map STACK w*
by (*induct w*) *auto*

lemma *fp-cl1-stack-id*[*simp*]: *fp-cl1 s* = *STACK* γ ⟷ *s*=*STACK* γ
by (*cases s*) *auto*

lemma *fp-cl1-eq-map-stack*[*simp*]:
map fp-cl1 la = *map STACK w* ⟷ *la*=*map STACK w*
apply (*induct w arbitrary*: *la*)
apply *simp*
apply (*case-tac la*)
apply *auto*
done

lemma *star-STACK*[*simplified*,*simp*]:
star {[*STACK* γ] | γ. *True*} = {*map STACK w* | *w*. *True*}
apply *auto*
proof −
case *goal1* **thus** *?case*
apply (*induct rule*: *star.induct*)
apply *auto*
apply (*rule-tac x*=γ#*w* **in** *exI*)
apply *simp*
done
next
case *goal2* **thus** *?case*
apply (*induct w*)
apply (*auto intro*: *star.ConsI*[*of* [*a*], *simplified*, *standard*])
done
qed

lemma *proj-c1-effective*: *c2cl* (*proj-c1 c*) = *map fp-cl1* (*c2cl c*)
by (*induct c*) *auto*

lemma *fp-inv-subst2I*[*intro*!, *simp*]:
CTRL (*p*,*s*)#*map STACK w* ∈ *fp-inv-subst2 s*
proof −
have *1*: [*CTRL* (*p*,*s*)] ∈ { [*CTRL* (*p*,*s*)] | *p*. *True* } **by** *auto*
have *2*: *map STACK w* ∈ (*star* {[*STACK* γ] | γ. *True*}) **by** *auto*
from *concI*[*OF 1 2*] **show** *?thesis* **by** (*auto simp add*: *fp-inv-subst2-def*)

qed

lemma *fp-inv-subst2E*:
⟦*cl*∈*fp-inv-subst2 s*; !!*p w*. *cl*=*CTRL* (*p*,*s*)#*map STACK w* ⟹ *P*⟧ ⟹ *P*
apply (*unfold fp-inv-subst2-def*)
apply (*erule concE*)
apply *fastsimp*
done

Idea of the operation on the original representations of configurations:

lemma *projH-effective'*:
projH H0 Cx = *lang-proj* (λ((*p*,*s*),*w*). (*p*,*w*))
(*lang-inv-proj* (λ((*p*,*s*),*w*). *s*) *H0* ∩ *Cx*)
by (*unfold projH-def lang-proj-def lang-inv-proj-def*) *auto*

Correctness of the list-level operation:

theorem *projH-effective*: *c2cl* ' *projH H0 Cx* = *projH-cl H0* (*c2cl* ' *Cx*)
apply (*unfold projH-effective' lang-proj-def lang-inv-proj-def projH-cl-def*)
apply *auto*
proof −
case (*goal1 cx*) **thus** *?case* **proof** (*induct cx arbitrary*: *Cx H0*)
case *Nil* **thus** *?case*
by (*force simp add*: *subst-def subst-word-def*)
next
case (*Cons πx cx*)
obtain *p s w* **where** [*simp*]: *πx*=((*p*,*s*),*w*) **by** (*cases πx*) *auto*
from *Cons.prems*[*simplified*] **have**
P: *cx*∈{ *cx'* . ((*p*,*s*),*w*)#*cx'*∈*Cx* }   *proj-c2 cx* ∈ { *ss* . *s*#*ss*∈*H0* }
by *auto*
from *Cons.hyps*[*OF P*] **show** *?case*
apply *auto*
proof −
case *goal1*
from *imageI*[*OF goal1*(*3*), *of c2cl*, *simplified*] **have**
CTRL (*p*, *s*) # *map STACK w* @ *c2cl xa* ∈ *c2cl* ' *Cx* **.**
moreover from *goal1*(*2*) **have**
CTRL (*p*, *s*) # *map STACK w* @ *c2cl xa* ∈ *subst fp-inv-subst2 H0*
apply (*auto simp add*: *subst-def subst-word-def*)
apply (*rule-tac x*=*s*#*x* **in** *bexI*)
apply *auto*
apply (*simp only*: *append.simps*(*2*)[*symmetric*])
apply (*rule concI*)
apply *auto*
done
ultimately have
CTRL (*p*, *s*) # *map STACK w* @ *c2cl xa* ∈
subst fp-inv-subst2 H0 ∩ *c2cl* ' *Cx*
by *blast*
from *imageI*[*OF this*, *of map fp-cl1*] **show** *?case* **by** *simp*

```
    qed
  qed
next
  case (goal2 cx) thus ?case
  proof (induct cx arbitrary: Cx H0)
    case Nil thus ?case
      apply (auto simp add: subst-def subst-word-def fp-inv-subst2-def)
      apply (case-tac x)
      apply (auto simp add: conc-def)
      done
  next
    case (Cons πx cx Cx H0)
    obtain p s w where [simp]: πx=((p,s),w) by (cases πx) auto
    from Cons.prems[simplified] have
      CTRL (p, s) # map STACK w @ c2cl cx ∈ subst fp-inv-subst2 H0
      ((p, s), w) # cx ∈ Cx
      by auto
    hence
      P: c2cl cx ∈
          { cl. CTRL (p, s) # map STACK w @ cl ∈ subst fp-inv-subst2 H0 }
      cx ∈ { cx . ((p,s),w)#cx∈Cx }
      by auto
    from P(1) have P': c2cl cx ∈ subst fp-inv-subst2 { ss . s#ss∈H0 }
      apply (auto simp add: subst-def subst-word-def)
      apply (case-tac x)
      apply simp
      apply simp
      apply (erule concE)
      apply auto
      apply (erule fp-inv-subst2E)
      apply auto
      apply (rule-tac x=list in exI)
      apply auto
    proof -
      case (goal1 list b wa) hence wa=w ∧ b=c2cl cx
        apply (cases list)
        apply simp
        apply (cases cx)
        apply simp-all
        apply (erule concE)
        apply auto
        apply (erule fp-inv-subst2E)
        apply simp
        apply (cases cx)
        apply simp-all
        apply (erule fp-inv-subst2E)
        apply simp
        apply (cases cx)
        apply auto
```

```
      done
    thus c2cl cx ∈ conc-list (map fp-inv-subst2 list) using goal1(2) by simp
  qed
  from Cons.hyps[OF P' P(2)] show ?case by force
 qed
qed
```

```
lemma c2cl-empty-rev: [] = c2cl [] by simp

theorem xdpnCL-effective: c2cl ` (xdpnCL H) = xdpnCL-cl H
 apply (unfold c2cl-def-raw xdpnCL-cl-def)
 apply safe
proof -
 case goal1 thus ?case proof (induct c)
   case Nil thus ?case by simp
 next
   case (Cons π c)
   from Cons have
     IHAPP: c2cl-abbrv c ∈
       RegSet.star ( ⋃{op # (CTRL (p, s)) ` map STACK ` W |
                      s p W. HAR-LEAF s p W ∈ H}
                 )
     by auto
   moreover from Cons.prems have
     pc2cl π ∈ ( ⋃{op # (CTRL (p, s)) ` map STACK ` W |
                  s p W. HAR-LEAF s p W ∈ H}
             )
     by (auto) (auto simp add: split-paired-all)
   ultimately show ?case by auto
 qed
next
 case goal2 thus ?case proof (induct rule: star.induct)
   case NilI have []∈xdpnCL H by auto
   thus ?case by (blast intro: c2cl-empty-rev[unfolded c2cl-def])
 next
   case (ConsI πl cl)
   from ConsI.hyps(1) obtain p s w W where
     [simp]: πl = CTRL (p,s)# map STACK w and
       P: w∈W    HAR-LEAF s p W ∈ H
     by auto
   hence
     [simp]: πl=pc2cl ((p,s),w) and
       C1: [((p,s),w)]∈xdpnCL H
     by auto
   from ConsI.hyps(3) obtain c where
     [simp]: cl = c2cl-abbrv c and
```

```
      C2: c∈xdpnCL H
    by auto
  from C1 C2 have ((p,s),w)#c∈xdpnCL H by auto
  moreover have πl@cl = c2cl-abbrv (((p,s),w)#c) by auto
  ultimately show ?case by blast
 qed
qed


lemma inv-proj-c1-effective:
 c2cl ‘ { cx . proj-c1 cx ∈ C } = lang-inv-proj fp-cl1 (c2cl ‘ C)
 apply (unfold c2cl-def-raw)
 apply safe
proof −
 case goal1
 thus ?case proof (induct c arbitrary: C)
  case Nil hence []∈C by auto
  thus ?case
    by (auto simp add: lang-inv-proj-def)
      (blast intro: c2cl-empty-rev[unfolded c2cl-def])
 next
  case (Cons π c)
  then obtain p s w where [simp]: π=((p,s),w) by (cases π) auto
  from Cons.prems have P: proj-c1 c ∈ { c1 . (p,w)#c1 ∈ C } by auto
  from Cons.hyps[OF P] show ?case
   apply (auto simp add: lang-inv-proj-def)
   apply (drule-tac f=c2cl-abbrv in imageI)
   apply simp
   done
 qed
next
 case (goal2 cl) thus ?case
  apply (auto simp add: lang-inv-proj-def)
 proof −
  case goal1 thus ?thesis
  proof (induct c arbitrary: C cl)
   case Nil hence [simp]: cl=[] by (cases cl) auto
   from Nil(2) have []∈{cx. proj-c1 cx ∈ C} by simp
   thus ?case by (drule-tac f=c2cl-abbrv in imageI) simp
  next
   case (Cons π c)
   obtain p w where [simp]: π=(p,w) by (cases π) auto
   from Cons.prems have P1: c∈{ c . π#c ∈ C } by simp
   from Cons.prems(1)[simplified] obtain s cl′ where
     [simp]: cl=CTRL (p,s) # map STACK w @ cl′ and
       P2: map fp-cl1 cl′ = c2cl-abbrv c
     apply −
     apply (elim map-eq-consE map-eq-concE)
     apply (case-tac a)
```

```
      apply fastsimp
      apply simp
      done
    from Cons.hyps[OF P2 P1] show ?case
      apply auto
    proof -
      case (goal1 cx) hence ((p,s),w)#cx ∈ {cx. proj-c1 cx ∈ C} by auto
      thus ?case by (drule-tac f=c2cl-abbrv in imageI) auto
    qed
  qed
 qed
qed

theorem xdpnC-effective: c2cl ‘ (xdpnC C H) = xdpnC-cl (c2cl ‘ C) H
 apply (unfold xdpnC-def xdpnC-cl-def)
 apply (simp only: c2cl-img-Int)
 apply (simp only: inv-proj-c1-effective xdpnCL-effective)
 done


theorem prehc-effective:
 c2cl ‘ prehc Δ (langh (H0,H)) C′ = prehc-cl Δ (H0,H) (c2cl ‘ C′)
 apply (simp add: xdpn-correct prehc-cl-def)
 apply (simp add: xdpnC-effective[symmetric] precl-star-is-pre-star projH-effective)
 done
```

8.3 What Does This Proof Tell You ?

In order to believe that our construction is effective, you have to believe that the RHS of Theorem *prehc-effective* is really effective.

The effectiveness of the pre^* - computation is shown in [2], and we have also an unpublished formal proof of the algorithm presented there. We are planning to adapt this proof to our model definition and the latest Isabelle version in near future, and then publish it.

The effectiveness of the involved automata computations is well-known. In a future version of this formalization, we plan to formalize or adopt an automata library and use it to generate executable code.

end

9 DPNs With Locks

theory *LockSem*
imports *DPN Semantics*
begin

In this theory, we define an extension of DPNs, where synchronization of the processes via a finite set of locks is allowed.

For this purpose, we assume that the rules are labeled with lock operations.

9.1 Model

— If a label has either no effect on locks, we allow it to be labeled by some other generic type $'L$. Otherwise, the label indicates either the acquisition or the release of a lock:

datatype $('L,'X)$ *lockstep* = *LNone* $'L$ | *LAcq* $'X$ | *LRel* $'X$

— Abbreviation for the datatype of a DPN with locks:

types $('P,'\Gamma,'L,'X)$ *ldpn* = $('P,'\Gamma,('L,'X)$ *lockstep*$)$ *dpn*

We encode DPNs with locks in a locale.

To save some case distinctions in proofs, we assume that only non-spawning rules are labeled with lock operations.

locale *LDPN* = *DPN* +
  constrains
    Δ :: $('P,'\Gamma,'L,'X$::*finite*$)$ *ldpn*
  assumes
    spawn-no-locks: $[\![(p,\gamma \hookrightarrow_a ps,ws \sharp p',w) \in \Delta;\ !!l.\ a{=}LNone\ l \Longrightarrow P]\!] \Longrightarrow P$
begin
  lemma *snl-simps*[*simp*, *intro*!]:
    $(p,\gamma \hookrightarrow_{LAcq\ x} ps,ws \sharp p',w) \notin \Delta$
    $(p,\gamma \hookrightarrow_{LRel\ x} ps,ws \sharp p',w) \notin \Delta$
    by (*auto elim*: *spawn-no-locks*)

  lemma *X-finite*: *finite* (*UNIV*::$'X$ *set*) **by** *simp*
end

9.2 Interleaving Semantics

The following predicate models the step-relation on the set of allocated locks:

inductive *lock-valid* :: $'X$ *set* $\Rightarrow$ $('L,'X)$ *lockstep* $\Rightarrow$ $'X$ *set* $\Rightarrow$ *bool* **where**
  — A *LNone*-step does not change the set of allocated locks:
  lv-none: *lock-valid* X (*LNone* l) X |
  — A *LAcq*-step adds the acquired lock to the set of locks. It is only executable if the lock was not allocated before:
  lv-acquire: *lock-valid* $(X-\{x\})$ (*LAcq* x) (*insert* x X) |
  — A *LRel*-step removes the released lock from the set of locks. It is only executable if the lock was allocated before:
  lv-release: *lock-valid* (*insert* x X) (*LRel* x) $(X-\{x\})$

lemma *lock-valid-simps*[*simp*]:
  lock-valid X (*LNone* l) $X' \longleftrightarrow X{=}X'$
  lock-valid X (*LAcq* x) $X' \longleftrightarrow X'{=}insert\ x\ X \wedge x{\notin}X$
  lock-valid X (*LRel* x) $X' \longleftrightarrow X{=}insert\ x\ X' \wedge x{\notin}X'$
  apply (*auto elim*: *lock-valid.cases intro*: *lock-valid.intros*)

apply (*subst set-minus-singleton-eq*[*symmetric*], *assumption*)
apply (*rule lock-valid.intros*)
apply (*subst* (*3*) *set-minus-singleton-eq*[*symmetric*], *assumption*)
apply (*rule lock-valid.intros*)
done

Configurations of the lock-sensitive step-relation consists of the list of processes and the set of currently acquired locks. Note that, at this point in the formalization, we do not make any assumptions on which process may release a lock, or on well-nestedness of locks.

That is, we allow a process releasing a lock that it has not acquired before, or locks being used in non-well-nestedness fashion.

However, in Section 10, we formalize such assumptions.

The lock-sensitive step-relation is the intersection of the original step-relation and the step-relation on allocated locks.

definition *ldpntr*
:: (*'P*,'T,*'L*,*'X*) *ldpn* ⇒ ((*'P*,'T) *conf* × *'X set*, (*'L*,*'X*) *lockstep*) *LTS*
where
ldpntr Δ = { ((*c*,*X*),*l*,(*c'*,*X'*)) . (*c*,*l*,*c'*) ∈ *dpntr* Δ ∧ *lock-valid X l X'*}

abbreviation *ldpntrc* Δ == *trcl* (*ldpntr* Δ)

lemma *ldpntr-subset*: ((*c*,*X*),*w*,(*c'*,*X'*))∈*ldpntr* Δ ⟹ (*c*,*w*,*c'*)∈*dpntr* Δ
by (*auto simp add*: *ldpntr-def*)
lemma *ldpntrc-subset*: ((*c*,*X*),*w*,(*c'*,*X'*))∈*ldpntrc* Δ ⟹ (*c*,*w*,*c'*)∈*dpntrc* Δ
by (*induct rule*: *trcl-pair-induct*) (*auto dest*: *ldpntr-subset*)

9.3 Tree Semantics

For the tree semantics, we only need to redefine the scheduler, such that it keeps track of the allocated locks.

— Abbreviation for type of execution trees and hedges with locks:
types (*'Q*,'T,*'L*,*'X*) *lex-tree* = (*'Q*,'T,(*'L*,*'X*) *lockstep*) *ex-tree*
types (*'Q*,'T,*'L*,*'X*) *lex-hedge* = (*'Q*,'T,(*'L*,*'X*) *lockstep*) *ex-hedge*

— The definition of the lock-sensitive scheduler is straightforward:
inductive *lsched*
:: (*'Q*,'T,*'L*,*'X*) *lex-hedge* ⇒ *'X set* ⇒ (*'L*,*'X*) *lockstep list* ⇒ *bool*
where
lsched-final: *final h* ⟹ *lsched h X* [] |
lsched-nospawn:
⟦*lsched* (*h1*@*t*#*h2*) *X'* *w*; *lock-valid X l X'*⟧ ⟹
lsched (*h1*@(*NNOSPAWN l t*)#*h2*) *X* (*l*#*w*) |
lsched-spawn:
⟦*lsched* (*h1*@*ts*#*t*#*h2*) *X'* *w*; *lock-valid X l X'*⟧ ⟹
lsched (*h1*@(*NSPAWN l ts t*)#*h2*) *X* (*l*#*w*)

— Obviously, a lock-sensitive schedule is also a schedule of the original scheduler:

lemma *lsched-is-sched*: *lsched h X ll* $\Longrightarrow$ *sched h ll*
  by (*induct rule*: *lsched.induct*) (*auto intro*: *sched.intros*)

9.4 Equivalence of Interleaving and Tree Semantics

— Straightforward adoption of proof of *sched-correct1*
lemma *lsched-correct1*:
  $((c,X),ll,(c',X'))\in ldpntrc\ \Delta \Longrightarrow \exists h.\ hsem\ \Delta\ c\ h\ c' \wedge lsched\ h\ X\ ll$
proof (*induct rule*: *trcl-pair-induct*)
  case (*empty c X*)
  thus *?case*
    by (*induct c*)
      (*fastsimp intro*!: *hsem-cons-single lsched-final elim*: *lsched.cases*)+
next
  case (*cons c X l ch Xh ll c' X'*)
  from *cons.hyps*(*3*) **obtain** *h* **where**
    IHAPP: *hsem* Δ *ch h c'*   *lsched h Xh ll*
    by *blast*
  from *cons.hyps*(*1*) **have**
    $(c,l,ch)\in dpntr\ \Delta$ **and**
    LV: *lock-valid X l Xh*
    by (*unfold ldpntr-def*) *auto*
  thus *?case* **proof** (*cases*)
    case (*dpntr-no-spawn p* γ *la p' w c1 r c2*)
    hence
      C-simp[*simp*]: $c = c1\ @\ (p,\ \gamma\ \#\ r)\ \#\ c2$
                $ch = c1\ @\ (p',\ w\ @\ r)\ \#\ c2$ **and**
      C: $(p,\gamma \hookrightarrow_l p',w) \in \Delta$
      by *auto*
    from *hsem-lel*[*OF IHAPP*(*1*)[*simplified*]] **obtain** *h1 t h2 c1' ct' c2'* **where**
      [*simp*]: $h = h1\ @\ t\ \#\ h2$ $c' = c1'\ @\ ct'\ @\ c2'$ **and**
      HSPLIT: *hsem* Δ *c1 h1 c1'*   *tsem* Δ $(p',\ w\ @\ r)$ *t ct'*
        hsem Δ *c2 h2 c2'*
      .
    from *tsem-nospawn*[*OF C HSPLIT*(*2*)] **have**
      ST: *tsem* Δ $(p,\gamma\#r)$ (*NNOSPAWN l t*) *ct'* **.**
    from *hsem-conc-lel*[*OF HSPLIT*(*1*) *ST HSPLIT*(*3*)] **have**
      hsem Δ *c* (*h1* @ *NNOSPAWN l t* # *h2*) *c'*
      by *simp*
    moreover from *lsched-nospawn*[*OF IHAPP*(*2*)[*simplified*] *LV*] **have**
      lsched (*h1* @ *NNOSPAWN l t* # *h2*) *X* (*l#ll*) **.**
    ultimately show *?thesis* **by** *blast*
  next
    case (*dpntr-spawn p* γ *la ps ws p' w c1 r c2*)
    hence
      [*simp*]: $c = c1\ @\ (p,\ \gamma\ \#\ r)\ \#\ c2$
        $ch = c1\ @\ (ps,\ ws)\ \#\ (p',\ w\ @\ r)\ \#\ c2$ **and**

C: (*p*,*γ* $\hookrightarrow_l$ *ps*,*ws* ♯ *p*′,*w*) ∈ Δ
by *auto*
from *IHAPP*(*1*)[*simplified*] **obtain** *h1 ts t h2 c1*′ *cs*′ *ct*′ *c2*′ **where**
[*simp*]: *h* = *h1* @ *ts* # *t* # *h2* *c*′ = *c1*′ @ *cs*′ @ *ct*′ @ *c2*′ **and**
HSPLIT: *hsem* Δ *c1 h1 c1*′ *tsem* Δ (*ps*,*ws*) *ts cs*′
tsem Δ (*p*′, *w* @ *r*) *t ct*′ *hsem* Δ *c2 h2 c2*′
by (*fastsimp elim*: *hsem-split hsem-split-single*)
from *tsem-spawn*[*OF C HSPLIT*(*2*,*3*)] **have**
ST: *tsem* Δ (*p*,*γ*#*r*) (*NSPAWN l ts t*) (*cs*′@*ct*′) **.**
from *hsem-conc-lel*[*OF HSPLIT*(*1*) *ST HSPLIT*(*4*)] **have**
hsem Δ *c* (*h1* @ *NSPAWN l ts t* # *h2*) *c*′
by *simp*
moreover from *lsched-spawn*[*OF IHAPP*(*2*)[*simplified*] *LV*] **have**
lsched (*h1* @ *NSPAWN l ts t* # *h2*) *X* (*l*#*ll*) **.**
ultimately show *?thesis* **by** *blast*
qed
qed

— Straightforward adoption of proof of *sched-correct2*
lemma *lsched-correct2*:
⟦ *lsched h X ll*; *hsem* Δ *c h c*′ ⟧ ⟹ ∃ *X*′. ((*c*,*X*),*ll*,(*c*′,*X*′))∈*ldpntrc* Δ
proof (*induct h X ll arbitrary*: *c c*′ *rule*: *lsched.induct*)
case (*lsched-final h X c c*′) **thus** *?case* **by** (*auto dest*: *final-hsem-nostep*)
next
case (*lsched-nospawn h1 t h2 Xh ll X l c c*′)
from *hsem-lel-h*[*OF lsched-nospawn.prems*] **obtain** *c1 pγr c2 c1*′ *ct*′ *c2*′ **where**
[*simp*]: *c* = *c1* @ *pγr* # *c2* *c*′ = *c1*′ @ *ct*′ @ *c2*′ **and**
SPLIT: *hsem* Δ *c1 h1 c1*′ *tsem* Δ *pγr* (*NNOSPAWN l t*) *ct*′
hsem Δ *c2 h2 c2*′
.
from *SPLIT*(*2*) **obtain** *p γ r p*′ *w* **where**
[*simp*]: *pγr*=(*p*,*γ*#*r*) **and**
ST: (*p*,*γ* $\hookrightarrow_l$ *p*′,*w*)∈Δ *tsem* Δ (*p*′,*w*@*r*) *t ct*′
by (*erule-tac tsem.cases*) *fastsimp*+
from *dpntr-no-spawn*[*OF ST*(*1*)] **have** (*c*,*l*,*c1* @ (*p*′, *w* @ *r*) # *c2*)∈*dpntr* Δ
by *auto*
with *lsched-nospawn.hyps*(*3*) **have**
((*c*,*X*),*l*,(*c1* @ (*p*′, *w* @ *r*) # *c2*,*Xh*))∈*ldpntr* Δ
by (*unfold ldpntr-def*) *auto*
also
from *lsched-nospawn.hyps*(*2*)[*OF hsem-conc-lel*[*OF SPLIT*(*1*) *ST*(*2*) *SPLIT*(*3*)]]

obtain *X*′ **where**
SST: ((*c1* @ (*p*′, *w* @ *r*) # *c2*,*Xh*), *ll*, (*c1*′ @ *ct*′ @ *c2*′,*X*′)) ∈ *ldpntrc* Δ
by *blast*
finally show *?case* **by** *auto*
next
case (*lsched-spawn h1 ts t h2 Xh ll X l c c*′)
from *hsem-lel-h*[*OF lsched-spawn.prems*] **obtain** *c1 pγr c2 c1*′ *ct*′ *c2*′ **where**

[*simp*]: $c = c1 @ p\gamma r \# c2 \quad c' = c1' @ ct' @ c2'$ **and**
SPLIT: $hsem\ \Delta\ c1\ h1\ c1' \quad tsem\ \Delta\ p\gamma r\ (NSPAWN\ l\ ts\ t)\ ct'$
$hsem\ \Delta\ c2\ h2\ c2'$
.
from *SPLIT*(*2*) **obtain** $p\ \gamma\ r\ ps\ ws\ p'\ w\ cts'\ ctt'$ **where**
[*simp*]: $p\gamma r=(p,\gamma\#r) \quad ct'=cts'@ctt'$ **and**
ST: $(p,\gamma \hookrightarrow_l ps,ws \sharp p',w)\in\Delta \quad tsem\ \Delta\ (ps,ws)\ ts\ cts'$
$tsem\ \Delta\ (p',w@r)\ t\ ctt'$
by (*erule-tac tsem.cases*) *fastsimp+*
from *dpntr-spawn*[*OF ST*(*1*)] **have**
$(c,l,c1 @ (ps,ws) \# (p', w @ r) \# c2)\in dpntr\ \Delta$
by *auto*
with *lsched-spawn.hyps*(*3*) **have**
$((c,X),l,(c1 @ (ps,ws)\#(p', w @ r) \# c2,Xh))\in ldpntr\ \Delta$
by (*unfold ldpntr-def*) *auto*
also from
lsched-spawn.hyps(*2*)[*OF hsem-conc-leel*[*OF SPLIT*(*1*) *ST*(*2*,*3*) *SPLIT*(*3*)]]
obtain X' **where**
SST: $((c1 @ (ps,ws) \# (p', w @ r) \# c2,Xh),\ ll,\ (c',X')) \in ldpntrc\ \Delta$
by *fastsimp*
finally show *?case* **by** *auto*
qed

theorem *lsched-correct*:
$(\exists X'.\ ((c,X),ll,(c',X'))\in ldpntrc\ \Delta) \longleftrightarrow (\exists h.\ hsem\ \Delta\ c\ h\ c' \wedge lsched\ h\ X\ ll)$
by (*auto intro*: *lsched-correct1 lsched-correct2*)

end

10 Well-Nestedness of Locks

theory *WellNested*
imports *DPN Semantics LockSem*
begin

Well-nestedness of locks is the property that no locks are re-acquired by the same process and a released locks is always the last one that was acquired and not yet released by the releasing process. Usually, these two properties are called non-reentrance and well-nestedness.

In this theory, we formulate a sufficient condition for well-nestedness, that regards every possible lock-insensitive run of the DPN from some initial configuration. We then define an equivalent condition on execution hedges.

Note that our condition may rule out DPNs where some non-well-nested runs are blocked by deadlocks or other lock-induced effects. However, important classes of programs, in particular programs that use locks in a block-structured way (like synchronized-blocks in Java), always satisfy our condition.

Further work required at this point is to formalize a program analysis or some sufficient conditions (like block-structured lock-acquisition [monitors]) for well-nestedness. We would then be able to prove some non-trivial DPNs to have well-nested configurations, thus giving a stronger indication that the well-nestedness assumption is correct. In the current state, we have no formal proof that the well-nestedness assumption is correct, i.e. an uncorrect well-nestedness assumption, e.g. a too strict one, would affect the scope of all our proofs that use this assumption. In the worst case, there would be no well-nested DPNs at all (or only trivial ones).

10.1 Well-Nestedness Condition on Paths

We first define the set of all paths that may occur from a process. We collect local paths and environment paths.

ppairs (q,w) *False* l means that there is a local path l from process (q,w).

ppairs (q,w) *True* l means that we can reach a spawn step from process (q,w) that spawns a process having path "l".

inductive *ppairs*
  :: $('P,'\Gamma,'L,'X)$ *ldpn* $\Rightarrow$ $('P,'\Gamma)$ *pconf* $\Rightarrow$ *bool* $\Rightarrow$ $('L,'X)$ *lockstep list* $\Rightarrow$ *bool*
  for Δ **where**
  ppairs-empty: *ppairs* Δ (q,w) *False* [] |
  ppairs-prepend1:
  $[\![(q,\gamma \hookrightarrow_a q',w) \in \Delta;$ *ppairs* Δ $(q',w@r)$ *False* l $]\!] \Longrightarrow$
    ppairs Δ $(q,\gamma\#r)$ *False* $(a\#l)$ |
  ppairs-mvenv1:
  $[\![(q,\gamma \hookrightarrow_a q',w) \in \Delta;$ *ppairs* Δ $(q',w@r)$ *True* l $]\!] \Longrightarrow$
    ppairs Δ $(q,\gamma\#r)$ *True* l |
  ppairs-prepend2:
  $[\![(q,\gamma \hookrightarrow_a qs,ws \sharp q',w) \in \Delta;$ *ppairs* Δ $(q',w@r)$ *False* l $]\!] \Longrightarrow$
    ppairs Δ $(q,\gamma\#r)$ *False* $(a\#l)$ |
  ppairs-mvenv2: $[\![(q,\gamma \hookrightarrow_a qs,ws \sharp q',w) \in \Delta;$ *ppairs* Δ $(q',w@r)$ *True* l $]\!] \Longrightarrow$
    ppairs Δ $(q,\gamma\#r)$ *True* l |
  ppairs-genenv: $[\![(q,\gamma \hookrightarrow_a qs,ws \sharp q',w) \in \Delta;$ *ppairs* Δ (qs,ws) x $l]\!] \Longrightarrow$
    ppairs Δ $(q,\gamma\#r)$ *True* l

This function checks whether a path is well-nested by using a lock stack.

fun *wn-p* :: $('L,'X)$ *lockstep list* $\Rightarrow$ $'X$ *list* $\Rightarrow$ *bool* **where**
  wn-p [] μ = *distinct* μ |
  wn-p (*LAcq* $x\#l$) $\mu \longleftrightarrow$ *wn-p* l $(x\#\mu)$ |
  wn-p (*LRel* $x\#l$) $\mu \longleftrightarrow (\exists \mu'. \mu=x\#\mu' \wedge x \notin set\ \mu' \wedge$ *wn-p* $l\ \mu')$ |
  wn-p $(\text{-}\#l)$ $\mu \longleftrightarrow$ *wn-p* l μ

A process π is defined to be well-nested w.r.t. some initial lock stack μ if all reachable path – local paths and environment paths – are well-nested.

definition *wn-π* Δ π μ ==
  case π *of* $(p,w) \Rightarrow$
    $\forall\, l.$ (*ppairs* Δ (p,w) *False* $l \longrightarrow$ *wn-p* l μ) $\wedge$
      (*ppairs* Δ (p,w) *True* $l \longrightarrow$ *wn-p* l [])

Introduction and elimination rules for *wn-π*

lemma *wn-πI*:
  $[\![$
    !!*l*. *ppairs* Δ (q,w) *False* $l \Longrightarrow$ *wn-p* l μ;
    !!*l*. *ppairs* Δ (q,w) *True* $l \Longrightarrow$ *wn-p* l []
  $]\!] \Longrightarrow$ *wn-π* Δ (q,w) μ
  by (*unfold wn-π-def*) *auto*

lemma *wn-πE*:
  $[\![$*wn-π* Δ (q,w) μ;
    $[\![$
      !!*l*. *ppairs* Δ (q,w) *False* $l \Longrightarrow$ *wn-p* l μ;
      !!*l*. *ppairs* Δ (q,w) *True* $l \Longrightarrow$ *wn-p* l []
    $]\!] \Longrightarrow P$
  $]\!] \Longrightarrow P$
  by (*unfold wn-π-def*) *auto*

We have set up the definitions such that well-nestedness w.r.t a lock stack implies distinctness of this lock stack.

lemma *wn-p-distinct*: *wn-p* l $\mu \Longrightarrow$ *distinct* μ
  by (*induct rule*: *wn-p.induct*) *auto*

lemma *wn-π-distinct*: *wn-π* Δ π $\mu \Longrightarrow$ *distinct* μ
  using *ppairs.intros*(*1*)
  apply (*unfold wn-π-def*)
  apply (*simp split*: *prod.split-asm*)
  apply (*rule wn-p-distinct*)
  apply (*fast*)
  done

Well-nestedness is preserved by steps:

lemma *wn-π-none*:
  $[\![$ $(q,\gamma \hookrightarrow_{(LNone\ l)} q',w) \in \Delta$; *wn-π* Δ $(q,\gamma\#r)$ μ $]\!] \Longrightarrow$ *wn-π* Δ $(q',w@r)$ μ
  by (*unfold wn-π-def*) (*auto intro*: *ppairs.intros*)
lemma (**in** *LDPN*) *wn-π-spawn1*:
  $[\![$ $(q,\gamma \hookrightarrow_{a} qs,ws \sharp q',w) \in \Delta$; *wn-π* Δ $(q,\gamma\#r)$ μ $]\!] \Longrightarrow$ *wn-π* Δ $(q',w@r)$ μ
  by (*cases a*, *unfold wn-π-def*) (*auto intro*: *ppairs.intros*)
lemma *wn-π-spawn2*:
  $[\![$ $(q,\gamma \hookrightarrow_{a} qs,ws \sharp q',w) \in \Delta$; *wn-π* Δ $(q,\gamma\#r)$ μ $]\!] \Longrightarrow$ *wn-π* Δ (qs,ws) []
  by (*cases a*, *unfold wn-π-def*) (*auto intro*: *ppairs.intros*)
lemma *wn-π-acq*:
  $[\![$ $(q,\gamma \hookrightarrow_{LAcq\ x} q',w) \in \Delta$; *wn-π* Δ $(q,\gamma\#r)$ μ $]\!] \Longrightarrow$ *wn-π* Δ $(q',w@r)$ $(x\#\mu)$
  by (*unfold wn-π-def*) (*auto intro*: *ppairs.intros*)
lemma *wn-π-rel*:

assumes *A*: $(q,\gamma \hookrightarrow_{LRel\ x}\ q',w) \in \Delta$ $\quad$ *wn-π* Δ $(q,\gamma\#r)$ μ **and**
C: $!!\mu'$. $[\![\mu=x\#\mu'; x \notin set\ \mu'; wn\text{-}\pi\ \Delta\ (q',w@r)\ \mu'\ ]\!] \Longrightarrow P$
shows *P*
proof −
from *wn-πE*[*OF A*(*2*)] **have** *X*: $!!l.$ *ppairs* Δ $(q,\ \gamma\ \#\ r)$ *False* $l \Longrightarrow$ *wn-p* $l\ \mu$
by *blast*
from *X*[*OF ppairs-prepend1*[*OF A*(*1*) *ppairs-empty*],*simplified*] **obtain** μ' **where**
[*simp*]: $\mu=x\#\mu'$ $\quad x \notin set\ \mu'$
by *blast*
moreover from *A* **have** *wn-π* Δ $(q',w@r)$ μ'
by (*unfold wn-π-def*) (*auto intro*: *ppairs.intros*)
ultimately show *P* **by** (*rule C*)
qed

lemma (**in** *LDPN*) *wn-π-preserve*:
$[\![$ $(q,\gamma \hookrightarrow_l q',w) \in \Delta$; *wn-π* Δ $(q,\gamma\#r)$ *xs*;
$!!xs'$. *wn-π* Δ $(q',w@r)$ $xs' \Longrightarrow P$
$]\!] \Longrightarrow P$

$[\![$ $(q,\gamma \hookrightarrow_l qs,ws \sharp q',w) \in \Delta$; *wn-π* Δ $(q,\gamma\#r)$ *xs*;
$!!xs'$. $[\![$ *wn-π* Δ $(q',w@r)$ xs'; *wn-π* Δ (qs,ws) [] $]\!] \Longrightarrow P$
$]\!] \Longrightarrow P$
apply (*cases l*)
apply (*auto dest*!: *wn-π-none wn-π-acq elim*!: *wn-π-rel*) [*3*]
apply (*frule* (*1*) *wn-π-spawn1*)
apply (*auto dest*!: *wn-π-spawn2*)
done

10.2 Well-Nestedness of Configurations

The locks of a list of lock stacks

abbreviation *locks-μ* :: *'X list list* ⇒ *'X set* **where**
locks-μ μ == *list-collect-set set* μ

A configuration $c=\pi_1\ldots\pi_n$ is well-nested w.r.t. a list $\mu=s_1\ldots s_n$ of lock stacks (*wn-h h* μ), iff all π_i are well-nested w.r.t. stack s_i and μ is consistent, i.e. contains no duplicate locks.

fun *wn-c* **where**
wn-c Δ [] [] $\longleftrightarrow$ *True* |
wn-c Δ $(\pi\#c)$ $(xs\#\mu)$ $\longleftrightarrow$
wn-c Δ c μ $\wedge$ *set xs* $\cap$ *locks-μ* $\mu = \{\}$ $\wedge$ *wn-π* Δ π *xs* |
wn-c Δ - - $\longleftrightarrow$ *False*

10.2.1 Auxilliary Lemmas about *wn-c*

lemma *wn-c-simps*[*simp*]:
wn-c Δ c [] $\longleftrightarrow$ c=[]
wn-c Δ [] μ $\longleftrightarrow$ μ=[]
apply (*induct c*)

```
  apply auto
  apply (induct μ)
  apply auto
  done

lemma wn-c-length: wn-c Δ c μ ⟹ length c = length μ
  by (induct Δ c μ rule: wn-c.induct) auto

lemma wn-c-prepend-c:
  ⟦ wn-c Δ (π#c) μ;
    !!xs μ'. ⟦ μ=xs#μ'; wn-c Δ c μ';
              set xs ∩ locks-μ μ' = {}; wn-π Δ π xs
            ⟧ ⟹ P
  ⟧ ⟹ P
  by (induct μ arbitrary: π c) fastsimp+

lemma wn-c-prepend-μ:
  ⟦ wn-c Δ c (xs#μ);
    !!π c'. ⟦ c=π#c'; wn-c Δ c' μ;
             set xs ∩ locks-μ μ = {}; wn-π Δ π xs
           ⟧ ⟹ P
  ⟧ ⟹ P
  by (induct c arbitrary: μ) auto

lemma wn-c-append-c-helper:
  assumes
    A: wn-c Δ c μ    c1@c2=c and
    C: !!μ1 μ2. ⟦ μ=μ1@μ2 ∧ wn-c Δ c1 μ1 ∧ wn-c Δ c2 μ2 ∧
                 locks-μ μ1 ∩ locks-μ μ2 = {}
               ⟧ ⟹ P
  shows P
  using A C
  apply (induct Δ c μ arbitrary: c1 c2 P rule: wn-c.induct)
  apply auto
  apply fastsimp
  apply (case-tac c1)
  apply fastsimp
  apply auto
proof -
  case goal1
  show P
    apply (rule goal1(1))
    apply simp
    apply (rule-tac ?μ1.0 = xs#μ1 and ?μ2.0 = μ2 in goal1(2))
    apply (insert goal1(3-))
    apply auto
    done
qed
```

lemma *wn-c-append-c*:
⟦*wn-c* Δ (*c1*@*c2*) *μ*;
!!*μ1 μ2*. ⟦ *μ*=*μ1*@*μ2* ∧ *wn-c* Δ *c1 μ1* ∧ *wn-c* Δ *c2 μ2* ∧
locks-μ μ1 ∩ *locks-μ μ2* = {}⟧ ⟹ *P*
⟧ ⟹ *P*
using *wn-c-append-c-helper*
by *blast*

lemma *wn-c-append-μ-helper*:
assumes
A: *wn-c* Δ *c μ*   *μ1*@*μ2*=*μ* **and**
C: !!*c1 c2*. ⟦ *c*=*c1*@*c2* ∧ *wn-c* Δ *c1 μ1* ∧ *wn-c* Δ *c2 μ2* ∧
locks-μ μ1 ∩ *locks-μ μ2* = {}⟧ ⟹ *P*
shows *P*
using *A C*
apply (*induct* Δ *c μ arbitrary*: *μ1 μ2 P rule*: *wn-c.induct*)
apply *auto*
apply (*case-tac μ1*)
apply *fastsimp*
apply *auto*
proof −
case *goal1*
show *P*
apply (*rule goal1*(*1*))
apply *simp*
apply (*rule-tac ?c1.0* = (*a*,*b*)#*c1* **and** *?c2.0* = *c2* **in** *goal1*(*2*))
apply (*insert goal1*(*3*−))
apply *auto*
done
qed

lemma *wn-c-append-μ*:
⟦*wn-c* Δ *c* (*μ1*@*μ2*);
!!*c1 c2*. ⟦ *c*=*c1*@*c2* ∧ *wn-c* Δ *c1 μ1* ∧ *wn-c* Δ *c2 μ2* ∧
locks-μ μ1 ∩ *locks-μ μ2* = {}⟧ ⟹ *P*
⟧ ⟹ *P*
using *wn-c-append-μ-helper*
by *blast*

lemma *wn-c-appendI*:
⟦*wn-c* Δ *c1 μ1*; *wn-c* Δ *c2 μ2*; *locks-μ μ1* ∩ *locks-μ μ2* = {}⟧ ⟹
wn-c Δ (*c1*@*c2*) (*μ1*@*μ2*)
by (*induct* Δ *c1 μ1 arbitrary*: *c2 μ2 rule*: *wn-c.induct*) *auto*

lemma *wn-c-prependI*:
⟦*wn-π* Δ *π xs*; *wn-c* Δ *c μ*; *set xs* ∩ *locks-μ μ* = {}⟧ ⟹ *wn-c* Δ (*π*#*c*) (*xs*#*μ*)
by *auto*

lemma *wn-c-singlecE*: ⟦*wn-c* Δ [*π*] *μ*; !!*xs*. ⟦*μ*=[*xs*]; *wn-π* Δ *π xs*⟧ ⟹ *P* ⟧ ⟹ *P*

by (*cases* μ) *auto*

lemma *wn-c-split-aux*:
  assumes
    WN: *wn-c* Δ *c* μ **and**
    HFMT[*simp*]: *c*=*c1*@π#*c2* **and**
    C: !!μ*1 xs* μ*2*. ⟦ μ=μ*1*@*xs*#μ*2*; *wn*-π Δ π *xs*; *wn-c* Δ *c1* μ*1*; *wn-c* Δ *c2* μ*2*;
        locks-μ μ*1* ∩ *set xs* = {}; *locks*-μ μ*1* ∩ *locks*-μ μ*2* = {};
        set xs ∩ *locks*-μ μ*2* = {}
      ⟧ ⟹ *P*
  shows *P*
  using *WN*[*simplified*]
  apply (*elim wn-c-append-c wn-c-prepend-c conjE*)
  apply (*rule C*)
  apply (*auto*)
  done

Well-nestedness of configurations is preserved by lock-sensitive steps.

lemma (**in** *LDPN*) *wnc-preserve-singlestep*:
  assumes
    A: ((*c*,*locks*-μ μ),*l*,(*c'*,*X'*))∈*ldpntr* Δ *wn-c* Δ *c* μ **and**
    C: !!μ'. ⟦*X'*=*locks*-μ μ'; *wn-c* Δ *c'* μ'⟧ ⟹ *P*
  shows *P*
proof −
  from *A* **have** *TR*: (*c*,*l*,*c'*)∈*dpntr* Δ **and** *LV*: *lock-valid* (*locks*-μ μ) *l X'*
    by (*auto simp add*: *ldpntr-def*)
  from *TR* **show** *?thesis* **proof** (*cases rule*: *dpntr.cases*)
    case (*dpntr-no-spawn p* γ - *p'* *w c1 r c2*)
    hence
      FMT[*simp*]: *c* = *c1* @ (*p*, γ # *r*) # *c2*   *c'* = *c1* @ (*p'*, *w* @ *r*) # *c2* **and**
      R: (*p*,γ $\hookrightarrow_l$ *p'*,*w*) ∈ Δ
      by *auto*
    from *wn-c-split-aux*[*OF A*(*2*) *FMT*(*1*)] **obtain** μ*1 xs* μ*2* **where**
      [*simp*]: μ = μ*1* @ *xs* # μ*2* **and**
      WNS: *wn*-π Δ (*p*, γ # *r*) *xs*   *wn-c* Δ *c1* μ*1*   *wn-c* Δ *c2* μ*2* **and**
      DISJ: *locks*-μ μ*1* ∩ *set xs* = {}   *locks*-μ μ*1* ∩ *locks*-μ μ*2* = {}
        set xs ∩ *locks*-μ μ*2* = {}
      .
    obtain *xs'* **where**
      wn-π Δ (*p'*,*w*@*r*) *xs'*   *X'*=(*locks*-μ (μ*1*@*xs'*#μ*2*))
      locks-μ μ*1* ∩ *set xs'* = {}   *set xs'* ∩ *locks*-μ μ*2* = {}
    proof (*cases l*)
      case *LNone*[*simp*]
      from *that*[*OF wn*-π-*none*[*OF R*[*simplified*] *WNS*(*1*)]] *DISJ LV* **show** *?thesis*
        by *simp*
    next
      case (*LAcq x*)[*simp*]
      from *that*[*OF wn*-π-*acq*[*OF R*[*simplified*] *WNS*(*1*)]] *LV DISJ* **show** *?thesis*
        by *simp*

next
case (*LRel x*)[*simp*]
from *wn-π-rel*[*OF R*[*simplified*] *WNS*(*1*)] **obtain** *xs′* **where**
[*simp*]: *xs*=*x*#*xs′* **and**
1: *x*∉*set xs′* **and**
2: *wn-π* Δ (*p′*,*w*@*r*) *xs′*
.
from *1 LV DISJ* **show** *?thesis* **by** (*rule-tac that*[*OF 2*]) *auto*
qed
with *WNS*(*2*,*3*) *DISJ*(*2*) **show** *P*
by (*rule-tac μ′*=*μ1*@*xs′*#*μ2* **in** *C*) (*auto intro*!: *wn-c-appendI wn-c-prependI*)
next
case (*dpntr-spawn p γ - ps ws p′ w c1 r c2*)
hence
FMT[*simp*]: *c* = *c1* @ (*p*, *γ* # *r*) # *c2*
c′ = *c1* @ (*ps*, *ws*) # (*p′*, *w* @ *r*) # *c2* **and**
R: (*p*,*γ* ↪$_l$ *ps*,*ws* ♯ *p′*,*w*) ∈ Δ
by *auto*
from *R* **obtain** *ll* **where** [*simp*]: *l*=*LNone ll* **by** (*cases l*) *auto*
from *wn-c-split-aux*[*OF A*(*2*) *FMT*(*1*)] **obtain** *μ1 xs μ2* **where**
[*simp*]: *μ* = *μ1* @ *xs* # *μ2* **and**
WNS: *wn-π* Δ (*p*, *γ* # *r*) *xs*   *wn-c* Δ *c1 μ1*   *wn-c* Δ *c2 μ2* **and**
DISJ: *locks-μ μ1* ∩ *set xs* = {}   *locks-μ μ1* ∩ *locks-μ μ2* = {}
set xs ∩ *locks-μ μ2* = {}
.
from *wn-π-spawn1*[*OF R WNS*(*1*)] *wn-π-spawn2*[*OF R WNS*(*1*)]
WNS(*2*,*3*) *DISJ*
have *wn-c* Δ *c′* (*μ1*@[]#*xs*#*μ2*)
by (*auto intro*!: *wn-c-appendI wn-c-prependI*)
thus *?thesis* **using** *LV* **by** (*rule-tac μ′*=*μ1*@[]#*xs*#*μ2* **in** *C*) *auto*
qed
qed

lemma (**in** *LDPN*) *wnc-preserve*:
assumes *A*: ((*c*,*locks-μ μ*),*ll*,(*c′*,*X′*))∈*ldpntrc* Δ   *wn-c* Δ *c μ* **and**
C: !!*μ′*. ⟦*X′*=*locks-μ μ′*; *wn-c* Δ *c′ μ′*⟧ ⟹ *P*
shows *P*
proof −
{
fix *c X μ ll c′ X′ P*
assume *A*: ((*c*,*X*),*ll*,(*c′*,*X′*))∈*ldpntrc* Δ   *wn-c* Δ *c μ*   *X*=*locks-μ μ* **and**
C: !!*μ′*. ⟦*X′*=*locks-μ μ′*; *wn-c* Δ *c′ μ′*⟧ ⟹ *P*
hence *P*
proof (*induct arbitrary*: *μ P rule*: *trcl-pair-induct*)
case *empty* **thus** *?case* **by** *auto*
next
case (*cons c x l ch Xh ll c′ X′ μ P*) **note** [*simp*]=‹*x*=*locks-μ μ*›
from *wnc-preserve-singlestep*[*OF cons.hyps*(*1*)[*simplified*] *cons.prems*(*1*)]
obtain *μ′* **where** *P*: *wn-c* Δ *ch μ′*   *Xh*=*locks-μ μ′* .

    from *cons.hyps*(*3*)[*OF P*] *cons.prems*(*3*) **show** *?case* **by** *blast*
  qed
} **with** *A C* **show** *?thesis* **by** *blast*
qed

10.3 Well-Nestedness Condition on Trees

Now we define well-nestedness on scheduling trees. Note that scheduling trees that contain spawn steps with locks interaction are not well-nested.

We define two equivalent formulations of well-nestedness of a tree:

fun *wn-t'* :: ('*P*,'Γ,'*L*,'*X*) *lex-tree* ⇒ '*X list* ⇒ *bool* **where**
  wn-t' (*NLEAF* π) μ ⟷ *distinct* μ |
  wn-t' (*NNOSPAWN* (*LNone l*) *t*) μ ⟷ *wn-t' t* μ |
  wn-t' (*NSPAWN* (*LNone l*) *ts t*) μ ⟷ *wn-t' t* μ ∧ *wn-t' ts* [] |
  wn-t' (*NNOSPAWN* (*LAcq x*) *t*) μ ⟷ *wn-t' t* (*x*#μ) ∧ *x*∉*set* μ |
  wn-t' (*NNOSPAWN* (*LRel x*) *t*) μ ⟷
      (∃ μ'. μ=*x*#μ' ∧ *wn-t' t* μ' ∧ *x*∉*set* μ') |
  wn-t' - - ⟷ *False*

inductive *wn-t* :: ('*P*,'Γ,'*L*,'*X*) *lex-tree* ⇒ '*X list* ⇒ *bool* **where**
  distinct μ ⟹ *wn-t* (*NLEAF* π) μ |
  wn-t t μ ⟹ *wn-t* (*NNOSPAWN* (*LNone l*) *t*) μ |
  ⟦*wn-t t* μ; *wn-t ts* [] ⟧ ⟹ *wn-t* (*NSPAWN* (*LNone l*) *ts t*) μ |
  ⟦*wn-t t* (*x*#μ); *x*∉*set* μ⟧ ⟹ *wn-t* (*NNOSPAWN* (*LAcq x*) *t*) μ |
  ⟦*wn-t t* μ; *x*∉*set* μ⟧ ⟹ *wn-t* (*NNOSPAWN* (*LRel x*) *t*) (*x*#μ)

inductive *lock-valid-xs* **where**
  distinct xs ⟹ *lock-valid-xs* (*LNone l*) *xs xs* |
  ⟦*distinct xs*; *x*∉*set xs*⟧ ⟹ *lock-valid-xs* (*LRel x*) (*x*#*xs*) *xs* |
  ⟦*distinct xs*; *x*∉*set xs*⟧ ⟹ *lock-valid-xs* (*LAcq x*) *xs* (*x*#*xs*)

The two formulations of well-nestedness of trees are, indeed, equivalent:

lemma *wnt-eq-wnt'*: *wn-t t* μ = *wn-t' t* μ
  apply *safe*
  apply (*induct rule*: *wn-t.induct*)
  apply *auto*
  apply (*induct rule*: *wn-t'.induct*)
  apply (*auto intro*: *wn-t.intros*)
  done

Well-nestedness of trees also implies distinctness of the lock stacks

lemma *wnt-distinct*: *wn-t t* μ ⟹ *distinct* μ
  by (*induct rule*: *wn-t.induct*) *auto*
lemma *wnt-distinct'*: *wn-t' t ms* ⟹ *distinct ms*
  using *wnt-distinct wnt-eq-wnt'* **by** *auto*

lemma *all-t-wnt-distinct*: ∀ *t c'*. *tsem* Δ (*q*,*w*) *t c'* ⟶ *wn-t t* μ ⟹ *distinct* μ
  by (*auto intro*: *wn-t.intros wnt-distinct*)

10.4 Well-Nestedness of Hedges

The well-nestedness property of a hedge expresses that each tree is well-nested, and the allocated locks of the trees are consistent.

Consistency of a list of lock stacks. $\mu = s_1 \ldots s_n$ is consistent, iff all s_i are distinct and $\forall\, i\, j.\ i \neq j \longrightarrow set\ s_i \cap set\ s_j = \{\}$.

fun *cons-μ* :: *'X list list* ⇒ *bool* **where**
cons-μ [] ⟷ *True* |
cons-μ (*xs*#*μ*) ⟷ *cons-μ μ* ∧ *distinct xs* ∧ *set xs* ∩ *locks-μ μ* = {}

A hedge $h = t_1 \ldots t_n$ is well-nested w.r.t. a list $\mu = s_1 \ldots s_n$ of lock stacks (*wn-h h μ*), iff all t_i are well-nested w.r.t. stack s_i and μ is consistent.

fun *wn-h* **where**
wn-h [] [] ⟷ *True* |
wn-h (*t*#*h*) (*xs*#*μ*) ⟷ *wn-h h μ* ∧ *set xs* ∩ *locks-μ μ* = {} ∧ *wn-t' t xs* |
wn-h - - ⟷ *False*

lemma *cons-μ-append*[*simp*]:
cons-μ (*μ1*@*μ2*) ⟷ *cons-μ μ1* ∧ *cons-μ μ2* ∧ *locks-μ μ1* ∩ *locks-μ μ2* = {}
by (*induct μ1 arbitrary*: *μ2*) *auto*

10.4.1 Auxilliary Lemmas about *wn-h*

lemma *wn-h-simps*[*simp*]:
wn-h h [] ⟷ *h*=[]
wn-h [] *μ* ⟷ *μ*=[]
apply (*induct h*)
apply *auto*
apply (*induct μ*)
apply *auto*
done

lemma *wn-h-length*: *wn-h h μ* ⟹ *length h* = *length μ*
by (*induct h μ rule*: *wn-h.induct*) *auto*

lemma *wn-h-prepend-h*:
⟦ *wn-h* (*t*#*h*) *μ*;
!!*xs μ'*. ⟦ *μ*=*xs*#*μ'*; *wn-h h μ'*; *set xs* ∩ *locks-μ μ'* = {}; *wn-t' t xs* ⟧ ⟹ *P*
⟧ ⟹ *P*
by (*induct μ arbitrary*: *t h*) *auto*

lemma *wn-h-prepend-μ*:
⟦ *wn-h h* (*xs*#*μ*);
!!*t h'*. ⟦ *h*=*t*#*h'*; *wn-h h' μ*; *set xs* ∩ *locks-μ μ* = {}; *wn-t' t xs* ⟧ ⟹ *P*
⟧ ⟹ *P*
by (*induct h arbitrary*: *s μ*) *auto*

lemma *wn-h-append-h-helper*:

assumes
A: *wn-h h* μ *h1*@*h2*=*h* **and**
C: !!μ*1* μ*2*. ⟦ μ=μ*1*@μ*2* ∧ *wn-h h1* μ*1* ∧ *wn-h h2* μ*2* ∧
locks-μ μ*1* ∩ *locks-*μ μ*2* = {}⟧ ⟹ *P*
shows *P*
using *A C*
apply (*induct h* μ *arbitrary*: *h1 h2 P rule*: *wn-h.induct*)
apply *auto*
apply *fastsimp*
apply (*case-tac h1*)
apply *fastsimp*
apply *auto*
proof −
case *goal1*
show *P*
apply (*rule goal1*(*1*))
apply *simp*
apply (*rule-tac ?*μ*1.0* = *xs*#μ*1* **and** *?*μ*2.0* = μ*2* **in** *goal1*(*2*))
apply (*insert goal1*(*3*−))
apply *auto*
done
qed

lemma *wn-h-append-h*:
⟦*wn-h* (*h1*@*h2*) μ;
!!μ*1* μ*2*. ⟦ μ=μ*1*@μ*2* ∧ *wn-h h1* μ*1* ∧ *wn-h h2* μ*2* ∧
locks-μ μ*1* ∩ *locks-*μ μ*2* = {}⟧ ⟹ *P*
⟧ ⟹ *P*
using *wn-h-append-h-helper*
by *blast*

lemma *wn-h-append-*μ*-helper*:
assumes
A: *wn-h h* μ μ*1*@μ*2*=μ **and**
C: !!*h1 h2*. ⟦ *h*=*h1*@*h2* ∧ *wn-h h1* μ*1* ∧ *wn-h h2* μ*2* ∧
locks-μ μ*1* ∩ *locks-*μ μ*2* = {}⟧ ⟹ *P*
shows *P*
using *A C*
apply (*induct h* μ *arbitrary*: μ*1* μ*2 P rule*: *wn-h.induct*)
apply *auto*
apply (*case-tac* μ*1*)
apply *fastsimp*
apply *auto*
proof −
case *goal1*
show *P*
apply (*rule goal1*(*1*))
apply *simp*
apply (*rule-tac ?h1.0* = *t*#*h1* **and** *?h2.0* = *h2* **in** *goal1*(*2*))

```
    apply (insert goal1(3−))
    apply auto
    done
qed

lemma wn-h-append-μ:
  ⟦wn-h h (μ1@μ2);
    !!h1 h2. ⟦ h=h1@h2 ∧ wn-h h1 μ1 ∧ wn-h h2 μ2 ∧
               locks-μ μ1 ∩ locks-μ μ2 = {}
             ⟧ ⟹ P
  ⟧ ⟹ P
  using wn-h-append-μ-helper by blast

lemma wn-h-appendI:
  ⟦wn-h h1 μ1; wn-h h2 μ2; locks-μ μ1 ∩ locks-μ μ2 = {}⟧ ⟹
    wn-h (h1@h2) (μ1@μ2)
  by (induct h1 μ1 arbitrary: h2 μ2 rule: wn-h.induct) auto

lemma wn-h-prependI:
  ⟦wn-t' t xs; wn-h h μ; set xs ∩ locks-μ μ = {}⟧ ⟹ wn-h (t#h) (xs#μ)
  by auto

lemma wn-h-singlehE: ⟦wn-h [t] μ; !!xs. ⟦μ=[xs]; wn-t' t xs⟧ ⟹ P ⟧ ⟹ P
  by (cases μ) auto
```

Auxilliary lemma to split the list of lock-stacks w.r.t. to that a hedge is well-nested by some tree in that hedge.

```
lemma wn-h-split-aux:
  assumes
  WN: wn-h h μ and
  HFMT[simp]: h=h1@t#h2 and
  C: !!μ1 xs μ2. ⟦
      μ=μ1@xs#μ2;
      wn-t' t xs; wn-h h1 μ1; wn-h h2 μ2;
      locks-μ μ1 ∩ set xs = {}; locks-μ μ1 ∩ locks-μ μ2 = {};
      set xs ∩ locks-μ μ2 = {}
    ⟧ ⟹ P
  shows P
  using WN[simplified]
  apply (elim wn-h-append-h wn-h-prepend-h conjE)
  apply (rule C)
  apply (auto)
  done
```

10.4.2 Relation to Path Condition

We show that the notion of well-nestedness on paths and trees are equivalent, i.e. a configuration is well-nested w.r.t. a lock stack μ if and only if all trees from that configuration are well-nested w.r.t. μ.

A process π is well-nested w.r.t. some stack of locks μ, if all its execution trees are well-nested w.r.t. μ:

definition *wn-π-t* Δ π *xs* == (∀ *t c'*. *tsem* Δ π *t c'* ⟶ *wn-t t xs*)

definition *wn-c-h* Δ *c* μ == (∀ *h c'*. *hsem* Δ *c h c'* ⟶ *wn-h h* μ)

lemma *wn-π-tI*[*intro?*]: ⟦!!*t c'*. *tsem* Δ π *t c'* ⟹ *wn-t t xs* ⟧ ⟹ *wn-π-t* Δ π *xs*
  by (*auto simp add*: *wn-π-t-def*)
lemma *wn-c-hI*[*intro?*]: ⟦!!*h c'*. *hsem* Δ *c h c'* ⟹ *wn-h h* μ ⟧ ⟹ *wn-c-h* Δ *c* μ
  by (*auto simp add*: *wn-c-h-def*)

lemma *wn-π-t-distinct*: *wn-π-t* Δ π μ ⟹ *distinct* μ
  apply (*cases* π)
  apply (*unfold wn-π-t-def*)
  by (*auto intro*: *wn-t.intros wnt-distinct*)

lemma *wn-c-h-prepend1*: **assumes** *A*: *wn-c-h* Δ (π#*c*) (*xs*#μ)
  shows *wn-π-t* Δ π *xs*   *wn-c-h* Δ *c* μ   *set xs* ∩ *locks-μ* μ = {}
proof −
  from *A* **have** *A'*: !!*h c'*. *hsem* Δ (π#*c*) *h c'* ⟹ *wn-h h* (*xs*#μ)
    by (*auto simp add*: *wn-c-h-def*)
  from *A'*[*of map NLEAF* (π#*c*)   π#*c*, *simplified*]
  show *set xs* ∩ *locks-μ* μ = {}
    by *auto*
  show *wn-π-t* Δ π *xs* **proof**
    fix *t c'* **assume** *A*: *tsem* Δ π *t c'*
    from *A'*[*OF hsem-cons*[*OF A hsem-id*]] **show** *wn-t t xs*
      by (*auto simp add*: *wnt-eq-wnt'*)
  qed

  show *wn-c-h* Δ *c* μ **proof**
    fix *h c'* **assume** *A*: *hsem* Δ *c h c'*
    from *A'*[*OF hsem-cons*[*OF tsem-leaf A*]] **show** *wn-h h* μ **by** *auto*
  qed
qed

lemma *wn-c-h-prepend2*:
  ⟦*wn-π-t* Δ π *xs*; *wn-c-h* Δ *c* μ; *set xs* ∩ *locks-μ* μ = {}⟧ ⟹
    wn-c-h Δ (π#*c*) (*xs*#μ)
  apply (*auto simp add*: *wn-c-h-def wn-π-t-def*)
  apply (*erule hsem-split-single*)
  apply (*auto simp add*: *wnt-eq-wnt'*)
  done

lemma *wn-c-h-prepend*[*simp*]:
  wn-c-h Δ (π#*c*) (*xs*#μ) ⟷
    wn-π-t Δ π *xs* ∧ *wn-c-h* Δ *c* μ ∧ *set xs* ∩ *locks-μ* μ = {}
  using *wn-c-h-prepend1 wn-c-h-prepend2* **by** *fast*

```
lemma wn-c-h-empty[simp]: wn-c-h Δ c [] ⟷ (c=[]) by (auto simp add: wn-c-h-def)

lemma wn-c-h-prepend-c:
  ⟦wn-c-h Δ (π#c) μ;
    !!xs μ'. ⟦μ=xs#μ'; wn-π-t Δ π xs; wn-c-h Δ c μ';
             set xs ∩ locks-μ μ' = {} ⟧ ⟹ P
  ⟧ ⟹ P
  by (cases μ) (auto)

lemma wn-c-h-simps[simp]: wn-c-h Δ [] μ ⟷ (μ=[])
  by (unfold wn-c-h-def) (auto)

lemma (in LDPN) wnπ2wnt: ⟦tsem Δ (q,w) t c'; wn-π Δ (q,w) μ⟧ ⟹ wn-t t μ
proof (induct arbitrary: μ rule: tsem.induct)
  case tsem-leaf thus ?case by (auto intro: wn-t.intros dest: wn-π-distinct)
next
  case (tsem-nospawn q γ l q' w r t ct' μ) note C=this
  show ?case proof (cases l)
    case LNone[simp]
    from C have wn-t t μ
      by (rule-tac C) (auto intro: ppairs.intros C simp add: wn-π-def)
    thus ?thesis by (auto intro: wn-t.intros)
  next
    case (LAcq x)[simp]
    from C have wn-t t (x#μ)
      by (rule-tac C) (auto intro: ppairs.intros C simp add: wn-π-def)
    moreover hence x∉set μ by (auto dest: wnt-distinct)
    ultimately show ?thesis by (auto intro: wn-t.intros)
  next
    case (LRel x)[simp]
    from wn-π-rel[OF tsem-nospawn.hyps(1)[simplified] tsem-nospawn.prems]
    obtain μ' where [simp]: μ = x # μ'   x ∉ set μ' .
    from C have wn-t t μ'
      by (rule-tac C) (auto intro: ppairs.intros C simp add: wn-π-def)
    thus ?thesis by (auto intro: wn-t.intros)
  qed
next
  case (tsem-spawn q γ l qs ws q' w ts cs' r t ct' μ) note C=this
  then obtain ll where [simp]: l=LNone ll by (cases l) auto
  from C have wn-t t μ
    apply simp-all
    apply (rule-tac C)
    apply (auto intro: ppairs.intros C simp add: wn-π-def)
    done
  moreover from tsem-spawn.hyps(1,3) tsem-spawn.prems[rule-format]
  have wn-t ts [] by (auto intro: wn-π-spawn2)
  ultimately show ?case by (auto intro: wn-t.intros)
qed
```

```
lemma (in LDPN) wnt2wnp:
  ⟦ppairs Δ (q,w) en l; ∀ t c'. tsem Δ (q,w) t c' ⟶ wn-t t μ⟧ ⟹
    (¬en ⟶ wn-p l μ) ∧ (en ⟶ wn-p l [])
proof (induct arbitrary: μ rule: ppairs.induct)
  case ppairs-empty thus ?case by (auto intro: all-t-wnt-distinct)
next
  case (ppairs-genenv q γ a qs ws q' w en l r μ)
  have ∀ t c'. tsem Δ (qs, ws) t c' ⟶ wn-t t [] proof (intro allI impI)
    fix t c'
    assume A: tsem Δ (qs, ws) t c'
    from ppairs-genenv.prems[rule-format,
            OF tsem-spawn[OF ppairs-genenv.hyps(1) A tsem-leaf]
        ]
    show wn-t t [] by (auto elim: wn-t.cases)
  qed
  from ppairs-genenv.hyps(3)[OF this] show ?case by blast
next
  case (ppairs-mvenv1 q γ a q' w r l μ)[simplified] show ?case
  proof (simp, cases a)
    case LNone[simp]
    from ppairs-mvenv1.prems have ∀ t c'. tsem Δ (q', w @ r) t c' ⟶ wn-t t μ
     by auto (drule tsem-nospawn[OF ppairs-mvenv1.hyps(1)], auto elim: wn-t.cases)
    with ppairs-mvenv1.hyps(3) show wn-p l [] by auto
  next
    case (LAcq x)
    with tsem-nospawn[OF ppairs-mvenv1.hyps(1)] ppairs-mvenv1.prems
    show wn-p l []
      by (fastsimp intro: ppairs-mvenv1.hyps(3)[rule-format] elim: wn-t.cases)
  next
    case (LRel x) note [simp]=this
    from tsem-nospawn[OF ppairs-mvenv1.hyps(1)[simplified] tsem-leaf]
    have T: Ex (tsem Δ
                  (q, γ # r)
                  (NNOSPAWN (LRel x) (NLEAF (q', w @ r)))
              )
      by blast
    obtain μ' where [simp]: μ=x#μ'    x∉set μ'
      apply (rule wn-t.cases[OF ppairs-mvenv1.prems[rule-format, OF T]])
      by simp-all
    from tsem-nospawn[OF ppairs-mvenv1.hyps(1)] ppairs-mvenv1.prems
    show wn-p l []
      by (fastsimp intro: ppairs-mvenv1.hyps(3)[rule-format] elim: wn-t.cases)


  qed
next
  case (ppairs-mvenv2 q γ a qs ws q' w r l μ)[simplified]
  show ?case
```

using *tsem-spawn*[*OF ppairs-mvenv2.hyps*(*1*)] *ppairs-mvenv2.prems*
ppairs-mvenv2.hyps(*1*)
apply (*cases a*)
apply (*blast intro*: *ppairs-mvenv2.hyps*(*3*)[*rule-format*] *elim*: *wn-t.cases*)
apply *auto*
done
next
case (*ppairs-prepend1 q* γ *a q′ w r l* μ)[*simplified*] **show** *?case*
proof (*simp*, *cases a*)
case *LNone*
with *tsem-nospawn*[*OF ppairs-prepend1.hyps*(*1*)] *ppairs-prepend1.prems*
show *wn-p* (*a#l*) μ
by (*fastsimp intro*: *ppairs-prepend1.hyps*(*3*)[*rule-format*] *elim*: *wn-t.cases*)
next
case (*LAcq x*)
with *tsem-nospawn*[*OF ppairs-prepend1.hyps*(*1*)] *ppairs-prepend1.prems*
show *wn-p* (*a#l*) μ
by (*fastsimp intro*: *ppairs-prepend1.hyps*(*3*)[*rule-format*] *elim*: *wn-t.cases*)
next
case (*LRel x*) **note** [*simp*]=*this*
from *tsem-nospawn*[*OF ppairs-prepend1.hyps*(*1*)[*simplified*] *tsem-leaf*] **have**
T: *Ex* (*tsem* Δ (*q*, γ # *r*) (*NNOSPAWN* (*LRel x*) (*NLEAF* (*q′*, *w* @ *r*))))
by *blast*
obtain μ' **where** [*simp*]: μ=*x*#μ' *x*∉*set* μ'
apply (*rule wn-t.cases*[*OF ppairs-prepend1.prems*[*rule-format*, *OF T*]])
by *simp-all*
from *tsem-nospawn*[*OF ppairs-prepend1.hyps*(*1*)] *ppairs-prepend1.prems*
show *wn-p* (*a#l*) μ
by (*fastsimp intro*: *ppairs-prepend1.hyps*(*3*)[*rule-format*] *elim*: *wn-t.cases*)

qed
next
case (*ppairs-prepend2 q* γ *a qs ws q′ w r l* μ)[*simplified*]
from *ppairs-prepend2.prems*[*rule-format*] **have**
H: !!*c t*. *tsem* Δ (*q*, γ # *r*) *t c* ⟹ *wn-t t* μ **by** *blast*
show *?case* **using** *ppairs-prepend2.hyps*(*1*)
by (*cases a*)
(*auto intro*: *ppairs-prepend2.hyps*(*3*)[*rule-format*]
dest: *tsem-spawn*[*OF ppairs-prepend2.hyps*(*1*) *tsem-leaf*] *H*
elim: *wn-t.cases*
)
qed

theorem (**in** *LDPN*) *wnπ-eq-wnπt*: *wn-π* Δ π μ ⟷ *wn-π-t* Δ π μ **using** *wnt2wnp*
by (*auto intro*: *wnπ2wnt simp add*: *wn-π-def wn-π-t-def*)

theorem (**in** *LDPN*) *wnc-eq-wnch*: *wn-c* Δ *c* μ ⟷ *wn-c-h* Δ *c* μ

apply *rule*
apply (*induct c arbitrary*: μ)
apply *simp*
apply (*erule wn-c-prepend-c*)
apply (*simp add*: *wnπ-eq-wnπt*)
apply (*induct c arbitrary*: μ)
apply (*auto simp add*: *wn-c-h-def*) [*1*]
apply (*erule wn-c-h-prepend-c*)
apply (*simp add*: *wnπ-eq-wnπt*)
done

10.5 Well-Nestedness and Tree Scheduling

In this section we show that well-nestedness is invariant under the tree scheduling relation. This is important, as it shows that we cannot reach non-well-nested trees from well-nested ones.

lemma *wnt-preserve-nospawn*:
⟦ *lock-valid* (*set xs*) *l X'*; *wn-t'* (*NNOSPAWN l t*) *xs*⟧ ⟹
∃ *xs'*. *X'*=*set xs'* ∧ *lock-valid-xs l xs xs'* ∧ *wn-t' t xs'*
apply (*cases l*)
apply (*rule-tac x*=*xs* **in** *exI*)
apply (*force intro*: *lock-valid-xs.intros dest*: *wnt-distinct'*)
apply (*rule-tac x*=(*X*#*xs*) **in** *exI*)
apply (*force intro*: *lock-valid-xs.intros dest*: *wnt-distinct'*)
apply (*rule-tac x*=*tl xs* **in** *exI*)
apply (*force simp add*: *insert-ident intro*: *lock-valid-xs.intros dest*: *wnt-distinct'*)
done

lemma *wn-h-preserve-nospawn*:
⟦ *lock-valid* (*locks-μ μ*) *l X'*; *wn-h* (*h1*@(*NNOSPAWN l t*)#*h2*) μ⟧ ⟹
∃ μ'. *X'*=*locks-μ* μ' ∧ *wn-h* (*h1*@*t*#*h2*) μ'
apply (*cases l*)
apply (*auto elim*!: *wn-h-prepend-h wn-h-append-h*)
apply (*rule-tac x*=μ*1*@*xs*#μ' **in** *exI*)
apply (*force intro*!: *wn-h-appendI*)
apply (*rule-tac x*=μ*1*@(*X*#*xs*)#μ' **in** *exI*)
apply (*force intro*!: *wn-h-appendI*)
apply (*rule-tac x*=μ*1*@(μ'*a*)#μ' **in** *exI*)
apply (*rule conjI*)
apply (*rule iffD1*[*OF insert-ident*])
apply *assumption*
apply (*auto intro*!: *wn-h-appendI*)
done

All-in-one lemma for reasoning about a non-spawning step on a well-nested hedge. In words: If we make a non-speaining step on a well-nested hedge:

- We can split the list of lock stacks according to the tree that made the step,

- The lock stack of the tree that made the step changes according to the label (cf. *lock-valid-xs*),
- And the resulting hedge is well-nested w.r.t. the new locks, too.

lemma *wn-h-split-nospawn*:
  assumes
  A: *lock-valid* (*locks-μ μ*) *l Xh*   *wn-h* (*h1*@(*NNOSPAWN l t*)#*h2*) *μ* **and**
  C: !!*μ1 xs μ2 xsh*. ⟦
    μ=*μ1*@*xs*#*μ2*;
    Xh=*locks-μ μ1* ∪ *set xsh* ∪ *locks-μ μ2*;
    lock-valid-xs l xs xsh;
    wn-t′ (*NNOSPAWN l t*) *xs*;
    wn-t′ t xsh;
    wn-h h1 μ1;
    wn-h h2 μ2;
    wn-h (*h1*@*t*#*h2*) (*μ1*@*xsh*#*μ2*);
    locks-μ μ1 ∩ *set xs* = {};
    locks-μ μ1 ∩ *set xsh* = {};
    locks-μ μ1 ∩ *locks-μ μ2* = {};
    locks-μ μ2 ∩ *set xs* = {};
    locks-μ μ2 ∩ *set xsh* = {}
  ⟧ ⟹ *P*
  shows *P*
proof −
  from *A*(*2*) **obtain** *μ1 xs μ2* **where**
    SPLIT-simp[*simp*]: *μ*=*μ1*@*xs*#*μ2* **and**
    SPLIT: *wn-h h1 μ1*   *wn-t′* (*NNOSPAWN l t*) *xs*   *wn-h h2 μ2*
        locks-μ μ1 ∩ *set xs* = {}   *locks-μ μ1* ∩ *locks-μ μ2* = {}
        set xs ∩ *locks-μ μ2* = {}
    by (*fastsimp elim*: *wn-h-prepend-h wn-h-append-h*)
  show *?thesis* **proof** (*cases l*)
    case *LNone*[*simp*]
    from *SPLIT*(*2*) **have** *wn-t′ t xs*   *lock-valid-xs l xs xs*
      by (*auto intro*: *lock-valid-xs.intros dest*: *wnt-distinct′*)
    moreover with *SPLIT* **have** *wn-h* (*h1*@*t*#*h2*) (*μ1*@*xs*#*μ2*)
      by (*auto intro*!: *wn-h-appendI wn-h-prependI*)
    ultimately show *?thesis* **using** *A*(*1*)[*simplified*] *SPLIT SPLIT-simp*
      by (*blast intro*!: *C*)
  next
    case (*LRel x*)[*simp*]
    from *SPLIT*(*2*) **obtain** *xsh* **where**
      [*simp*]: *xs*=*x*#*xsh* **and**
        WN′: *wn-t′ t xsh*   *x*∉*set xsh*
      by *auto*
    moreover with *SPLIT* **have** *wn-h* (*h1*@*t*#*h2*) (*μ1*@*xsh*#*μ2*)
      by (*auto intro*!: *wn-h-appendI wn-h-prependI*)
    moreover from *wnt-distinct′*[*OF WN′*(*1*)] *WN′*(*2*) **have**

lock-valid-xs l xs xsh
by (*auto intro*: *lock-valid-xs.intros*)
ultimately show *?thesis*
using *A*(*1*)[*simplified*] *WN′ SPLIT SPLIT-simp* **by** (*fastsimp intro*!: *C*)
next
case (*LAcq x*)[*simp*]
from *SPLIT*(*2*) **have** *wn-t′ t* (*x#xs*) *lock-valid-xs l xs* (*x#xs*)
by (*auto intro*: *lock-valid-xs.intros dest*!: *wnt-distinct′*)
moreover with *SPLIT A*(*1*)[*simplified*] **have** *wn-h* (*h1@t#h2*) (*μ1@*(*x#xs*)*#μ2*)
by (*auto intro*!: *wn-h-appendI wn-h-prependI*)
ultimately show *?thesis*
using *A*(*1*)[*simplified*] *SPLIT SPLIT-simp*
apply (*rule-tac C*)
apply *assumption+*
defer
apply *assumption+*
apply *auto*
done
qed
qed

lemma *wn-h-preserve-spawn*:
⟦ *lock-valid* (*locks-μ μ*) *l X′*; *wn-h* (*h1@*(*NSPAWN l ts t*)*#h2*) *μ*⟧ ⟹
∃ *μ′*. *X′=locks-μ μ′* ∧ *wn-h* (*h1@ts#t#h2*) *μ′*
apply (*cases l*)
apply (*auto elim*!: *wn-h-prepend-h wn-h-append-h*)
apply (*rule-tac x=μ1@*[]*#xs#μ′* **in** *exI*)
apply (*auto intro*!: *wn-h-appendI*)
done

lemma *wn-h-preserve-spawn′*:
⟦ *lock-valid* (*locks-μ μ*) *l X′*; *wn-h* (*h1@*(*NSPAWN l ts t*)*#h2*) *μ*⟧ ⟹
∃ *μ1 xs μ2*. *μ=μ1@xs#μ2* ∧ *X′=locks-μ μ1* ∪ *set xs* ∪ *locks-μ μ2* ∧
wn-h (*h1@ts#t#h2*) (*μ1@*[]*#xs#μ2*)
apply (*cases l*)
apply (*auto elim*!: *wn-h-prepend-h wn-h-append-h*)
apply (*rule-tac x=μ1* **in** *exI*)
apply (*rule-tac x=xs* **in** *exI*)
apply (*rule-tac x=μ′* **in** *exI*)
apply (*auto intro*!: *wn-h-appendI*)
done

lemma *wn-h-preserve-rel*:
⟦ (*h*,*l*,*h′*)∈*sched-rel*; *lock-valid* (*locks-μ μ*) *l X′*; *wn-h h μ*;
!!*μ′*. ⟦ *X′=locks-μ μ′*; *wn-h h′ μ′*⟧ ⟹ *P*
⟧ ⟹ *P*
by (*auto elim*!: *sched-rel.cases dest*: *wn-h-preserve-spawn wn-h-preserve-nospawn*)

lemma *wn-h-spawn-simps*[*simp*]:

¬*wn-h* (*h* @ (*NSPAWN* (*LAcq x*) *ts t*) # *h'*) *μ*
¬*wn-h* (*h* @ (*NSPAWN* (*LRel x*) *ts t*) # *h'*) *μ*
by (*auto elim*!: *wn-h-prepend-h wn-h-append-h*)

lemmas *wn-h-spawn-simps-add*[*simp*] =
wn-h-spawn-simps[**where** *h*=[], *simplified*]
wn-h-spawn-simps[**where** *h*=[*tx*], *simplified*, *standard*]

lemma *wn-h-spawn-imp-LNoneE*:
⟦*wn-h* (*h* @ (*NSPAWN l ts t*) # *h'*) *μ*; !!*ll*. *l*=*LNone ll* ⟹ *P*⟧ ⟹ *P*
by (*cases l*) *auto*

end

11 Acquisition Structures

theory *Acqh*
imports *Main Semantics WellNested SpecialLemmas*
begin

11.1 Utilities

11.1.1 Combinators for *option*-datatype

Extending a function to option datatype, where *None* indicates failure

fun *opt-ext1* :: (*'a* ⇒ *'b option*) ⇒ *'a option* ⇒ *'b option* **where**
opt-ext1 f None = *None* |
opt-ext1 f (*Some x*) = *f x*

fun *opt-ext2* :: (*'a* ⇒ *'b* ⇒ *'c option*) ⇒ *'a option* ⇒ *'b option* ⇒ *'c option*
where
opt-ext2 f None - = *None* |
opt-ext2 f - *None* = *None* |
opt-ext2 f (*Some x*) (*Some y*) = *f x y*

lemma *opt-ext2-simps*[*simp*]:
opt-ext2 f x None = *None* **by** (*cases x*) *auto*

lemma *opt-ext2-alt*:
opt-ext2 f x y = (
case x of
None ⇒ *None* |
Some xx ⇒ (*case y of*
None ⇒ *None* |

Some yy ⇒ *f xx yy*
)
)
by (*cases* (*f*,*x*,*y*) *rule*: *opt-ext2.cases*) *auto*

11.2 Acquisition Structures

Acquisition structures are an abstraction of scheduling trees, that are sufficient to decide whether a tree is schedulable. The basic concept of acquisition structures was invented by Kahlon et al. [4, 3] as abstraction of a linear execution of a single pushdown system. We extend this concept here to scheduling trees of DPNs.

An acquisition or release history is a partial map from locks to set of locks. This is the same representation as in [3]. Another, equivalent representation is as a set of locks and a graph on locks.

An acquisition structure is a triple of a release history, a set of locks and an acquisition history.

types
'X ah = *'X* ⇒ *'X set option*
'X as = *'X ah* × *'X set* × *'X ah*

This is a collection of the common split-lemmas required when reasoning about acquisition histories

lemmas *eahl-splits* = *option.split-asm list.split-asm prod.split-asm split-if-asm*

11.2.1 Parallel Composition

fun *as-comp* :: *'X as* ⇒ *'X as* ⇒ *'X as option* **where**
as-comp (*l*,*u*,*e*) (*l'*,*u'*,*e'*) = (
if dom l ∩ *dom l'* = {} ∧ *dom e* ∩ *dom e'* = {} *then*
Some (*l*++*l'*,*u*∪*u'*,*e*++*e'*)
else
None
)

definition *as-comp-op*
:: *'X as option* ⇒ *'X as option* ⇒ *'X as option* (**infixr** ‖ *56*) **where**
op ‖ == *opt-ext2 as-comp*

lemma *as-comp-op-simps*[*simp*]:
None ‖ *x* = *None*
x ‖ *None* = *None*
Some a ‖ *Some b* = *as-comp a b*
by (*unfold as-comp-op-def*) *auto*

lemma *as-comp-assoc-helper*:
(*Some x* ‖ *Some y*) ‖ *Some z* = *Some x* ‖ *Some y* ‖ *Some z*

by (*cases x*, *cases y*, *cases z*) *auto*

lemma *as-comp-assoc*: $(x\|y)\|z = x\|y\|z$
apply (*cases x*, *simp*)
apply (*cases y*, *simp*)
apply (*cases z*, *simp*)
apply (*simp only*: *as-comp-assoc-helper*)
done

interpretation *as-comp-acz*: *ACIZ*[*op* ‖ *Some* (*empty*,{},*empty*) *None*]
apply (*unfold-locales*)
apply (*auto simp add*: *as-comp-assoc*)
apply (*case-tac* (*as-comp*,*x*,*y*) *rule*: *opt-ext2.cases*)
apply (*auto simp add*: *map-add-comm*)
apply *auto*
apply (*case-tac x*)
apply *simp-all*
apply (*case-tac a*, *case-tac b*)
apply *simp*
done

lemma *as-comp-SomeE*:
⟦*h1* ‖ *h2* = *Some* (*l*,*u*,*e*);
!!*l1 u1 e1 l2 u2 e2*. ⟦ *h1*=*Some* (*l1*,*u1*,*e1*); *h2*=*Some* (*l2*,*u2*,*e2*);
dom l1 ∩ *dom l2* = {}; *dom e1* ∩ *dom e2* = {};
l=*l1*++*l2*; *u*=*u1*∪*u2*; *e*=*e1*++*e2*
⟧ ⟹ *P*
⟧ ⟹ *P*
apply (*unfold as-comp-op-def*)
apply (*cases h1*, *cases h2*, *simp-all*)
apply (*cases h2*, *simp-all*)
apply (*case-tac* (*a*,*aa*) *rule*: *as-comp.cases*)
apply (*simp split*: *split-if-asm*)
apply *blast*
done

11.2.2 Acquisition Structures of Scheduling Trees and Hedges

This function adds a set of locks to every entry in a release history. On graph interpretation, this corresponds to adding edges from any initially released lock to any lock in X.

definition *l-add-use* :: *'X ah* ⇒ *'X set* ⇒ *'X ah* **where**
l-add-use l X == λ*x*. *case l x of None* ⇒ *None* | *Some Y* ⇒ *Some* (*Y*∪*X*)

This function removes an initially released lock x from the release history. On graph interpretation, this corresponds to removing the node x from the graph.

definition *l-remove* :: *'X ah* ⇒ *'X* ⇒ *'X ah* **where**

l-remove l x == λy. *if y=x then None else l y*

The acquisition history of a tree is defined inductively over the tree structure. Note that we assume that spawn steps have no lock operation. For spawn steps with an operation on locks, the acquisition structure is defined to be *None*. We further assume that a tree contains no two initial releases of the same lock. In this case, its acquisition structure has no meaning any more. However, if an execution tree contains two final acquisitions of the same lock, its acquisition structure is defined to be *None*.

Intuitively, the release history maps all locks that are initially released to the set of locks that have to be used before the initial release. The set of used locks contains the locks that are used by the execution tree (But not the locks that are only initially released or finally acquired). The acquisition history maps all locks that are finally acquired to the set of locks that have to be used after the final acquisition.

```
fun as :: ('P,'Γ,'L,'X) lex-tree ⇒ 'X as option where
  as (NLEAF π) = Some (empty,{},empty) |
  as (NNOSPAWN (LNone l) t) = as t |
  as (NSPAWN (LNone l) ts t) = as ts ∥ as t |
  as (NNOSPAWN (LAcq x) t) = (
    case as t of
      None ⇒ None |
      Some (l,u,e) ⇒
        if x∈dom l then
          Some (l-add-use (l-remove l x) {x},insert x u,e)
        else if x∉dom e then
          Some (l,u, e(x↦u))
        else
          None
  ) |
  as (NNOSPAWN (LRel x) t) = (
    case as t of
      None ⇒ None |
      Some (l,u,e) ⇒ Some (l(x↦{}),u,e)
  ) |
  as - = None
```

The aquisition structure of a hedge is the parallel composition of the acquisition structures of its trees. The acquisition structure of the empty hedge is the identity acquisition structure *Some* (*empty*, {}, *empty*).

```
fun ash :: ('P,'Γ,'L,'X) lex-hedge ⇒ 'X as option where
  ash [] = Some (empty,{},empty) |
  ash (t#h) = as t ∥ ash h
```

```
lemma l-add-use-dom[simp]: dom (l-add-use l X) = dom l
  by (unfold l-add-use-def) (auto split: option.split-asm)
```

```
lemma l-add-use-empty[simp]: l-add-use empty X = empty
```

by (*rule ext*) (*auto simp add*: *l-add-use-def split*: *option.split*)

lemma *l-add-use-eq-empty*[*simp*]: *l-add-use f X* = *empty* ⟷ *f*=*empty*
apply (*auto*)
apply (*rule ext*)
apply (*drule-tac x*=*x* **in** *fun-cong*)
apply (*simp add*: *l-add-use-def split*: *option.split-asm*)
done

lemma *l-add-use-add*[*simp*]:
l-add-use (*l*++*l′*) *X* = *l-add-use l X* ++ *l-add-use l′ X*
apply (*unfold l-add-use-def*)
apply (*rule ext*)
by (*auto split*: *option.split simp add*: *map-add-def*)

lemma *l-add-use-le*: *l* ≤ *l-add-use l X*
apply (*auto simp add*: *l-add-use-def intro*!: *le-funI*)
apply (*case-tac l x*)
apply *auto*
done

lemma *l-remove-add*[*simp*]: *l-remove* (*l1*++*l2*) *m* = *l-remove l1 m* ++ *l-remove l2 m*
by (*unfold l-remove-def map-add-def*) (*auto intro*: *ext*)

lemma *l-remove-no-eff*[*simp*]: *x*∉*dom l* ⟹ *l-remove l x* = *l*
by (*unfold l-remove-def*) (*auto intro*: *ext*)

lemma *l-remove-dom*[*simp*]: *dom* (*l-remove l x*) = *dom l* − {*x*}
by (*unfold l-remove-def*) (*auto split*: *split-if-asm*)

lemma *l-remove-app*[*simp*]:
l-remove l x x = *None*
x≠*x′* ⟹ *l-remove l x x′* = *l x′*
by (*unfold l-remove-def*) *auto*

lemma *l-remove-eq-empty*: *l-remove l x* = *empty* ⟹ *dom l* ⊆ {*x*}
by (*fastsimp simp add*: *l-remove-def dest*: *fun-cong split*: *split-if-asm*)

lemma *l-remove-le-l* [*simp*]: *l-remove l x* ≤ *l*
by (*auto simp add*: *l-remove-def intro*: *le-funI*)

lemma *as-ran-e-le-u*: *as t* = *Some* (*l*,*u*,*e*) ⟹ ⋃*ran e* ⊆ *u*
apply (*induct t arbitrary*: *l u e*)
apply *fastsimp*
apply (*case-tac L*)
apply (*simp-all split*: *eahl-splits*)
apply *fastsimp*
apply *fastsimp*

apply (*case-tac L*)
apply (*simp-all*)
apply (*fastsimp elim*: *as-comp-SomeE*)
done

lemma *ash-le-u*: *ash h* = *Some* (*l*,*u*,*e*) $\Longrightarrow$ $\bigcup$*ran e* $\subseteq$ *u*
proof (*induct h arbitrary*: *l u e rule*: *ash.induct*)
case *1* **thus** *?case* **by** *auto*
next
case *2* **thus** *?case*
apply *simp*
apply (*erule as-comp-SomeE*)
apply (*fastsimp dest*!: *as-ran-e-le-u*)
done
qed

lemma *ash-final*[*simp*]: *final h* $\Longrightarrow$ *ash h*=*Some* (*empty*,{},*empty*)
apply (*induct h*)
apply *auto*
apply (*case-tac a*)
apply *simp-all*
done

lemma *ash-append*[*simp*]: *ash* (*h1*@*h2*) = *ash h1* ∥ *ash h2*
by (*induct h1 arbitrary*: *h2*) (*auto simp add*: *as-comp-acz.simps*)

lemma *ash-LNone-simps*[*simp*]:
ash (*h1*@*NSPAWN* (*LNone l*) *ts t*#*h2*) = *ash* (*h1*@*ts*#*t*#*h2*)
ash (*h1*@*NNOSPAWN* (*LNone l*) *t*#*h2*) = *ash* (*h1*@*t*#*h2*)
by (*simp-all add*: *as-comp-acz.simps*)

11.3 Consistency of Acquisition Structures

The consistency criterium of an acquisition structure decides whether the corresponding hedge can be scheduled. Note that we currently do not check this criterium during construction of the acquisition structure, but only at the end, for the completely constructed acquisition structure.

The consistency criterium has two parts. The first part is a generalization of the $\neg\exists\, m_1,m_2.\ m_1{\in}h_1(m_2) \wedge m_2{\in}h_2(m_1)$-condition of [4]. There, the condition was checked for two separate acquisition histories h_1 and h_2 that resulted from executions of two independent pushdown systems. Here, we have one execution described as a tree. This criterium can be interpreted as checking acyclicity of a graph defined by the acquisition histories. In [4], every possible cycle has length two, hence their condition is sufficient. In our setting, a cycle may have arbitrary length (bounded only by the number of locks), hence we use a general cyclicity check.

The acquisition and release histories encode a graph between locks. For an acquisition history e, the graph contains an edge $(x,\ x')$ if x has to be finally acquired before x' is used, that is if $x \in \mathit{dom}\ e \wedge x' \in \mathit{the}\ (e\ x)$

For a release history l, the graph contains an edge $(x,\ x')$ if x has to be used before x' is initially released, that is if $x' \in \mathit{dom}\ l \wedge x \in \mathit{the}\ (l\ x')$

definition *agraph* :: *'X ah* ⇒ (*'X*×*'X*) *set* **where**
agraph e == { (*x*,*x'*) . *x*∈*dom e* ∧ *x'*∈*the* (*e x*) }
definition *rgraph* :: *'X ah* ⇒ (*'X*×*'X*) *set* **where**
rgraph l == { (*x*,*x'*) . *x'*∈*dom l* ∧ *x*∈*the* (*l x'*) }

lemma *agraph-alt*: *agraph e* = { (*x*,*x'*) . ∃ *X'*. *e x* = *Some X'* ∧ *x'*∈*X'*}
by (*unfold agraph-def*) *auto*
lemma *rgraph-alt*: *rgraph l* = { (*x*,*x'*) . ∃ *X*. *l x'* = *Some X* ∧ *x*∈*X* }
by (*unfold rgraph-def*) *auto*

For the same map, the acquisition graph is the converse of the release graph. This lemma makes reasoning simpler at some points, as acquisition and release histories have the same type, and cyclicity is equivalent for a graph and its converse.

lemma *agraph-rgraph-converse*: *agraph h* = (*rgraph h*)$^{-1}$
by (*unfold agraph-def rgraph-def*) *auto*

lemma *agraph-add-union*:
⟦*dom e* ∩ *dom e'* = {}⟧ ⟹ *agraph* (*e*++*e'*) = *agraph e* ∪ *agraph e'*
by (*unfold agraph-def*) (*auto simp add*: *map-add-def split*: *option.split-asm*)

lemma *rgraph-add-union*:
⟦*dom l* ∩ *dom l'* = {}⟧ ⟹ *rgraph* (*l*++*l'*) = *rgraph l* ∪ *rgraph l'*
by (*unfold rgraph-def*) (*auto simp add*: *map-add-def split*: *option.split-asm*)

lemma *agraph-domain-simp*[*simp*]:
Domain (*agraph h*) = *dom h* − { *x* . *h x* = *Some* {} }
by (*unfold agraph-def*) *auto*

lemma *agraph-range-simp*[*simp*]: *Range* (*agraph h*) = ⋃ *ran h*
by (*unfold agraph-def*) (*auto simp add*: *ran-def*)

lemma *rgraph-domain-simp*[*simp*]: *Domain* (*rgraph h*) = ⋃ *ran h*
by (*unfold rgraph-def*) (*auto simp add*: *ran-def*)

lemma *rgraph-range-simp*[*simp*]:
Range (*rgraph h*) = *dom h* − { *x* . *h x* = *Some* {} }
by (*unfold rgraph-def*) *auto*

lemma *graph-empty*[*simp*]:
agraph empty = {}
rgraph empty = {}
by (*auto simp add*: *agraph-def rgraph-def*)

lemma *rgraph-add-use*: *rgraph* (*l-add-use l X*) = *rgraph l* ∪ *X*×*dom l*
by (*unfold rgraph-def l-add-use-def*) (*auto split*: *option.split-asm*)

lemma *rgraph-remove*: *rgraph* (*l-remove l x*) = *rgraph l* − *UNIV*×{*x*}
by (*unfold rgraph-def l-remove-def*) (*auto split*: *option.split-asm*)

lemma *rgraph-upd*: *x*∉*dom l* ⟹ *rgraph* (*l*(*x*↦*X*)) = *rgraph l* ∪ *X*×{*x*}
by (*unfold rgraph-def*) *auto*

lemmas *rgraph-ops* = *rgraph-add-use rgraph-remove rgraph-upd*

lemma *agraph-upd*: *x*∉*dom e* ⟹ *agraph* (*e*(*x*↦*X*)) = *agraph e* ∪ {*x*}×*X*
by (*unfold agraph-def*) (*auto split*: *split-if-asm*)

lemmas *agraph-ops* = *agraph-upd*

lemma *rgraph-mono*: *l*≤*l′* ⟹ *rgraph l* ⊆ *rgraph l′*
apply (*unfold rgraph-alt*)
apply *auto*
apply (*drule-tac x*=*b* **in** *le-funD*)
apply (*auto elim*: *le-optE*)
done

lemma *agraph-mono*: *e*≤*e′* ⟹ *agraph e* ⊆ *agraph e′*
by (*simp add*: *agraph-rgraph-converse rgraph-mono*)

An acquisition or release history is consistent, iff its graph is acyclic.

abbreviation *cons-rh* :: *′X ah* ⇒ *bool* **where** *cons-rh h* == *acyclic* (*rgraph h*)
abbreviation *cons-ah* :: *′X ah* ⇒ *bool* **where** *cons-ah h* == *acyclic* (*agraph h*)
abbreviation *cons-h* == *cons-rh*

As noted above, the cyclicity criterion is equivalent for a graph and its converse, such that we can use *cons-h* for both, acquisition and release histories.

lemma *cons-ah-rh-eq*:
cons-ah e = *cons-h e*
cons-rh r = *cons-h r*
by (*simp-all add*: *agraph-rgraph-converse*)

lemma *cons-h-empty*[*simp*]: *cons-h empty*
apply (*unfold rgraph-def*)
apply *auto*
apply (*metis Collect-def wfP-acyclicP wfP-empty*)
done

lemma *cons-h-add*:
⟦*dom h* ∩ *dom h′* = {}; *cons-h* (*h*++*h′*)⟧ ⟹ *cons-h h*
⟦*dom h* ∩ *dom h′* = {}; *cons-h* (*h*++*h′*)⟧ ⟹ *cons-h h′*
by (*auto dest*: *acyclic-union simp add*: *rgraph-add-union*)

lemma *cons-h-antimono*: ⟦$l \leq l'$; *cons-h* l'⟧ $\Longrightarrow$ *cons-h* l
  using *acyclic-subset*[*OF* - *rgraph-mono*] **.**

lemma *cons-h-update*:
  assumes *A*: *cons-h h* $X \cap$ *insert x* (*dom h*) = {}
  shows *cons-h* ($h(x \mapsto X)$)
proof –
  have *l-remove h x* $\leq$ *h* (**is** *?h* $\leq$ -) **by** *auto*
  with *cons-h-antimono A*(*1*) **have** *CONS*: *cons-h ?h* **by** *blast*
  have *MND*[*simp*]: $x \notin$ *dom ?h* **by** *auto*
  have [*simp*]: $h(x \mapsto X)$ = *?h*$(x \mapsto X)$ **by** (*auto simp add*: *l-remove-def intro*: *ext*)
  have *cons-h* (*?h*$(x \mapsto X)$) **proof** (*rule ccontr*, *erule cyclicE*)
    fix *y* **assume** $(y,y) \in$ (*rgraph* (*l-remove h x*$(x \mapsto X)$))$^+$
    hence $(y, y) \in$ (*rgraph* (*l-remove h x*) $\cup X \times \{x\})^+$ **by** (*simp add*: *rgraph-ops*)
    thus *False* **proof** (*cases rule*: *trancl-multi-insert*)
      case *orig* **with** *CONS* **show** *False* **by** (*auto simp add*: *acyclic-def*)
    next
      case (*via x'*) **hence** *C*: $(x,x') \in$(*rgraph ?h*)* **by** *auto*
      show *False* **using** *C* **proof** (*cases rule*: *rtrancl.cases*)
        case *rtrancl-refl* **with** *A*(*2*) *via*(*1*) **show** *False* **by** *auto*
      next
        case (*rtrancl-into-rtrancl* - *b*) **hence** $(b,x') \in$*rgraph ?h* **by** *auto*
        hence $x' \in$*dom ?h* **by** (*auto simp add*: *rgraph-def l-remove-def*)
        hence $x' \in$*dom h* **by** (*auto simp add*: *l-remove-def split*: *split-if-asm*)
        with *A*(*2*) *via*(*1*) **show** *False* **by** *auto*
      qed
    qed
  qed
  thus *?thesis* **by** *simp*
qed

lemma *cons-h-update2*:
  assumes *A*: *cons-h h* $x \notin$*dom h* $x \notin X$ $x \notin \bigcup$ *ran h*
  shows *cons-h* ($h(x \mapsto X)$)
proof –
  from *A*(*1*) **have** *A'*: *acyclic* (*agraph h*) **by** (*simp add*: *agraph-rgraph-converse*)
  from *A*(*4*) **have** *XNIR*: $x \notin$*Range* (*agraph h*) **by** *simp*
  hence [*simp*]: !!*y*. $\neg$ $(y,x) \in$(*agraph h*) **by** *blast*
  have *agraph* ($h(x \mapsto X)$) = *agraph h* $\cup \{x\} \times X$
    by (*simp add*: *agraph-ops*[*OF A*(*2*)])
  moreover have *acyclic* (*agraph h* $\cup \{x\} \times X$)
    apply (*rule ccontr*)
    apply (*erule cyclicE*)
  proof –
    fix *xa* **assume** (*xa*, *xa*) $\in$ (*agraph h* $\cup \{x\} \times X)^+$
    thus *False* **proof** (*cases rule*: *trancl-multi-insert2*)
      case *orig* **thus** *False* **using** *A'* **by** (*unfold acyclic-def*) *auto*

```
    next
      case (via xb) hence (xb,x)∈(agraph h)* by auto
      thus False proof (cases rule: rtrancl.cases)
        case rtrancl-refl
        with via(1) A(3) show False by auto
      next
        case (rtrancl-into-rtrancl a b c)
        hence (b,x)∈agraph h by simp
        thus False by simp
      qed
    qed
  qed
  ultimately have acyclic (agraph (h(x↦X))) by simp
  thus ?thesis by (simp add: agraph-rgraph-converse)
qed

lemma cons-h-remove: cons-h l ⟹ cons-h (l-remove l m)
  by (auto simp add: rgraph-ops intro: acyclic-subset)

lemma cons-h-add-use: ⟦m∉dom l; cons-h l⟧ ⟹ cons-h (l-add-use l {m})
  apply (rule ccontr)
  apply (erule cyclicE)
proof −
  fix x
  assume A: m ∉ dom l   cons-h l   (x, x) ∈ (rgraph (l-add-use l {m}))⁺
  from A(3) have (x,x)∈(rgraph l ∪ {m}×dom l)⁺ by (simp add: rgraph-ops)
  thus False
  proof (cases rule: trancl-multi-insert2)
    case orig
    with A(2) show False by (auto simp add: acyclic-def)
  next
    case (via xh) from via(2) show False
    proof (cases rule: rtrancl.cases)
      case rtrancl-refl
      hence [simp]: x=m by blast
      from via(3)[simplified] show False
      proof (cases rule: rtrancl.cases)
        case rtrancl-refl
        hence xh=m by blast
        with A(1) via(1) show False by simp
      next
        case rtrancl-into-rtrancl
        hence m∈dom l by (auto simp add: rgraph-def)
        with A(1) via(1) show False by simp
      qed
    next
      case rtrancl-into-rtrancl
      hence m∈dom l by (auto simp add: rgraph-def)
      with A(1) via(1) show False by simp
```

qed
qed
qed

lemma *cons-h-add-remove*: $cons\text{-}h\ l \Longrightarrow cons\text{-}h\ (l\text{-}add\text{-}use\ (l\text{-}remove\ l\ m)\ \{m\})$
by (*auto intro*: *cons-h-add-use cons-h-remove*)

lemma *cons-h-add-remove-partial*:
$[\![m \notin dom\ l1;\ cons\text{-}h\ (l1{+}{+}l2)]\!] \Longrightarrow$
$cons\text{-}h\ (l1\ {+}{+}\ l\text{-}add\text{-}use\ (l\text{-}remove\ l2\ m)\ \{m\})$
proof −
assume A: $m \notin dom\ l1$
hence
LE: $l1\ {+}{+}\ l\text{-}add\text{-}use\ (l\text{-}remove\ l2\ m)\ \{m\} \leq$
$l\text{-}add\text{-}use\ (l\text{-}remove\ (l1{+}{+}l2)\ m)\ \{m\}$
apply *simp*
apply (*rule map-add-first-le*)
apply (*simp add*: *l-add-use-le*)
done
assume $cons\text{-}h\ (l1{+}{+}l2)$
hence $cons\text{-}h\ (l\text{-}add\text{-}use\ (l\text{-}remove\ (l1{+}{+}l2)\ m)\ \{m\})$
by (*blast intro*: *cons-h-add-remove*)
with *cons-h-antimono*[*OF LE*] **show** *?thesis* **by** *blast*
qed

The consistency condition for acquisition structures checks available locks in addition to consistency of the acquisition and release histories.

fun *cons-as* :: $'X\ as \Rightarrow 'X\ set \Rightarrow bool$ **where**
$cons\text{-}as\ (l,u,e)\ \xi \longleftrightarrow$
$u \cap (\xi - dom\ l) = \{\} \wedge dom\ e \cap (\xi - dom\ l) = \{\} \wedge cons\text{-}h\ l \wedge cons\text{-}h\ e$

lemma *cons-as-antimono*: $[\![cons\text{-}as\ h\ \xi;\ \xi' \subseteq \xi]\!] \Longrightarrow cons\text{-}as\ h\ \xi'$
by (*cases h*) *auto*

fun *cons* **where**
$cons\ None\ X = False\ |$
$cons\ (Some\ (l,u,e))\ X = cons\text{-}as\ (l,u,e)\ X$

11.3.1 Minimal Elements

lemma *finite-acyclic-wf*: $[\![finite\ r;\ acyclic\ r]\!] \Longrightarrow wf\ r$
apply (*simp only*: *finite-wf-eq-wf-converse*[*symmetric*])
apply (*blast intro*: *finite-acyclic-wf-converse*)
done

The minimal elements of acquisition and release histories corresponds to those final acquisitions or initial releases that can safely be scheduled as next step — for an acquisition history without blocking any further locks usage and for a release history without requiring usage of already acquired locks.

abbreviation *rh-min l m* == *m*∈*dom l* ∧ *dom l* ∩ *the* (*l m*) = {}
abbreviation *ah-min e m* == *m*∈*dom e* ∧ *m*∉⋃ *ran e*

lemma *rh-min-alt*:
  rh-min l m = (*case l m of None* ⇒ *False* | *Some M* ⇒ *dom l* ∩ *M* = {})
  by (*fastsimp split*: *option.split-asm*)

There exists a minimal element in a consistent release history. Note that this lemma depends on the set of locks being finite, as assumed by the *LDPN* locale.

theorem (**in** *LDPN*) *cons-h-ex-rh-min*:
  fixes *l* :: *'X ah*
  assumes *A*: *l*≠*empty*   *cons-h l*
  shows ∃ *m*. *rh-min l m*
proof −
  {
    fix *M* **and** *mx*::*'X* **and** *k*
    assume ∀ *m*. ¬*rh-min l m*
    hence *B*: !!*m lm*. *l m* = *Some lm* ⟹ *dom l* ∩ *lm* ≠ {}
      by (*unfold rh-min-alt*) (*auto split*: *option.split-asm*)
    have ⟦ *card* (*UNIV*::*'X set*) − *card M* = *k*; *mx*∉*M*; *mx*∈*dom l*;
        !!*m*. *m*∈*M* ⟹ (*mx*,*m*)∈(*rgraph l*)$^+$
      ⟧ ⟹ *False*
    proof (*induct k arbitrary*: *M mx*)
      case *0* **hence** *M*=*UNIV* **by** *auto*
      with *0* **have** *False* **by** *simp*
      thus *?case* **..**
    next
      case (*Suc n*)
      then obtain *lmx* **where** *LMX*: *l mx* = *Some lmx* **by** *auto*
      with *B* **obtain** *m'* **where** *M'*: *m'*∈*dom l*   *m'*∈*lmx* **by** *blast*
      with *LMX* **have** *G*: (*m'*,*mx*)∈*rgraph l* **by** (*unfold rgraph-def*) *auto*
      {
        assume *m'*∈*M*
        with *Suc.prems* **have** (*mx*,*m'*)∈(*rgraph l*)$^+$ **by** *auto*
        also note *r-into-trancl*[*OF G*]
        finally have *False* **using** *A*(*2*) **by** (*unfold acyclic-def*) *auto*
      } **moreover** {
        assume *C*: *m'*∉*M*   *m'*≠*mx* **hence** *C'*: *m'*∉*M*∪{*mx*} **by** *auto*
        with *Suc.prems*(*4*) *G* **have** *1*: !!*m*. *m*∈*M*∪{*mx*} ⟹ (*m'*,*m*)∈(*rgraph l*)$^+$

          by (*auto intro*: *r-into-trancl trancl-trans*)
        from *Suc.prems*(*1*,*2*) **have**
          2: *card* (*UNIV*::*'X set*) − *card* (*M*∪{*mx*}) = *n*
          by (*simp*)
        from *Suc.hyps*[*OF 2 C' M'*(*1*) *1*] **have** *False* **.**
      } **moreover** {
        assume *m'*=*mx*
        with *r-into-trancl*[*OF G*] **have** *False* **using** *A*(*2*)

by (*unfold acyclic-def*) *auto*
} **ultimately show** *False* **by** *blast*
qed
} **note** X=*this*
from *A* **obtain** *m* **where** $m \in dom\ l$ **by** (*subgoal-tac dom l* $\neq$ {}) (*blast*, *auto*)
with *X*[*of* {} - *m*] *A* **show** *?thesis* **by** − (*rule ccontr*, *auto*)
qed

There exists a minimal element in a consistent acquisition history.

Note that this lemma depends on the set of locks being finite, as constrained by the *LDPN* locale.

theorem (**in** *LDPN*) *cons-h-ex-ah-min*:
fixes *e* :: *'X ah*
assumes *A*: $e \neq empty$ *cons-h e*
shows $\exists$ *m*. *ah-min e m*
proof (*cases agraph e* = {})
case *True* **from** *A*(*1*) **obtain** *m* **where** $m \in dom\ e$ **by** (*blast elim*: *nempty-dom*)
moreover with *True* **have** $m \notin \bigcup ran\ e$ **by** (*auto simp add*: *agraph-def ran-def*)
ultimately show *?thesis* **by** *blast*
next
case *False*
from *A*(*2*) *cons-ah-rh-eq*(*1*)[*symmetric*, *of e*] **have** *cons-ah e* **by** *simp*
hence *WF*: *wf* (*agraph e*) **by** (*auto intro*: *finite-acyclic-wf*)
from *wf-min*[*of agraph e*, *OF WF False*] **obtain** *m* **where**
$m \in Domain\ (agraph\ e) - Range\ (agraph\ e)$ **.**
hence $m \in dom\ e$ $m \notin \bigcup ran\ e$ **by** (*auto simp add*: *agraph-def ran-def*)
thus *?thesis* **by** *blast*
qed

11.3.2 Well-Nestedness and Acquisition Structures

Only locks that are on the lock-stack can be initially released:

lemma *wn-t-dom-l-lower-μ*:
$[\![wn\text{-}t'\ t\ \mu;\ as\ t = Some\ (l,\ u,\ e)]\!] \Longrightarrow dom\ l \subseteq set\ \mu$
apply (*induct t arbitrary*: *μ l u e*)
apply *fastsimp*
apply (*case-tac L*)
apply *fastsimp*
apply (*auto split*: *option.split-asm list.split-asm split-if-asm*
simp add: *l-remove-def l-add-use-def*)
apply (*fastsimp*)
apply (*fastsimp*)
apply (*fastsimp*)
apply (*case-tac L*)
apply (*fastsimp elim*: *as-comp-SomeE*)+
done

lemmas *wn-dom-l-empty* = *wn-t-dom-l-lower-μ*[*of* - [], *simplified*]

lemma *wn-h-dom-l-lower-μ*:
  ⟦*wn-h h μ*; *ash h* = *Some* (*l*,*u*,*e*)⟧ ⟹ *dom l* ⊆ *locks-μ μ*
  apply (*induct h μ arbitrary*: *l u e rule*: *wn-h.induct*)
  apply *auto*
  apply (*force dest*: *wn-t-dom-l-lower-μ elim*!: *as-comp-SomeE*)
  done

Due to well-nestedness, if a lock x is left, all locks that are above this lock on the stack are left, too. This lemma expresses leaving a lock by means of the domain of the release-history. Moreover, the release histories of the locks released before are smaller or equal than the release history of x, and do not contain x.

lemma *wn-t-dom-l-stack*: ⟦*wn-t′ t μ*; *as t* = *Some* (*l*,*u*,*e*); *x*∈*dom l*⟧ ⟹
  ∃ *μ1 μ2*. *μ*=*μ1*@*x*#*μ2* ∧ *set μ1* ⊆ *dom l* ∧
    (∀ *x′*∈*set μ1*. *l x′* ≤ *l x* ∧
      (*case l x′ of None* ⇒ *True* | *Some lx′* ⇒ *x*∉*lx′* ∧ *x′*∉*lx′*)
)
proof (*induct t arbitrary*: *μ l u e x*)
  case *NLEAF* **thus** *?case* **by** *fastsimp*
next
  case (*NSPAWN lab ts t*)
  from *NSPAWN.prems*(*1*) **obtain** *nlab* **where** [*simp*]: *lab*=*LNone nlab*
    by (*cases lab*, *simp-all*)
  from *NSPAWN.prems*(*1*) **have** *WN*: *wn-t′ ts* []   *wn-t′ t μ* **by** *auto*
  from *NSPAWN.prems*(*2*) **have** *as ts* ∥ *as t* = *Some* (*l*,*u*,*e*) **by** *simp*
  then obtain *l1 u1 e1 l2 u2 e2* **where**
    [*simp*]: *l*=*l1*++*l2*   *u*=*u1*∪*u2*   *e*=*e1*++*e2* **and**
      SPLIT: *as ts* = *Some* (*l1*,*u1*,*e1*)   *as t* = *Some* (*l2*,*u2*,*e2*)
        dom l1 ∩ *dom l2* = {}   *dom e1* ∩ *dom e2* = {}
    by (*blast elim*!: *as-comp-SomeE*)
  have [*simp*]: *l1* = *empty* **proof** −
    {
      fix *x* **assume** *A*: *x*∈*dom l1*
      from *NSPAWN.hyps*(*1*)[*OF WN*(*1*) *SPLIT*(*1*) *A*] **have** *False* **by** *blast*
    }
    thus *?thesis* **by** *force*
  qed
  from ‹*x*∈*dom l*› **have** *A*: *x*∈*dom l2* **by** *auto*
  from *NSPAWN.hyps*(*2*)[*OF WN*(*2*) *SPLIT*(*2*) *A*] **obtain** *μ1 μ2* **where**
    μ=*μ1*@*x*#*μ2*   *set μ1* ⊆ *dom l*
    ∀ *x′*∈*set μ1*. *l x′* ≤ *l x* ∧
      (*case l x′ of None* ⇒ *True* | *Some lx′* ⇒ *x* ∉ *lx′* ∧ *x′*∉*lx′*)
    by *auto*
  thus *?case* **by** *blast*
next
  case (*NNOSPAWN lab t*)
  show *?case* **proof** (*cases lab*)
    case (*LNone nlab*) **with** *NNOSPAWN* **show** *?thesis* **by** *simp blast*
  next

```
case (LAcq x')[simp]
from NNOSPAWN.prems(2) obtain l' u' e' where
  HTFMT: as t = Some (l',u',e')
  by (auto split: option.split-asm list.split-asm split-if-asm prod.split-asm)
with NNOSPAWN.prems(2,3) have MNE: x≠x'
  by (auto split: split-if-asm simp add: l-remove-def l-add-use-def)
from NNOSPAWN.prems(1) have WN: wn-t' t (x'#μ) by simp
{
  assume x'∈dom l'
  with NNOSPAWN.prems(2) HTFMT have
    [simp]: l=l-add-use (l-remove l' x') {x'}     u = insert x' u'     e'=e
    by (auto split: option.split-asm list.split-asm split-if-asm prod.split-asm)
  with MNE NNOSPAWN.prems(3) have MID: x∈dom l' by auto
  from NNOSPAWN.hyps[OF WN HTFMT MID] obtain μ1 μ2 where
    IHAPP: x'#μ = μ1@x#μ2     set μ1 ⊆ dom l'
          ∀x'∈set μ1. l' x' ≤ l' x ∧
             (case l' x' of None ⇒ True | Some lx' ⇒ x ∉ lx' ∧ x'∉lx')
    by blast
  from IHAPP(3) MNE have
    IHAPP3': ∀x'∈set μ1. l x' ≤ l x ∧
               (case l x' of None ⇒ True | Some lx' ⇒ x ∉ lx' ∧ x'∉lx')

    apply safe
    apply (case-tac x'=x'a)
    apply (simp add: l-add-use-def)
    apply (subgoal-tac l' x'a ≤ l' x)
    apply (erule le-optE)
    apply (simp add: l-add-use-def split: option.split)
    apply (auto simp add: l-add-use-def split: option.split) [1]
    apply simp
    apply (simp add: l-add-use-def l-remove-def)
    apply (split option.split-asm option.split)+
    apply meson
    apply fast+
    done
  from IHAPP(2) MNE have IHAPP2': l' x ≤ l x
    by (auto simp add: l-add-use-def split: option.split)
  from wnt-eq-wnt' WN wnt-distinct have distinct (x'#μ) by blast
  with MNE IHAPP IHAPP3' obtain μ1' where
    μ=μ1'@x#μ2     set μ1' ⊆ dom l
    ∀x'∈set μ1'. l x' ≤ l x ∧
      (case l x' of None ⇒ True | Some lx' ⇒ x ∉ lx' ∧ x'∉lx')
    by (cases μ1) auto
  hence ?thesis by blast
} moreover {
  assume A: x'∉dom l'
  with NNOSPAWN.prems(2) HTFMT have [simp]: l=l'
    by (auto split: split-if-asm)
 from NNOSPAWN.hyps[OF WN HTFMT NNOSPAWN.prems(3)[simplified]]
```

obtain $\mu 1$ $\mu 2$ **where** *IHAPP*: $x'\#\mu = \mu 1@x\#\mu 2$ *set* $\mu 1 \subseteq$ *dom* l'
  by *blast*
with *MNE* **have** $x'\in$*dom* l' **by** (*cases* $\mu 1$) *auto*
with *A* **have** *False* **..**
} **ultimately show** *?thesis* **by** *blast*
next
case (*LRel* x') [*simp*]
from *NNOSPAWN.prems*(*1*) **obtain** μ' **where** *WN*: $\mu=x'\#\mu'$ *wn-t'* t μ'
  by *auto*
from *NNOSPAWN.prems*(*2*) **obtain** l' u' **where**
  HTFMT: *as* t = *Some* (l',u',e) **and**
  [*simp*]: $l=l'(x'\mapsto\{\})$ $u=u'$
  by (*auto split*: *option.split-asm prod.split-asm list.split-asm*)
{
  assume $x=x'$
  with *WN*(*1*) **have** $\mu=[]@x\#\mu'$ *set* [] $\subseteq$ *dom* l
    ($\forall x'\in$*set* []. l x' $\leq$ l x $\wedge$
      (*case* l x' *of None* $\Rightarrow$ *True* | *Some* lx' $\Rightarrow$ $x \notin lx'$ $\wedge$ $x'\notin lx'$))
    by *auto*
  hence *?thesis* **by** *blast*
} **moreover** **{**
  assume *MNE*: $x\neq x'$
  with *NNOSPAWN.prems*(*3*) **have** *MIDL'*: $x\in$*dom* l'
    by (*auto simp add*: *l-add-use-def split*: *option.split-asm*)
  with *NNOSPAWN.hyps*[*OF WN*(*2*) *HTFMT*] **obtain** $\mu 1$ $\mu 2$ **where**
    IHAPP: $\mu'=\mu 1@x\#\mu 2$ *set* $\mu 1 \subseteq$ *dom* l'
      ($\forall x'\in$*set* $\mu 1$. l' x' $\leq$ l' x $\wedge$
        (*case* l' x' *of None* $\Rightarrow$ *True* | *Some* lx' $\Rightarrow$ $x \notin lx'$ $\wedge$ $x'\notin lx'$))
    by *blast*
  with *WN*(*1*) **have** $\mu=(x'\#\mu 1)@x\#\mu 2$ **by** *simp*
  moreover from *IHAPP*(*2*) *NNOSPAWN.prems*(*3*) **have**
    set ($x'\#\mu 1$) $\subseteq$ *dom* l
    by *auto*
  moreover from *IHAPP*(*3*) *MNE MIDL'* **have**
    ($\forall x'\in$*set* ($x'\#\mu 1$). l x' $\leq$ l x $\wedge$
      (*case* l x' *of None* $\Rightarrow$ *True* | *Some* lx' $\Rightarrow$ $x \notin lx'$ $\wedge$ $x'\notin lx'$))
    by (*fastsimp simp add*: *l-add-use-def split*: *option.split*)
  ultimately have *?thesis* **by** *blast*
} **ultimately show** *?thesis* **by** *blast*
qed
qed

lemma *wn-t-dom-l-stack'*: ⟦*wn-t'* t μ; *as* t = *Some* (l,u,e); $x\in$*dom* l⟧ $\Longrightarrow$
  $\exists \mu 1$ $\mu 2$. $\mu=\mu 1@x\#\mu 2$ $\wedge$ *set* $\mu 1 \subseteq$ *dom* l $\wedge$
    ($\forall x'\in$*set* $\mu 1$. l x' $\leq$ l x $\wedge$ $x\notin$*the* (l x') $\wedge$ $x'\notin$*the* (l x'))
apply (*drule* (*2*) *wn-t-dom-l-stack*)
apply (*elim exE*)

apply (*rule-tac x=μ1* **in** *exI*)
apply (*rule-tac x=μ2* **in** *exI*)
apply (*force*)
done

11.4 Soundness of the Consistency Condition

context *LDPN*
begin

The consistency condition for acquisition structures is sound, i.e. if a hedge h is schedulable with initial locks X, and is well-nested w.r.t. a lock stack list μ containing the locks from X, then the acquisition structure of h is consistent w.r.t. X.

theorem *acqh-sound*:
⟦ *lsched h X w*; *wn-h h μ*; *X=locks-μ μ* ⟧ ⟹
∃ *l u e*. *ash h* = *Some* (*l*,*u*,*e*) ∧ *cons-as* (*l*,*u*,*e*) (*locks-μ μ*)
— The proof works by induction over the schedule, in each induction step prepending a step to teh schedele.
For steps that have perform operation on locks, the proof is straightforward.
If the first step of the execution is a release of a lock, the acquisition history of the new hedge (with prepended release step at one tree) remains consistent. Acyclicity is preserved, as the release-step is the first step of the execution. Consistency w.r.t. used locks is also preserved.
If the first step of the execution is an acquisition step, we further have to distinguish whether it is a usage or a final acquisition.
proof (*induct arbitrary*: *μ rule*: *lsched.induct*)
case *lsched-final* **thus** *?case* **by** (*auto simp add*: *ash-final*)
next
case (*lsched-spawn h1 ts t h2 Xh w X lab μ*)
note [*simp*] = *lsched-spawn.prems*(*2*)
from *lsched-spawn.prems* **obtain** *nlab* **where** [*simp*]: *lab=LNone nlab*
by (*auto elim*: *wn-h-spawn-imp-LNoneE*)
from *lsched-spawn.hyps*(*3*) **have** [*simp*]: *Xh=X* **by** *auto*
from *wn-h-preserve-spawn*[*OF - lsched-spawn.prems*(*1*), *of X*, *simplified*]
obtain *μ′* **where** [*simp*]: *locks-μ μ* = *locks-μ μ′* *wn-h* (*h1@ts#t#h2*) *μ′*
by *blast*
from *lsched-spawn.hyps*(*2*)[*of μ′*, *simplified*] **obtain** *l u e* **where**
ash (*h1@ts#t#h2*) = *Some* (*l*,*u*,*e*) *cons-as* (*l*,*u*,*e*) (*locks-μ μ*)
by *auto*
moreover hence *ash* (*h1@NSPAWN lab ts t#h2*) = *Some* (*l*,*u*,*e*) **by** *simp*
ultimately show *?case* **by** *auto*
next
case (*lsched-nospawn h1 t h2 Xh w X lab μ*) **note** *lsched-nospawn.prems*(*2*)[*simp*]
from *wn-h-split-nospawn*[*OF lsched-nospawn.hyps*(*3*)[*simplified*]
lsched-nospawn.prems(*1*)] **obtain** *μ1 xs μ2 xsh* **where**
[*simp*]: *μ* = *μ1* @ *xs* # *μ2* *Xh* = *locks-μ μ1* ∪ *set xsh* ∪ *locks-μ μ2* **and**
LVX: *lock-valid-xs lab xs xsh* **and**
WNSPLIT: *wn-t′* (*NNOSPAWN lab t*) *xs* *wn-t′ t xsh*

wn-h h1 μ1   *wn-h h2 μ2* **and**
LDIST: *locks-μ μ1 ∩ set xs = {}*   *locks-μ μ1 ∩ set xsh = {}*
locks-μ μ1 ∩ locks-μ μ2 = {}   *locks-μ μ2 ∩ set xs = {}*
locks-μ μ2 ∩ set xsh = {} **and**
WNH: *wn-h (h1 @ t # h2) (μ1 @ xsh # μ2)*
.
have *WNHR*: *wn-h (h1@h2) (μ1@μ2)* **using** *WNSPLIT LDIST*
by (*auto intro*: *wn-h-appendI*)

from *lsched-nospawn.hyps(2)[OF WNH]* **obtain** *l u e* **where**
IHAPP: *ash h1 ∥ as t ∥ ash h2 = Some (l,u,e)*
cons-as (l,u,e) (locks-μ μ1 ∪ set xsh ∪ locks-μ μ2) **and**
IHAPP': *ash (h1 @ t # h2) = Some (l, u, e)*
by (*auto simp add*: *Un-ac*)
then obtain *lt ut et l2 u2 e2* **where**
[*simp*]: *as t = Some (lt,ut,et)*   *(ash h1 ∥ ash h2) = Some (l2,u2,e2)*
l=lt++l2   *u=ut∪u2*   *e=et++e2* **and**
ASS: *dom lt ∩ dom l2 = {}*   *dom et ∩ dom e2 = {}*
proof −
from *IHAPP* **have** *as t ∥ ash h1 ∥ ash h2 = Some (l,u,e)* **by** *simp*
thus *?thesis* **by** (*erule-tac as-comp-SomeE*) (*rule that*)
qed
from *wn-h-dom-l-lower-μ[OF WNHR]* **have**
DOML2: *dom l2 ⊆ locks-μ μ1 ∪ locks-μ μ2*
by *fastsimp*
from *wn-t-dom-l-lower-μ[OF WNSPLIT(2)]* **have**
DOMLT: *dom lt ⊆ set xsh*
by *fastsimp*
have *DOMDISJ*: *dom lt ∩ dom l2 = {}*
proof −
from *LDIST* **have** *set xsh ∩ (locks-μ μ1 ∪ locks-μ μ2) = {}* **by** *blast*
with *DOMLT DOML2* **show** *?thesis* **by** *blast*
qed
show *?case* **proof** (*cases lab*)
case (*LNone nlab*)[*simp*] **from** *LVX* **have** [*simp*]: *set xsh = set xs*
by (*auto elim*: *lock-valid-xs.cases*)
from *IHAPP* **show** *?thesis* **by** *auto*
next
case (*LRel x*)[*simp*]
from *LVX* **have** [*simp*]: *xs=x#xsh* **by** (*auto elim*: *lock-valid-xs.cases*)
have *ash (h1@(NNOSPAWN lab t)#h2) =*
as (NNOSPAWN lab t) ∥ Some (l2,u2,e2)
apply (*simp del*: *LRel*)
apply (*subst as-comp-acz.assoc[symmetric]*)
by (*simp*)
also from *IHAPP* **have** *as (NNOSPAWN lab t) = Some (lt(x↦{}),ut,et)*
by *simp*
hence *as (NNOSPAWN lab t) ∥ Some (l2,u2,e2) = Some (l(x↦{}),u,e)*
using *ASS DOML2 LDIST* **by** (*auto simp add*: *map-add-comm*)

finally have
G1: *ash* (*h1*@(*NNOSPAWN lab t*)#*h2*) = *Some* (*l*(*x*↦{}),*u*,*e*) **.**
moreover from *IHAPP*(*2*) **have** *G2*: *cons-as* (*l*(*x*↦{}),*u*,*e*) (*locks-μ μ*)
by *simp* (*blast intro*: *cons-h-update*[**where** *X*={}, *simplified*])
ultimately show *?thesis* **by** *blast*
next
case (*LAcq x*)[*simp*]
from *LVX* **have**
[*simp*]: *xsh*=*x*#*xs* **and**
XNIXS: *x*∉*set xs*
by (*auto elim*: *lock-valid-xs.cases*)
from *DOML2* **have** *XNIDL2*: *x*∉*dom l2* **using** *LDIST* **by** *auto*
show *?thesis* **proof** (*cases x*∈*dom lt*)
case *True* — The first step enters a lock that is left again, thus converting an initial release to a use step
— The consistency of the acquisition structure is preserved, as a use-step of a lock is added that is not initially released (any more)
have *ash* (*h1*@(*NNOSPAWN lab t*)#*h2*) =
as (*NNOSPAWN lab t*) ∥ *Some* (*l2*,*u2*,*e2*)
apply (*simp del*: *LAcq*)
apply (*subst as-comp-acz.assoc*[*symmetric*])
by (*simp*)
also from *True* **have**
as (*NNOSPAWN lab t*) =
Some (*l-add-use* (*l-remove lt x*) {*x*},*insert x ut*,*et*)
by *simp*
hence *as* (*NNOSPAWN lab t*) ∥ *Some* (*l2*,*u2*,*e2*) =
Some (*l2* ++ *l-add-use* (*l-remove lt x*) {*x*},*insert x u*,*e*)
using *ASS DOML2 LDIST*
by (*auto simp add*: *map-add-comm*)
finally have *G1*: *ash* (*h1*@(*NNOSPAWN lab t*)#*h2*) =
Some (*l2* ++ *l-add-use* (*l-remove lt x*) {*x*},*insert x u*,*e*) **.**
moreover
have *G2*: *cons-as* (*l2* ++ *l-add-use* (*l-remove lt x*) {*x*},*insert x u*,*e*)
(*locks-μ μ*)
proof −
from *IHAPP*(*2*) **have** *cons-h* (*l2* ++ *l-add-use* (*l-remove lt x*) {*x*})
using *cons-h-add-remove-partial*[*OF XNIDL2*, *of lt*]
by (*simp add*: *map-add-comm*[*OF DOMDISJ*])
moreover have
insert x u ∩
(*locks-μ μ* − *dom* (*l2* ++ *l-add-use* (*l-remove lt x*) {*x*})) = {}
using *XNIXS LDIST*[*simplified*] *IHAPP*(*2*) **by** *simp blast*
moreover have
dom e ∩ (*locks-μ μ* − *dom* (*l2* ++ *l-add-use* (*l-remove lt x*) {*x*})) = {}
using *XNIXS LDIST*[*simplified*] *IHAPP*(*2*) **by** *simp blast*
moreover from *IHAPP*(*2*) **have** *cons-h e* **by** *simp*
ultimately show *?thesis* **by** *simp*
qed

ultimately show *?thesis* **by** *blast*
next
case *False* — The first step finally enters a lock
from *False XNIDL2 IHAPP(2)* **have** *XNIUE*: $x \notin u \quad x \notin dom\ e$ **by** *auto*
— The consistency of the acquisition structure is preserved, as no cycles are added by insertion of the final acquisition.
have *ash (h1@(NNOSPAWN lab t)#h2) =*
as (NNOSPAWN lab t) ‖ Some (l2,u2,e2)
apply *(simp del: LAcq)*
apply *(subst as-comp-acz.assoc[symmetric])*
by *(simp)*
also from *False* **have** *as (NNOSPAWN lab t) = Some (lt,ut,et(x↦ut))*
using *XNIUE* **by** *simp*
hence *as (NNOSPAWN lab t) ‖ Some (l2,u2,e2) = Some (l,u,e(x↦ut))*
using *ASS XNIUE*
by *(auto simp add: map-add-comm)*
finally have
G1: ash (h1@(NNOSPAWN lab t)#h2) = Some (l,u,e(x↦ut)) **.**
moreover
from *cons-h-update2[of e x ut] IHAPP(2) ash-le-u[OF IHAPP′] XNIUE*
have *cons-h (e(x↦ut))* **by** *auto*
with *IHAPP(2)* **have** *cons-as (l,u,e(x↦ut)) (locks-μ μ)*
using *LDIST XNIXS* **by** *simp blast*
ultimately show *?thesis* **by** *blast*
qed
qed
qed
end

11.5 Precision of the Consistency Condition

11.5.1 Custom Size Function

In the following we construct a custom size function for hedges that is suited to do induction over hedges. This size function decreases on any step done on the hedge.

fun *list-size′* **where**
list-size′ f [] = (0::nat) |
list-size′ f (a#l) = f a + list-size′ f l

fun *size-t* **where**
size-t (NLEAF π) = Suc 0 |
size-t (NNOSPAWN lab t) = Suc (size-t t) |
size-t (NSPAWN lab ts t) = Suc (size-t ts + size-t t)

lemma *list-size′-conc[simp]: list-size′ f (a@b) = list-size′ f a + list-size′ f b*
by *(induct a) auto*

abbreviation *hedge-size* :: *('P,'T,'L,'X) lex-hedge ⇒ nat* **where**

hedge-size h == list-size′ size-t h

lemma *hedge-size-zero*[*simp*]: *hedge-size h = 0* ⟷ *h*=[]
  apply (*cases h*)
  apply *auto*
  apply (*case-tac a*)
  apply *simp-all*
done

This function checks whether a lock is released in the current execution tree, and returns the set of locks that are acquired before this lock is released. Note that this function ignores the lock-effect of labels of spawn-nodes, as we assume that spawn-nodes have no lock-operation.

fun *closing* :: *′X* ⇒ (*′P*,T,*′L*,*′X*) *lex-tree* ⇒ *′X set option* **where**
  closing x (*NLEAF* π) = *None* |
  closing x (*NSPAWN lab ts t*) = *closing x t* |
  closing x (*NNOSPAWN* (*LNone nlab*) *t*) = *closing x t* |
  closing x (*NNOSPAWN* (*LAcq x′*) *t*) = (
    case closing x t of None ⇒ *None* |
                Some X ⇒ *Some* (*insert x′ X*)
) |
  closing x (*NNOSPAWN* (*LRel x′*) *t*) = (*if x=x′ then Some* {} *else closing x t*)

Function that checks whether a tree starts with the acquisition of a lock that is used (i.e. not finally acquired) and returns all the locks that are used from the acquisition to to the release of that lock:

fun *closing′* **where**
  closing′ (*NNOSPAWN* (*LAcq x*) *t*) = *closing x t* |
  closing′ - = None

The following functions define the set of locks that are acquired at the roots of a tree/hedge. This function is used in the case of the precision proof, where all the roots of the hedge are either leafs or final acquisitions.

fun *rootlocks-t* **where**
  rootlocks-t (*NNOSPAWN* (*LAcq x*) *t*) = {*x*} |
  rootlocks-t - = {}
fun *rootlocks* **where**
  rootlocks [] = {} |
  rootlocks (*t # h*) = *rootlocks-t t* ∪ *rootlocks h*

lemma *rootlocks-conc*[*simp*]: *rootlocks* (*h1*@*h2*) = *rootlocks h1* ∪ *rootlocks h2*
  by (*induct h1*) *auto*

lemma *rootlocks-split*:
  ⟦ *x*∈*rootlocks h*; !!*h1 t h2*. *h*=*h1*@*NNOSPAWN* (*LAcq x*) *t*#*h2* ⟹ *P* ⟧ ⟹ *P*
proof (*induct h arbitrary*: *P*)
  case *Nil* **thus** *?case* **by** *simp*
next

case (*Cons tp h*) **from** *Cons.prems(1)[simplified]* **show** *?case* **proof**
assume $x \in$*rootlocks-t tp*
with *Cons.prems(2)[of [], simplified]* **show** *?thesis*
by (*cases tp rule: rootlocks-t.cases*) *auto*
next
assume *A*: $x \in$*rootlocks h* **from** *Cons.hyps[OF A]* **obtain** *h1 t h2* **where**
h = h1 @ NNOSPAWN (LAcq x) t # h2 **.**
hence *tp#h = (tp#h1)@NNOSPAWN (LAcq x) t # h2* **by** *simp*
thus *?thesis* **by** (*blast intro!: Cons.prems(2)*)
qed
qed

If a lock x is closed (before it is acquired), the value of the release history for x is precisely the set of used locks before x is closed. Closing x before it is acquired is expressed by well-nestedness w.r.t. a lock-stack that contains x.

lemma *closing-dom-l*:
⟦ *wn-t′ t (xs1@x#xs2)*; *closing x t = Some Xu*; *as t = Some (l,u,e)* ⟧ $\Longrightarrow$
l x = Some Xu
proof (*induct t arbitrary: xs1 l u e Xu*)
case *NLEAF* **thus** *?case* **by** *auto*
next
case (*NSPAWN lab ts t*)
then obtain *nlab* **where** [*simp*]: *lab=LNone nlab* **by** (*cases lab*) *auto*
from *NSPAWN* **show** *?case* **by** (*fastsimp elim: as-comp-SomeE dest: wn-dom-l-empty*)
next
case (*NNOSPAWN lab t*) **show** *?case* **proof** (*cases lab*)
case (*LNone nlab*) **with** *NNOSPAWN* **show** *?thesis* **by** *auto*
next
case (*LAcq x′*)[*simp*]
from *NNOSPAWN.prems* **obtain** *Xu′* **where**
HP1: *wn-t′ t ((x′#xs1)@x#xs2)*   *closing x t = Some Xu′* **and**
[*simp*]: *Xu=insert x′ Xu′*
by (*auto split: option.split-asm*)
from *NNOSPAWN.prems* **obtain** *l′ u′ e′* **where**
HP2: *as t = Some (l′,u′,e′)*
by (*auto split: eahl-splits*)
from *NNOSPAWN.hyps[OF HP1 HP2]* **have** *IHAPP*: *l′ x = Some Xu′* **.**

from *wn-t-dom-l-stack[OF HP1(1) HP2, of x]*
IHAPP distinct-match[OF wnt-distinct′[OF HP1(1)]] **have**
set (x′#xs1) $\subseteq$ *dom l′*
by *fastsimp*
hence *X′IDL′*: $x' \in$*dom l′* **by** *simp*
with *NNOSPAWN.prems(3) HP2 IHAPP*
have *l = l-add-use (l-remove l′ x′) {x′}* **by** (*simp split: eahl-splits*)
moreover from *wnt-distinct′[OF HP1(1)]* **have** *MNE*: $x' \neq x$ **by** (*auto*)
ultimately show *l x = Some Xu* **using** *IHAPP* **by** (*auto simp add: l-add-use-def*)
next

```
    case (LRel x')[simp]
    show ?thesis proof (cases x=x')
      case True with NNOSPAWN.prems have l x = Some {}    Xu={}
        by (auto split: eahl-splits)
      thus ?thesis by blast
    next
      case False with NNOSPAWN.prems obtain xs1' where
        [simp]: xs1=x'#xs1' and
          HP1: wn-t' t (xs1'@x#xs2)    closing x t = Some Xu
        by (cases xs1) auto
      from NNOSPAWN.prems obtain l' u' e' where
        HP2: as t = Some (l',u',e') and
        [simp]: l=l'(x'↦{})
        by (auto split: eahl-splits)
      from NNOSPAWN.hyps[OF HP1 HP2(1)] have l' x = Some Xu .
      with False show l x = Some Xu by auto
    qed
  qed
qed
```

A lock must not be used before it is closed.

```
lemma wn-closing-ni: ⟦wn-t' t (μ1@x#μ2); closing x t = Some Xu⟧ ⟹ x∉Xu
proof (induct t arbitrary: μ1 Xu)
  case NLEAF thus ?case by auto
next
  case (NSPAWN lab ts t)
  then obtain nlab where [simp]: lab=LNone nlab by (cases lab) auto
  from NSPAWN show ?case by auto
next
  case (NNOSPAWN lab t)
  show ?case proof (cases lab)
    case (LNone nlab) thus ?thesis using NNOSPAWN by auto
  next
    case (LAcq x')[simp]
    from NNOSPAWN.prems(1) have WN: wn-t' t ((x'#μ1)@x#μ2) by auto
    from NNOSPAWN.prems(2) obtain Xu' where
      CL: closing x t = Some Xu'    Xu = insert x' Xu'
      by (auto split: option.split-asm)
    from NNOSPAWN.hyps[OF WN CL(1)] have x∉Xu' .
    moreover from wnt-distinct'[OF WN] have x'≠x by auto
    ultimately show ?thesis by (auto simp add: CL(2))
  next
    case (LRel x')
    thus ?thesis
      using NNOSPAWN by (cases μ1) (auto split: split-if-asm)
  qed
qed
```

This lemma gives porperties of the acquisition structure after an acquisition step of a lock usage. It is used in the case when there is a tree starting with a usage, to reason about the acquisition structure after the root node of this tree has been scheduled.

lemma *wn-closing-as-fmt*:
  assumes *A*: *wn-t′* (*NNOSPAWN* (*LAcq x*) *t*) *μ*
    as (*NNOSPAWN* (*LAcq x*) *t*) = *Some* (*l*,*u*,*e*)
    closing x t = *Some Xu*
  assumes *C*: !!*l′ u′*. ⟦ *as t* = *Some* (*l′*,*u′*,*e*); *l′* ≤ *l*(*x*↦*Xu*);
    u=*insert x u′*; *dom l′* = *insert x* (*dom l*)
    ⟧ ⟹ *P*
  shows *P*
proof −
  from *A*(*1*) **have** *WN*: *wn-t′ t* ([]@*x*#*μ*) **by** *auto*
  from *A*(*2*) **obtain** *l′ u′ e′* **where** *AS′*: *as t* = *Some* (*l′*,*u′*,*e′*)
    by (*auto split*: *eahl-splits*)
  from *closing-dom-l*[*OF WN A*(*3*) *AS′*] **have** *L′X*: *l′ x* = *Some Xu* **.**
  with *A*(*2*) *AS′* **have**
    LFMT: *l* = *l-add-use* (*l-remove l′ x*) {*x*} **and**
    [*simp*]: *u*=*insert x u′* *e′*=*e*
    by (*auto split*: *eahl-splits*)
  from *LFMT L′X* **have** *G2*: *l′* ≤ *l*(*x*↦*Xu*)
    by (*rule-tac le-funI*) (*auto simp add*: *l-add-use-def split*: *option.split*)
  from *LFMT L′X* **have** *G3*: *dom l′* = *insert x* (*dom l*) **by** *auto*
  from *C*[*OF - G2 - G3*] **show** *P* **by** (*simp add*: *AS′*)
qed

A lock that occurs in the release history is closed in the execution tree, using the locks as described in the RH.

lemma *dom-l-closing*:
  ⟦ *as t* = *Some* (*l*,*u*,*e*); *wn-t′ t μ*; *l x* = *Some Xu* ⟧ ⟹ *closing x t* = *Some Xu*
proof (*induct t μ arbitrary*: *l u e Xu rule*: *wn-t′.induct*)
  case (*1 ms*) **thus** *?case* **by** *auto*
next
  case *2* **thus** *?case* **by** *force*
next
  case *3* **thus** *?case* **by** (*fastsimp elim*!: *as-comp-SomeE dest*!: *wn-dom-l-empty*)
next
  case (*4 xa t μ*) **note** *C*=*this*

  from *C*(*3*) **have** *WN*: *wn-t′ t* (*xa*#*μ*) **by** *auto*
  from *C*(*2*) **obtain** *l′ u′ e′* **where** *AS*: *as t* = *Some* (*l′*,*u′*,*e′*)
    by (*auto split*: *eahl-splits*)
  from *C*(*2*,*4*) **have** *XNE*: *xa*≠*x* **by** (*auto split*: *eahl-splits simp add*: *l-add-use-def*)
  with *AS C*(*2*,*4*) **obtain** *Xu′* **where** *P*: *l′ x* = *Some Xu′*
    by (*auto split*: *eahl-splits simp add*: *l-add-use-def*)
  from *C*(*1*)[*OF AS WN*, *OF P*] **have** *IHAPP*: *closing x t* = *Some Xu′* **.**
  from *wn-t-dom-l-stack′*[*OF WN AS*, *of x*] *P* **obtain** *μ1 μ2* **where**
    xa#*μ* = *μ1*@*x*#*μ2* *set μ1* ⊆ *dom l′*
    by *blast*
  with *XNE* **have** *xa*∈*dom l′* **by** (*cases μ1*) *auto*

with *AS C(2,4)* **have** *l = l-add-use (l-remove l′ xa) {xa}*
by *(auto split: eahl-splits)*
with *XNE P C(4)* **have** *Xu = (insert xa Xu′)* **by** *(auto simp add: l-add-use-def)*
moreover from *IHAPP*
have *closing x (NNOSPAWN (LAcq xa) t) = Some (insert xa Xu′)*
by *auto*
ultimately show *?case* **by** *blast*
next
case *5* **thus** *?case* **by** *(fastsimp split: eahl-splits)*
qed *auto*

If a tree starts with a final acquisition of x, its release history is empty and the acquisition history of x contains all the used locks.

With Lemma *as-ran-e-le-u* we then also have that the ranges of the acquisition histories contain precisely the used locks.

lemma *ncl-as-fmt-single*:
assumes *A*: *wn-t′ (NNOSPAWN (LAcq x) t) μ*
closing′ (NNOSPAWN (LAcq x) t) = None
as (NNOSPAWN (LAcq x) t) = Some (l,u,e)
shows *u=⋃ran e* *l=empty* *e x = Some u*
proof −
from *A(1)* **have** *WN*: *wn-t′ t (x#μ)* **by** *auto*
from *A(2)* **have** *NC*: *closing x t = None* **by** *auto*
from *A(3)* **obtain** *l′ u′ e′* **where** *AS*: *as t = Some (l′,u′,e′)*
by *(auto split: eahl-splits)*
from *dom-l-closing[OF AS WN] NC* **have** *XNIDL′*: *¬x∈dom l′* **by** *auto*
with *AS A(3)* **have**
EFMT: *e=e′(x↦u)* *x∉dom e′* **and**
[simp]: *l=l′*
by *(auto split: eahl-splits)*
from *EFMT(1)* **show** *e x = Some u* **by** *auto*
with *EFMT* **have** *u ⊆ ⋃ran e* **by** *auto*
with *as-ran-e-le-u[OF A(3)]* **show** *u=⋃ran e* **by** *simp*
{
fix *x′*
assume *CONTR*: *x′∈dom l′*
with *XNIDL′* **have** *XNE*: *x′≠x* **by** *auto*
from *wn-t-dom-l-stack′[OF WN AS CONTR]* **obtain** *μ1 μ2* **where**
DS: *x#μ = μ1@x′#μ2* *set μ1 ⊆ dom l′*
by *blast*
with *XNE* **have** *x∈dom l′* **by** *(cases μ1) auto*
with *XNIDL′* **have** *False* **..**
} **thus** *l=empty*
by *(auto simp add: dom-empty-simp[symmetric] simp del: dom-empty-simp)*
qed

This lemma describes properties of the acquisition structure of a tree after a final acquisition has been scheduled.

lemma *ncl-as-fmt-single′*:

assumes *A*: *wn-t′ (NNOSPAWN (LAcq x) t) μ*
closing′ (NNOSPAWN (LAcq x) t) = None
as (NNOSPAWN (LAcq x) t) = Some (l,u,e)
assumes *C*: !!*e′*. ⟦ *as t = Some (empty, u, e′)*;
$u = \bigcup ran\ e$; *l=empty*;
e = e′(x↦u); *x∉dom e′*
⟧ ⟹ *P*
shows *P*
proof –
from *A(1)* **have** *WN*: *wn-t′ t (x#μ)* **by** *auto*
from *A(2)* **have** *NC*: *closing x t = None* **by** *auto*
from *A(3)* **obtain** *l′ u′ e′* **where** *AS*: *as t = Some (l′,u′,e′)*
by (*auto split*: *eahl-splits*)
from *dom-l-closing[OF AS WN] NC* **have** *XNIDL′*: ¬*x*∈*dom l′* **by** *auto*
with *AS A(3)* **have**
EFMT: *e=e′(x↦u)*   *x∉dom e′* **and**
[*simp*]: *l′=l*   *u′=u*
by (*auto split*: *eahl-splits*)
with *EFMT* **have** $u \subseteq \bigcup ran\ e$ **by** *auto*
with *as-ran-e-le-u[OF A(3)]* **have** *UFMT*: $u = \bigcup ran\ e$ **by** *simp*
{
fix *x′*
assume *CONTR*: *x′*∈*dom l′*
with *XNIDL′* **have** *XNE*: *x′*≠*x* **by** *auto*
from *wn-t-dom-l-stack′[OF WN AS CONTR]* **obtain** *μ1 μ2* **where**
DS: *x#μ = μ1@x′#μ2*   *set μ1 ⊆ dom l′*
by *blast*
with *XNE* **have** *x*∈*dom l′* **by** (*cases μ1*) *auto*
with *XNIDL′* **have** *False* **..**
} **hence** *LFMT[simp]*: *l=empty*
by (*auto simp add*: *dom-empty-simp[symmetric] simp del*: *dom-empty-simp*)
from *C[OF - UFMT LFMT EFMT] AS* **show** *P* **by** *simp*
qed

The acquisition structure of a hedge whose trees start with final acquisitions or are leafs has a special structure:

- The release history is empty.
- The ranges of the acquisition histories contain precisely the used locks.
- The acquisition histories for the locks at the roots of the hedge contain precisely the used locks.
- The acquisistion histories are defined for the locks at the roots of the hedge.

The first proposition follows because an initial release cannot come after a final acquisition due to well-nestedness. The second and third propositions follow as the roots of the hedge preceed every other node in the hedge. The forth proposition follows directly from the assumption that every root node that acquired a lock is a final acquisistion.

```
lemma ncl-as-fmt:
  ⟦
    wn-h h μ; ash h = Some (l,u,e);
    !!Q t. ⟦ t∈set h; !!x t'. t=NNOSPAWN (LAcq x) t' ⟹ Q;
             !!p w. t=NLEAF (p,w) ⟹ Q
           ⟧ ⟹ Q;
    ∀ t∈set h. closing' t = None
  ⟧ ⟹ l=empty ∧ u=⋃ran e ∧
       ⋃ran (e |` rootlocks h) = ⋃ran e ∧
       rootlocks h ⊆ dom e
proof (induct h arbitrary: μ l u e)
  case Nil thus ?case by auto
next
  case (Cons t h)
  from Cons.prems(1) obtain xs μ' where
    [simp]: μ=xs#μ' and
      WN-SPLIT: wn-t' t xs    wn-h h μ' and
      WN-DISJ: set xs ∩ locks-μ μ' = {}
    by (auto elim!: wn-h-prepend-h)
  from Cons.prems(2) obtain l1 u1 e1 l2 u2 e2 where
    [simp]: l=l1++l2    u=u1∪u2    e=e1++e2 and
      AS-SPLIT: as t = Some (l1,u1,e1)    ash h = Some (l2,u2,e2) and
      AS-DISJ: dom l1 ∩ dom l2 = {}    dom e1 ∩ dom e2 = {}
    by (fastsimp elim!: as-comp-SomeE)
  have l2=empty ∧ u2=⋃ran e2 ∧
       ⋃ran (e2 |` rootlocks h) = ⋃ran e2 ∧ rootlocks h ⊆ dom e2
    apply (rule-tac Cons.hyps[OF WN-SPLIT(2) AS-SPLIT(2)])
    apply (rule-tac t=t in Cons.prems(3))
    apply auto
    apply (rule-tac Cons.prems(4)[rule-format])
    apply simp
    done
  hence IHAPP: l2=empty
             u2=⋃ran e2
             ⋃ran (e2 |` rootlocks h) = ⋃ran e2
             rootlocks h ⊆ dom e2
    by auto
  have t∈set (t#h) by simp
  thus ?case proof (cases rule: Cons.prems(3)[cases set, case-names acquire leaf])
   case leaf[simp] with AS-SPLIT(1) have [simp]: l1=empty    u1={}    e1=empty
by auto
    from IHAPP show ?thesis by simp
  next
    case (acquire x t')[simp]
    from ncl-as-fmt-single[of x t' xs l1 u1 e1] WN-SPLIT(1) AS-SPLIT(1)
        Cons.prems(4)[rule-format, of t] have
      P: l1=empty    u1=⋃ran e1    e1 x = Some u1
```

by *auto*
from *P IHAPP AS-DISJ* **have** *G1*: *l=empty* ∧ *u=*⋃ *ran e* **by** *auto*
from *P(3)* **have** *G2-1*: *rootlocks-t t* ⊆ *dom e1* **by** *auto*
from *P(2,3)* **have** *G3-1*: ⋃ *ran* (*e1* |‘ *rootlocks-t t*) = ⋃ *ran e1*
by (*auto simp add*: *restrict-map-def ran-def*)
from *G2-1 IHAPP(4) AS-DISJ* **have**
⋃ *ran* ((*e1* ++ *e2*) |‘ (*rootlocks-t t* ∪ *rootlocks h*)) = ⋃ *ran e1* ∪ ⋃ *ran e2*
by (*rule-tac union-ran-add-aux*[*OF G3-1 IHAPP(3)*]) *auto*
hence *G3*: ⋃ *ran* (*e*|‘ *rootlocks* (*t#h*)) = ⋃ *ran e* **using** *AS-DISJ* **by** *auto*
show *?thesis* **using** *G1 G2-1 IHAPP(4) G3* **by** *auto*
qed
qed

This lemma makes explicit the case-distinction along which the precision proof is done. The cases are:

final All trees are leaf nodes.

spawn There is a tree starting with a *NSPAWN x* - node.

none There is a tree starting with a *NNOSPAWN LNone* - node.

release There is a tree starting with a *NNOSPAWN* (*LRel x*)-node.

acquire All trees start with a *NNOSPAWN* (*LAcq x*)-node or are leafs. At least one tree is no leaf.

lemma *h-cases*[*case-names final spawn none release acquire*]:
assumes *C*:
final h ⟹ *P*
!!*h1 lab ts t h2*. *h=h1@NSPAWN lab ts t#h2* ⟹ *P*
!!*h1 t nlab h2*. *h=h1@NNOSPAWN* (*LNone nlab*) *t#h2* ⟹ *P*
!!*h1 x t h2*. *h=h1@NNOSPAWN* (*LRel x*) *t#h2* ⟹ *P*
⟦ !!*Q t*. ⟦ *t*∈*set h*; !!*x t′*. *t=NNOSPAWN* (*LAcq x*) *t′* ⟹ *Q*;
!!*p w*. *t=NLEAF* (*p*,*w*) ⟹ *Q*
⟧ ⟹ *Q*;
!!*Q*. ⟦ !!*t′ x*. *NNOSPAWN* (*LAcq x*) *t′* ∈ *set h* ⟹ *Q* ⟧ ⟹ *Q*
⟧ ⟹ *P*
shows *P*
proof (*cases h*=[])
case *True* **with** *C(1)* **show** *P* **by** *simp*
next
case *False* **hence** *set h* ≠ {} **by** *simp*
{
assume ∃ *t nlab*. *NNOSPAWN* (*LNone nlab*) *t*∈*set h*
with *C(3)* **have** *P* **by** (*blast elim*: *in-set-list-format*)
} moreover {
assume ∃ *t x*. *NNOSPAWN* (*LRel x*) *t*∈*set h*
with *C(4)* **have** *P* **by** (*blast elim*: *in-set-list-format*)
} moreover {

assume ∃ *lab ts t. NSPAWN lab ts t*∈*set h*
with *C*(*2*) **have** *P* **by** (*blast elim*: *in-set-list-format*)
} **moreover** **{**
assume ∀ *t*∈*set h*. ¬(∃ *lab t. NNOSPAWN lab t*∈*set h*) ∧
¬ (∃ *lab ts t. NSPAWN lab ts t*∈*set h*)
hence ∀ *t*∈*set h. final-t t*
apply *safe*
apply (*case-tac t*)
apply *auto*
done
with *C*(*1*) **have** *P* **by** (*auto simp add*: *list-all-iff*)
} **moreover** **{**
assume *A*: ¬(∃ *t nlab. NNOSPAWN* (*LNone nlab*) *t*∈*set h*)
¬(∃ *t x. NNOSPAWN* (*LRel x*) *t*∈*set h*)
¬(∃ *lab ts t. NSPAWN lab ts t*∈*set h*)
(∃ *lab t. NNOSPAWN lab t*∈*set h*)
hence (∃ *t x. NNOSPAWN* (*LAcq x*) *t*∈*set h*)
apply *auto*
apply (*case-tac lab*)
by *auto*
with *A*(*1*,*2*,*3*) **have** *P* **apply** *auto*
apply (*rule-tac C*(*5*))
apply *auto*
apply (*case-tac ta*)
apply *auto*
apply *fast*
apply (*case-tac L*)
apply *auto*
apply *fast*
done
} **ultimately show** *?thesis* **by** *blast*
qed

This lemma determines the tree within a hedge whose release history contains a specific lock.

lemma *ash-find-l-t*[*consumes 2*]:
⟦ *ash h* = *Some* (*l*,*u*,*e*); *x*∈*dom l*;
!!*h1 t h2 l1 u1 e1 l2 u2 e2*. ⟦
h=*h1*@*t*#*h2*; *l*=*l1*++*l2*; *u*=*u1*∪*u2*; *e*=*e1*++*e2*;
as t = *Some* (*l1*,*u1*,*e1*); *ash h1* ∥ *ash h2* = *Some* (*l2*,*u2*,*e2*);
x∈*dom l1*; *dom l1*∩*dom l2* = {}; *dom e1*∩*dom e2* = {}
⟧ ⟹ *P*
⟧ ⟹ *P*
proof (*induct h arbitrary*: *l u e P rule*: *ash.induct*)
case *1* **thus** *?case* **by** *fastsimp*
next
case (*2 t h*) **note** *C*=*this*
from *as-comp-SomeE*[*OF C*(*2*)[*simplified*]] **obtain** *l1 u1 e1 l2 u2 e2* **where**
SPLIT-simps[*simp*]: *l* = *l1* ++ *l2*   *u* = *u1* ∪ *u2*   *e* = *e1* ++ *e2* **and**

```
    SPLIT: as t = Some (l1, u1, e1)    ash h = Some (l2, u2, e2)
        dom l1 ∩ dom l2 = {}    dom e1 ∩ dom e2 = {}
    .
  from C(3) have x∈dom l1 ∨ x∈dom l2 by auto
  moreover {
    assume A: x∈dom l1
    moreover have t#h = []@t#h by simp
    ultimately have ?case
      by (rule-tac C(4)) (assumption, (simp add: SPLIT)+)
  } moreover {
    assume A: x∈dom l2
    from C(1)[OF SPLIT(2) A] obtain h1 tt h2 l21 u21 e21 l22 u22 e22 where
      IHAPP-simp[simp]: h = h1 @ tt # h2      l2=l21++l22
                 u2=u21∪u22              e2=e21++e22 and
      IHAPP: as tt = Some (l21, u21, e21)
          ash h1 ∥ ash h2 = Some (l22,u22,e22)
          x∈dom l21
          dom l21∩dom l22={}
          dom e21∩dom e22={}
      .
    from SPLIT IHAPP have
      DS: dom l1 ∩ dom l21 = {}    dom e1 ∩ dom e21 = {}
      by auto
    have t#h=(t#h1)@tt#h2      l = l21 ++ (l1++l22)
       u=u21∪(u1∪u22)       e=e21 ++ (e1++e22)
      by (auto simp add: map-add-comm[OF DS(1)] map-add-comm[OF DS(2)])
    moreover have ash (t#h1) ∥ ash h2 = Some (l1++l22,u1∪u22,e1++e22)
    proof −
      have ash (t#h1) ∥ ash h2 = as t ∥ (ash h1 ∥ ash h2) by (simp)
      also have ... = as-comp (l1,u1,e1) (l22,u22,e22)
        by (simp add: IHAPP(2) SPLIT(1))
      also have ... = Some (l1++l22,u1∪u22,e1++e22)
        using SPLIT IHAPP by auto
      finally show ?thesis .
    qed
    ultimately have ?case using SPLIT(3,4) IHAPP(1,3,4,5)
      by (rule-tac C(4)) (assumption+, auto)
  } ultimately show ?case by blast
qed
```

This lemma determines the tree within a hedge whose acquisition history contains a specific lock.

```
lemma ash-find-e-t[consumes 2]:
  ⟦ ash h = Some (l,u,e); x∈dom e;
    !!h1 t h2 l1 u1 e1 l2 u2 e2. ⟦
      h=h1@t#h2; l=l1++l2; u=u1∪u2; e=e1++e2;
      as t = Some (l1,u1,e1); ash h1 ∥ ash h2 = Some (l2,u2,e2);
      x∈dom e1; dom l1∩dom l2 = {}; dom e1∩dom e2 = {}
    ⟧ ⟹ P
```

⟧ ⟹ *P*
proof (*induct h arbitrary*: *l u e P rule*: *ash.induct*)
case *1* **thus** *?case* **by** *fastsimp*
next
case (*2 t h*) **note** *C=this*
from *as-comp-SomeE*[*OF C*(*2*)[*simplified*]] **obtain** *l1 u1 e1 l2 u2 e2* **where**
SPLIT-simps[*simp*]: *l = l1 ++ l2* *u = u1 ∪ u2* *e = e1 ++ e2* **and**
SPLIT: *as t = Some* (*l1*, *u1*, *e1*) *ash h = Some* (*l2*, *u2*, *e2*)
dom l1 ∩ dom l2 = {} *dom e1 ∩ dom e2 =* {}
.
from *C*(*3*) **have** *x∈dom e1 ∨ x∈dom e2* **by** *auto*
moreover {
assume *A*: *x∈dom e1*
moreover have *t#h =* []@*t#h* **by** *simp*
ultimately have *?case* **by** (*rule-tac C*(*4*)) (*assumption*, (*simp add*: *SPLIT*)+)

} **moreover** {
assume *A*: *x∈dom e2*
from *C*(*1*)[*OF SPLIT*(*2*) *A*] **obtain** *h1 tt h2 l21 u21 e21 l22 u22 e22* **where**
IHAPP-simp[*simp*]: *h = h1 @ tt # h2* *l2=l21++l22*
u2=u21∪u22 *e2=e21++e22* **and**
IHAPP: *as tt = Some* (*l21*, *u21*, *e21*)
ash h1 ∥ *ash h2 = Some* (*l22*,*u22*,*e22*)
x∈dom e21
dom l21∩dom l22={}
dom e21∩dom e22={}
.
from *SPLIT IHAPP* **have**
DS: *dom l1 ∩ dom l21 =* {} *dom e1 ∩ dom e21 =* {}
by *auto*
have *t#h=*(*t#h1*)*@tt#h2* *l = l21 ++* (*l1++l22*)
u=u21∪(*u1∪u22*) *e=e21 ++* (*e1++e22*)
by (*auto simp add*: *map-add-comm*[*OF DS*(*1*)] *map-add-comm*[*OF DS*(*2*)])
moreover have *ash* (*t#h1*) ∥ *ash h2 = Some* (*l1++l22*,*u1∪u22*,*e1++e22*)
proof −
have *ash* (*t#h1*) ∥ *ash h2 = as t* ∥ (*ash h1* ∥ *ash h2*) **by** (*simp*)
also have ... = *as-comp* (*l1*,*u1*,*e1*) (*l22*,*u22*,*e22*)
by (*simp add*: *IHAPP*(*2*) *SPLIT*(*1*))
also have ... = *Some* (*l1++l22*,*u1∪u22*,*e1++e22*)
using *SPLIT IHAPP* **by** *auto*
finally show *?thesis* .
qed
ultimately have *?case* **using** *SPLIT*(*3*,*4*) *IHAPP*(*1*,*3*,*4*,*5*)
by (*rule-tac C*(*4*)) (*assumption+*, *auto*)
} **ultimately show** *?case* **by** *blast*
qed

Auxilliary lemma to split the acquisistion history of a hedge by some tree in that hedge.

lemma *ash-split-aux*:
  assumes *AS*: *ash h* = *Some* (*l*,*u*,*e*) **and**
    HFMT[*simp*]: *h*=*h1*@*t*#*h2* **and**
    C: !!*l1 u1 e1 l2 u2 e2*. ⟦
      l=*l1*++*l2*; *u*=*u1*∪*u2*; *e*=*e1*++*e2*; *as t* = *Some* (*l1*,*u1*,*e1*);
      ash h1 ∥ *ash h2* = *Some* (*l2*,*u2*,*e2*);
      dom l1 ∩ *dom l2* = {}; *dom e1* ∩ *dom e2* = {}
    ⟧ ⟹ *P*
  shows *P*
proof −
  have *as t* ∥ (*ash h1* ∥ *ash h2*) = *ash h* **by** *simp*
  also note *AS*
  finally have *1*: *as t* ∥ *ash h1* ∥ *ash h2* = *Some* (*l*, *u*, *e*) **.**
  show *P* **by** (*rule as-comp-SomeE*[*OF 1*], *rule C*) *assumption*+
qed

Auxilliary lemma that combines *ash-split-aux* and *wn-h-split-aux*.

lemma *wn-ash-split-aux*:
  assumes
    WN: *wn-h h μ* **and**
    AS: *ash h* = *Some* (*l*,*u*,*e*) **and**
    HFMT[*simp*]: *h*=*h1*@*t*#*h2* **and**
    C: !!*μ1 xs μ2 l1 u1 e1 l2 u2 e2*. ⟦
      μ=*μ1*@*xs*#*μ2*; *l*=*l1*++*l2*; *u*=*u1*∪*u2*; *e*=*e1*++*e2*;
      wn-t′ t xs; *wn-h h1 μ1*; *wn-h h2 μ2*;
      as t = *Some* (*l1*,*u1*,*e1*); *ash h1* ∥ *ash h2* = *Some* (*l2*,*u2*,*e2*);
      locks-μ μ1 ∩ *set xs* = {}; *locks-μ μ1* ∩ *locks-μ μ2* = {};
      set xs ∩ *locks-μ μ2* = {}; *dom l1* ∩ *dom l2* = {}; *dom e1* ∩ *dom e2* = {}
    ⟧ ⟹ *P*
  shows *P*
  apply (*rule wn-h-split-aux*[*OF WN HFMT*])
  apply (*rule ash-split-aux*[*OF AS HFMT*])
  apply (*rule C*)
  apply *assumption*+
  done

context *LDPN*
begin

Precision of the acqusisition structure construction, i.e. for a well-nested hedge, a consistent acquisistion history implies a schedule.

theorem *acqh-precise*:
  fixes *h*::(′*P*,′T,′*L*,′*X*) *lex-hedge*
  assumes *A*: *ash h*=*Some* (*l*,*u*,*e*)   *cons-as* (*l*,*u*,*e*) (*locks-μ μ*)   *wn-h h μ*
  shows ∃ *w*. *lsched h* (*locks-μ μ*) *w*
  — The proof is done by induction on the size of the hedge.
Given a non-empty hedge, it constructs the first step of the schedule and shows that the acquisistion structure remains consistent.

It considers the following cases:

- If the hedge contains a root that has no effect on locks, this root is scheduled. Those steps can always be scheduled, as the acquisition structure and the set of acquired locks do not change.
- If the hedge contains a root that initially releases a lock x, it is scheduled. A release can always be scheduled, as it cannot block. The new acquisition structure remains consistent: The acqusisition history is unchanged, the release history decreases (the lock x is removed). Consistency is preserved, as the lock x does not occur in the set of acquired locks any more.
- If the hedge contains only roots that are lock acquisitions or leafs, we further distinguish whether some of the roots are usages, or there are only final acquisitions.
  - If some of the roots are usages, we can find a usage where the used locks are disjoint from the domain of the release history (Due to acyclicity of the RH). Intuitively, this is a usage where the required locks are already released. This usage could be scheduled as a whole, without changing the RH, AH or set of acquired locks, and only decreasing the set of used locks. However, we chose another way here and show that scheduling only the first acquisition step of the usage also preserves consistency of the AS. We chose this approach in order to not having to formalize the scheduling of a usage. We assume that this simplifies formalization overhead (Perhaps at the cost of increased proof complexity).
  - If all of the roots are leafs or final acquisitions, due to acyclicity of the AH, we can select a final acquisition that acquires a lock that is not used in the rest of the hedge. Scheduling this acquisition preserves consistency of the AS.

proof –
 {
  fix *h*::(*'P*,'T,*'L*,*'X*) *lex-hedge* **and** *l u e μ s*
  assume *A*: *ash h*=*Some* (*l*,*u*,*e*) *cons-as* (*l*,*u*,*e*) (*locks-μ μ*) *wn-h h μ*
     hedge-size h = *s*
  from *A* **have** ∃ *w*. *lsched h* (*locks-μ μ*) *w*
  proof (*induct s arbitrary*: *h l u e μ rule*: *nat-compl-induct'*)
   case *0* — Empty hedge, the proposition is trivial
   thus *?case* **by** (*rule-tac x*=[] **in** *exI*) (*auto intro*: *lsched.intros*)
  next
   case (*Suc s*)
    — Non-empty hedge. Make the case-distinction depicted above
   show *?case*
   proof (*cases rule*: *h-cases*[*of h*])
    case *final* — The hedge only contains leafs. The proposition is also trivial then, as the empty path is a valid schedule.
    thus *?thesis* **by** (*rule-tac x*=[] **in** *exI*) (*auto intro*: *lsched.intros*)
   next

case (*spawn h1 lab ts t h2*)[*simp*] — The hedge contains a spawn step. By assumption, spawn steps have no effect on locks. hence, scheduling the spawn step does not affect the consistency criteria.

from *Suc.prems*(*3*)[*simplified*] **obtain** *nlab* **where**
[*simp*]: *lab=LNone nlab*
by (*auto elim*: *wn-h-spawn-imp-LNoneE*)
have *SIZE*: *hedge-size* (*h1@ts#t#h2*) $\leq$ *s* **using** *Suc.prems*(*4*) **by** *simp*
from *wn-h-preserve-spawn*[*of* μ *LNone nlab* *locks-*μ μ,
OF - Suc.prems(*3*)[*simplified*]]
obtain μ' **where**
[*simp*]: *locks-*μ μ'=*locks-*μ μ **and**
WNH: *wn-h* (*h1@ts#t#h2*) μ'
by *auto*
from *Suc.hyps*[*OF SIZE - - WNH*] *Suc.prems*(*1*,*2*) **obtain** *w* **where**
LS: *lsched* (*h1@ts#t#h2*) (*locks-*μ μ') *w*
by *fastsimp*
from *lsched-spawn*[*OF LS*, *of locks-*μ μ *LNone nlab*] **show** *?thesis*
by *auto*
next

case (*none h1 t nlab h2*)[*simp*] — The hedge contains a non-spawning step with no effects on locks. Scheduling this step does not affect the consistency criteria.

have *SIZE*: *hedge-size* (*h1@t#h2*) $\leq$ *s* **using** *Suc.prems*(*4*) **by** *simp*
from *wn-h-preserve-nospawn*[*of* μ *LNone nlab* *locks-*μ μ,
OF - Suc.prems(*3*)[*simplified*]]
obtain μ' **where**
[*simp*]: *locks-*μ μ'=*locks-*μ μ **and**
WNH: *wn-h* (*h1@t#h2*) μ'
by *auto*
from *Suc.hyps*[*OF SIZE - - WNH*] *Suc.prems*(*1*,*2*) **obtain** *w* **where**
LS: *lsched* (*h1@t#h2*) (*locks-*μ μ') *w*
by *fastsimp*
from *lsched-nospawn*[*OF LS*, *of locks-*μ μ *LNone nlab*] **show** *?thesis*
by *auto*
next

case (*release h1 x t h2*)[*simp*] — We have at least one release step. Scheduling a release step is always possible and will not make the release history inconsistent, as its effect is to remove an entry from the release history

have *SIZE*: *hedge-size* (*h1@t#h2*) $\leq$ *s* **using** *Suc.prems*(*4*) **by** *simp*
from *Suc.prems*(*3*)[*simplified*] **obtain** μ*1 xs* μ*2* **where**
[*simp*]: μ=μ*1@xs#*μ*2* **and**
WN-SPLIT: *wn-h h1* μ*1* *wn-t'* (*NNOSPAWN* (*LRel x*) *t*) *xs*
wn-h h2 μ*2* **and**
WN-DISJ: *locks-*μ μ*1* $\cap$ *set xs* = {} *locks-*μ μ*1* $\cap$ *locks-*μ μ*2* = {}
set xs $\cap$ *locks-*μ μ*2* = {}
by (*fastsimp elim*: *wn-h-prepend-h wn-h-append-h*)
from *WN-SPLIT*(*2*) **obtain** *xsh* **where**
[*simp*]: *xs=x#xsh* **and**

XS-SPLIT: $x \notin set\ xsh$ *wn-t′ t xsh*
by *auto*
from *WN-SPLIT WN-DISJ XS-SPLIT* **have**
WNH: *wn-h* (*h1*@*t*#*h2*) (*μ1*@*xsh*#*μ2*) **and**
WNH′: *wn-h* (*h1*@*h2*) (*μ1*@*μ2*)
by (*auto intro*!: *wn-h-appendI wn-h-prependI*)
have *ash* (*h1*@(*NNOSPAWN* (*LRel x*) *t*)#*h2*) =
as (*NNOSPAWN* (*LRel x*) *t*) ‖ *ash* (*h1*@*h2*)
by *auto*
with *Suc.prems*(*1*) **have**
as (*NNOSPAWN* (*LRel x*) *t*) ‖ *ash* (*h1*@*h2*) = *Some* (*l*,*u*,*e*)
by *simp*
then obtain *lt ut et l2 u2 e2* **where**
ASS-simps: *as* (*NNOSPAWN* (*LRel x*) *t*) = *Some* (*lt*,*ut*,*et*)
ash (*h1*@*h2*) = *Some* (*l2*,*u2*,*e2*)
l=*lt*++*l2*   *u*=*ut*∪*u2*   *e*=*et*++*e2* **and**
ASS: *dom lt* ∩ *dom l2* = {}   *dom et* ∩ *dom e2* = {}
by (*erule-tac as-comp-SomeE*) *blast*
from *ASS-simps*(*1*) **have** *XIDLT*: *x*∈*dom lt* **by** (*auto split*: *eahl-splits*)
from *wn-h-dom-l-lower-μ*[*OF WNH′*, *simplified*] *WN-DISJ*[*simplified*]
have *XNIDL2*: *x*∉*dom l2* **by** (*simp add*: *ASS-simps*[*simplified*]) *blast*
from *ASS-simps*(*1*) **have** *AS-T*: *as t* = *Some* (*l-remove lt x*, *ut*, *et*)
apply (*auto split*: *option.split-asm prod.split-asm*)
apply (*drule-tac wn-t-dom-l-lower-μ*[*OF XS-SPLIT*(*2*)])
apply (*force simp add*: *l-remove-def intro*!: *ext iff add*: *XS-SPLIT*(*1*))
done
have *ash* (*h1*@*t*#*h2*) = *as t* ‖ *ash* (*h1*@*h2*) **by** *simp*
also from *XNIDL2 ASS*
have *as t* ‖ *ash* (*h1*@*h2*) = *Some* (*l-remove l x*, *u*, *e*)
apply (*simp only*: *AS-T ASS-simps*(*2*))
apply (*simp add*: *ASS-simps*)
apply (*auto simp add*: *l-remove-def map-add-comm*)
apply (*force intro*!: *ext simp add*: *map-add-def split*: *option.split*)
done
finally have *G1*: *ash* (*h1*@*t*#*h2*) = *Some* (*l-remove l x*, *u*, *e*) **.**
from *Suc.prems*(*2*) **have**
G2: *cons-as* (*l-remove l x*, *u*, *e*) (*locks-μ* (*μ1*@*xsh*#*μ2*))
using *XIDLT WN-DISJ*[*simplified*] *XS-SPLIT*(*1*)
by *simp* (*blast 5 intro*!: *cons-h-remove*)
from *Suc.hyps*[*OF SIZE G1 G2 WNH*] **obtain** *w* **where**
IHAPP: *lsched* (*h1* @ *t* # *h2*) (*locks-μ* (*μ1* @ *xsh* # *μ2*)) *w*
by *blast*
moreover have *lock-valid* (*locks-μ μ*) (*LRel x*) (*locks-μ* (*μ1*@*xsh*#*μ2*))
using *WN-DISJ XS-SPLIT*(*1*) **by** *simp*
ultimately have *lsched* (*h*) (*locks-μ μ*) ((*LRel x*)#*w*)
by (*auto intro*: *lsched.intros*)
thus *?thesis* **by** *blast*
next
case *acquire* — All the trees start either with acquisitions or are leafs. This

case is the complex part of the proof.

We first distinguish whether there is a usage or all acquisitions are final acquisitions.

{

assume *C*: ∃ *Xu*. ∃ *t*∈*set h*. *closing′ t* = *Some Xu* — There is a usage

— Find a tree that starts with a usage, where the used locks are disjoint from the release history.

obtain *x Xu t* **where**

USE: *NNOSPAWN* (*LAcq x*) *t* ∈ *set h*   *closing x t* = *Some Xu*

insert x Xu ∩ *dom l* = {}

proof (*cases dom l* = {})

case *True*[*simp*] — Simple case: Domain of RH is empty, hence we can take any tree in h

from *C* **obtain** *Xu t* **where** *1*: *t*∈*set h*   *closing′ t* = *Some Xu*

by *blast*

then obtain *x t′* **where**

[*simp*]: *t*=*NNOSPAWN* (*LAcq x*) *t′* **and**

CL: *closing x t′* = *Some Xu*

by (*cases t rule*: *closing′.cases*) *auto*

with *1* **show** *?thesis* **by** (*rule-tac that*) *simp-all*

next

case *False* — Complex case: Domain of RH is not empty, we have to take tree that contains minimal element of RH

with *Suc.prems*(*2*) **obtain** *x* **where** *MIN*: *rh-min l x*

by (*force dest*: *cons-h-ex-rh-min*)

hence *MIDL*: *x*∈*dom l* **by** (*auto split*: *option.split-asm*)

from *ash-find-l-t*[*OF Suc.prems*(*1*) *MIDL*]

obtain *h1 t h2 l1 u1 e1 l2 u2 e2* **where**

FT-simps[*simp*]: *h* = *h1* @ *t* # *h2*   *l* = *l1* ++ *l2*

u = *u1* ∪ *u2*   *e* = *e1* ++ *e2* **and**

FT: *as t* = *Some* (*l1*, *u1*, *e1*)

ash h1 ∥ *ash h2* = *Some* (*l2*, *u2*, *e2*) **and**

MIDL1: *x* ∈ *dom l1* **and**

FT-DISJ: *dom l1* ∩ *dom l2* = {}   *dom e1* ∩ *dom e2* = {} **.**

obtain *x′ t′* **where** *TFMT*[*simp*]: *t*=*NNOSPAWN* (*LAcq x′*) *t′*

using *FT*(*1*) *MIDL1*

by (*subgoal-tac t*∈*set h*)

(*erule acquire*(*1*), *auto split*: *option.split-asm*)

have *G1*: *NNOSPAWN* (*LAcq x′*) *t′* ∈ *set h* **by** *simp*

from *Suc.prems*(*3*) **obtain** *μ1 xs μ2* **where**

[*simp*]: *μ*=*μ1*@*xs*#*μ2* **and**

WN-SPLIT: *wn-h h1 μ1*   *wn-t′ t xs*   *wn-h h2 μ2* **and**

WN-DISJ: *locks-μ μ1* ∩ *set xs* = {}   *locks-μ μ1* ∩ *locks-μ μ2* = {}

set xs ∩ *locks-μ μ2* = {}

by (*fastsimp elim*: *wn-h-append-h wn-h-prepend-h*)

from *WN-SPLIT*(*2*) **have** *WN′*: *wn-t′ t′* (*x′*#*xs*) **by** *simp*

from *FT*(*1*) **obtain** *l1′ u1′ e1′* **where**

AS: *as t′* = *Some*(*l1′*,*u1′*,*e1′*) **and**

UU: *dom l1* ⊆ *dom l1'*   *x'*∉*dom l1*
by (*force split*: *eahl-splits*)
from *UU MIDL1* **have** *MIDL'*: *x*∈*dom l1'* **by** *auto*
from *MIDL1 UU* **have** *MNE*: *x*≠*x'* **by** *auto*
from *wn-t-dom-l-stack'*[*OF WN' AS MIDL'*]
obtain *xs1 xs2* **where**
x'#*xs* = *xs1*@*x*#*xs2*   *set xs1* ⊆ *dom l1'*
∀ *x'*∈*set xs1*. *l1' x'* ≤ *l1' x* ∧ *x* ∉ *the* (*l1' x'*) ∧ *x'* ∉ *the* (*l1' x'*)
by *blast*
then obtain *Xu* **where** *L1'X'*: *l1' x'* = *Some Xu*   *Some Xu* ≤ *l1' x*
using *MNE* **by** (*cases xs1*) *auto*
from *dom-l-closing*[*OF AS WN'*, *OF L1'X'*(*1*)] **have**
G2: *closing x' t'* = *Some Xu* **.**
from *L1'X'*(*1*) *FT*(*1*) *AS* **have**
L1FMT[*simp*]: *l1* = *l-add-use* (*l-remove l1' x'*) {*x'*} **and**
X'IU: *x'*∈*u*
by (*auto split*: *eahl-splits*)
from *MNE MIDL'* **have**
l1' x ≤ *l1 x* **and**
X'IL1X: *x'* ∈ *the* (*l1 x*)
by (*auto simp add*: *l-add-use-def split*: *option.split*)
with *L1'X'* **have** *Some Xu* ≤ *l1 x* **by** *auto*
with *FT-DISJ MIDL1* **have**
XULE: *Some Xu* ≤ *l x*
by (*auto simp del*: *L1FMT simp add*: *map-add-def split*: *option.split*)
with *MIN* **have** *the* (*l x*) ∩ *dom l* = {} **by** *auto*
moreover from *XULE MIDL* **have** *Xu* ⊆ *the* (*l x*)
by (*auto simp add*: *le-option-def split*: *option.split-asm*)
moreover from *X'IL1X FT-DISJ MIDL1* **have** *x'*∈*the* (*l x*)
by (*auto simp add*: *map-add-def split*: *option.split*)
ultimately have *G3*: *insert x' Xu* ∩ *dom l* = {} **by** *auto*
from *that*[*OF G1 G2 G3*] **show** *?thesis* **.**
qed

— Split h (This duplicates some work done in the complex case of the proof above)
from *in-set-list-format*[*OF USE*(*1*)] **obtain** *h1 h2* **where**
HFMT[*simp*]: *h*=*h1*@(*NNOSPAWN* (*LAcq x*) *t*)#*h2* **.**
from *Suc.prems*(*3*) **obtain** *μ1 xs μ2* **where**
[*simp*]: *μ*=*μ1*@*xs*#*μ2* **and**
WN-SPLIT: *wn-h h1 μ1*   *wn-t'* (*NNOSPAWN* (*LAcq x*) *t*) *xs*
wn-h h2 μ2 **and**
WN-DISJ: *locks-μ μ1* ∩ *set xs* = {}   *locks-μ μ1* ∩ *locks-μ μ2* = {}
set xs ∩ *locks-μ μ2* = {}
by (*fastsimp elim*: *wn-h-append-h wn-h-prepend-h*)
from *WN-SPLIT*(*2*) **have** *WN'*: *wn-t' t* (*x*#*xs*) **by** *simp*

— Split acquisition structure according to splitting of h
from *Suc.prems*(*1*) **obtain** *l1 u1 e1 l2 u2 e2* **where**

```
  AS-SPLIT: as (NNOSPAWN (LAcq x) t) = Some (l1,u1,e1)
            ash h1 ∥ ash h2 = Some (l2,u2,e2) and
  [simp]: l=l1++l2    u=u1∪u2    e=e1++e2 and
    AS-DISJ: dom l1 ∩ dom l2 = {}    dom e1 ∩ dom e2 = {}
proof -
  have as (NNOSPAWN (LAcq x) t) ∥ (ash h1 ∥ ash h2) = ash h
    by auto
  also have ... = Some (l,u,e) using Suc.prems(1) .
 finally show ?thesis by (erule-tac as-comp-SomeE) (blast intro!: that)
qed

— Obtain facts about new tree's acquisition structure
from wn-closing-as-fmt[OF WN-SPLIT(2) AS-SPLIT(1) USE(2)]
obtain l1' u1' where
  S: as t = Some (l1', u1', e1)    l1' ≤ l1(x ↦ Xu)
    u1 = insert x u1'    dom l1' = insert x (dom l1) .

from USE(3) have XNIDL: x∉dom l by simp
from S(3) XNIDL Suc.prems(2) have XNILM: x∉locks-μ μ by auto

— Construct new hedge's acquisition structure
have ash (h1@t#h2) = as t ∥ (ash h1 ∥ ash h2) by simp
also have ... = as-comp (l1',u1',e1) (l2,u2,e2)
  by (simp add: S(1) AS-SPLIT(2))
also have ... = Some (l1'++l2,u1'∪u2, e)
  using XNIDL S(4) AS-DISJ by auto
finally have
  ASH': ash (h1 @ t # h2) = Some (l1' ++ l2, u1' ∪ u2, e) .

— Collect facts for induction hypothesis
from XNILM WN-DISJ WN-SPLIT WN' have
  WNH': wn-h (h1@t#h2) (μ1@(x#xs)#μ2)
  by (auto intro!: wn-h-appendI wn-h-prependI)

have CONS': cons-as (l1' ++ l2, u1' ∪ u2, e) (locks-μ (μ1@(x#xs)#μ2))

proof -
  have CONSL': cons-h (l1'++l2) proof -
    from S(2) have LLE: l1'++l2 ≤ l(x ↦ Xu)
      using XNIDL by (rule-tac le-funI, drule-tac x=xa in le-funD)
                (auto simp add: map-add-def split: option.split)
    from Suc.prems(2) have CL: cons-h l by simp
    from wn-closing-ni[where ?μ1.0=[], simplified, OF WN' USE(2)]
    have x∉Xu .
    with cons-h-update[OF CL, of Xu x] USE(3)
    have cons-h (l(x↦Xu)) by auto
    with cons-h-antimono[OF LLE] show ?thesis by simp
  qed
```

from *Suc.prems*(*2*) **have** *1*: (*locks-μ μ* − *dom l*) ∩ (*u*∪*dom e*) = {}
  by *auto*
from *S*(*4*) **have**
  2: (*locks-μ μ* − *dom l*) ⊇
    (*locks-μ* (*μ1*@(*x*#*xs*)#*μ2*) − *dom* (*l1'* ++ *l2*))
  by *auto*
from *S*(*3*) **have** *3*: (*u*∪*dom e*) ⊇ *u1'* ∪ *u2* ∪ *dom e* **by** *auto*
from *disjoint-mono*[*OF 2 3 1*] **have**
  (*locks-μ* (*μ1*@(*x*#*xs*)#*μ2*) − *dom* (*l1'* ++ *l2*)) ∩
    (*u1'* ∪ *u2* ∪ *dom e*) = {} **.**
moreover from *Suc.prems*(*2*) **have** *cons-h e* **by** *auto*
moreover note *CONSL'*
ultimately show *?thesis* **by** (*auto*)
qed

have *SIZE*: *hedge-size* (*h1*@*t*#*h2*) ≤ *s* **using** *Suc.prems*(*4*) **by** *simp*

— Apply induction hypothesis
from *Suc.hyps*[*OF SIZE ASH' CONS' WNH'*] **obtain** *w* **where**
  IHAPP: *lsched* (*h1* @ *t* # *h2*) (*locks-μ* (*μ1* @ (*x* # *xs*) # *μ2*)) *w*
  by *blast*

— Show that we can schedule the first step
have *LV*:
  lock-valid (*locks-μ μ*) (*LAcq x*) (*locks-μ* (*μ1*@(*x*#*xs*)#*μ2*))
  using *XNILM* **by** *simp*
from *lsched-nospawn*[*OF IHAPP LV*] **have** *?thesis* **by** *auto*
} moreover {
assume *C*: ∀ *t*∈*set h*. *closing' t* = *None*
  — All the acquisitions at the roots of the hedge are final.

— The release history is empty, and any used lock occurs after a final acquisition
have *l*=*empty* ∧ *u*=⋃*ran e* ∧
    ⋃*ran* (*e* |' *rootlocks h*) = ⋃*ran e* ∧ *rootlocks h* ⊆ *dom e*
  by (*blast intro*!: *ncl-as-fmt*[*OF Suc.prems*(*3*,*1*) - *C*] *intro*: *acquire*(*1*))
hence
  [*simp*]: *l*=*empty* **and**
  NCL: *u*=⋃*ran e* **and**
  XMS: ⋃*ran* (*e* |' *rootlocks h*) = ⋃*ran e*   *rootlocks h* ⊆ *dom e*
  by *auto*

— There is at least one tree starting with an acquisition, thus the acquisition history is not empty
have *RLNE*: *rootlocks h* ≠ {} **and** *ENE*: *e*≠*empty* **proof** −
  obtain *t' x h1 h2* **where**
    HFMT[*simp*]: *h*=*h1*@(*NNOSPAWN* (*LAcq x*) *t'*)#*h2*
    by (*blast intro*: *acquire*(*2*) *elim*: *in-set-list-format*)
  thus *rootlocks h* ≠ {} **by** *auto*

with *XMS*(*2*) **show** *e≠empty* **by** *auto*
qed

— We can obtain a minimal lock that is acquired at a root of some tree
obtain *x* **where** *XIR*: *x∈rootlocks h* **and** *MIN*: *ah-min e x* **proof** −
have *1*: *e* |' *rootlocks h* ≠ *empty* **using** *XMS*(*2*) *RLNE*
by (*subgoal-tac dom* (*e* |' *rootlocks h*) ≠ {}) *fastsimp+*
from *cons-h-ex-ah-min*[*OF 1 cons-h-antimono*[*of e*|'*rootlocks h e*]]
Suc.prems(*2*)
obtain *x* **where** *ah-min* (*e* |' *rootlocks h*) *x*
by *auto*
with *XMS*(*1*) **show** *?thesis* **by** (*auto intro*!: *that*)
qed

— Find the tree with *x* at the root
from *rootlocks-split*[*OF XIR*] **obtain** *h1 t h2* **where**
HFMT[*simp*]: *h=h1@NNOSPAWN* (*LAcq x*) *t#h2* **.**
— Split lock-stacks and acquisistion structures
from *wn-ash-split-aux*[*OF Suc.prems*(*3*,*1*) *HFMT*]
obtain *μ1 xs μ2 l1 u1 e1 l2 u2 e2* **where**
SPLIT-simps[*simp*]: *μ* = *μ1* @ *xs* # *μ2*   *u* = *u1* ∪ *u2*
e = *e1* ++ *e2* **and**
WNS: *wn-t'* (*NNOSPAWN* (*LAcq x*) *t*) *xs*   *wn-h h1 μ1*
wn-h h2 μ2 **and**
ASS: *as* (*NNOSPAWN* (*LAcq x*) *t*) = *Some* (*l1*, *u1*, *e1*)
ash h1 ∥ *ash h2* = *Some* (*l2*, *u2*, *e2*) **and**
DISJ: *locks-μ μ1* ∩ *set xs* = {}   *locks-μ μ1* ∩ *locks-μ μ2* = {}
set xs ∩ *locks-μ μ2* = {}   *dom l1* ∩ *dom l2* = {}
dom e1 ∩ *dom e2* = {} **and**
LL: *l* = *l1* ++ *l2*
.
from *LL* **have** [*simp*]: *l1=empty*   *l2=empty* **by** *auto*

— Get acquisition structure of *t*
obtain *e1'* **where**
AS': *as t* = *Some* (*empty*,*u1*,*e1'*)   *e1* = *e1'*(*x* ↦ *u1*)   *x* ∉ *dom e1'*
by (*rule-tac*
ncl-as-fmt-single'[*OF WNS*(*1*)
C[*rule-format*, *of NNOSPAWN* (*LAcq x*) *t*]
ASS(*1*)]
)
(*simp*)

— Get acqusition structure of new hedge
have *ASH'*: *ash* (*h1@t#h2*) = *Some* (*empty*,*u*,*e1'++e2*) **proof** −
from *AS'*(*2*,*3*) *DISJ* **have** *D'*: *dom e1'* ∩ *dom e2* = {} **by** *simp*
have *ash* (*h1@t#h2*) = *as t* ∥ (*ash h1* ∥ *ash h2*) **by** *simp*
also from *DISJ D' AS' ASS*(*2*) **have** ... = *Some* (*empty*,*u*,*e1'++e2*)

```
          by simp
        finally show ?thesis .
      qed

      — The new hedge is well-nested
      from AS'(2) Suc.prems(2) have XNILM: x∉locks-μ μ by auto
      have WN': wn-h (h1@t#h2) (μ1@(x#xs)#μ2)
      using WNS DISJ XNILM by (auto intro!: wn-h-appendI wn-h-prependI)

      — The new acquisition history is consistent
      have CONS': cons-as (empty,u,e1'++e2) (locks-μ (μ1@(x#xs)#μ2))
      proof −
        have cons-h (e1'++e2) proof −
          from AS'(2,3) have e1' ≤ e1 by (simp add: le-fun-def dom-def)
          hence 1: e1'++e2 ≤ e by (auto intro!: map-add-first-le)
          from cons-h-antimono[OF 1] Suc.prems(2) show ?thesis by auto
        qed
        moreover
        have insert x (locks-μ μ) ∩ (dom (e1'++e2) ∪ u) = {} proof −
          from AS' have DEF: dom e = insert x (dom (e1' ++ e2)) by auto
          from Suc.prems(2) have DJO: locks-μ μ ∩ (dom e ∪ u) = {}
            by auto
          have 1: (dom (e1' ++ e2) ∪ u) ⊆ dom e ∪ u using DEF by auto
          from disjoint-mono[of locks-μ μ    locks-μ μ, OF - 1 DJO] have
            locks-μ μ ∩ (dom (e1' ++ e2) ∪ u) = {}
            by simp
          moreover from AS' DISJ have x∉dom (e1'++e2) by auto
          moreover from MIN NCL have x∉u by simp
          ultimately show ?thesis by simp
        qed
        ultimately show ?thesis by fastsimp
      qed

      — Now we can apply the induction hypothesis and finnish the proof
      have SIZE: hedge-size (h1@t#h2) ≤ s using Suc.prems(4) by simp

      from Suc.hyps[OF SIZE ASH' CONS' WN'] obtain w where
        IHAPP: lsched (h1 @ t # h2) (locks-μ (μ1 @ (x#xs) # μ2)) w
        by blast
    moreover have lock-valid (locks-μ μ) (LAcq x) (locks-μ (μ1@(x#xs)#μ2))
        using XNILM by simp
      ultimately have lsched (h) (locks-μ μ) ((LAcq x)#w)
        by (auto intro: lsched.intros)
      hence ?thesis by blast
    } ultimately show ?thesis by force
  qed
qed
}
with A show ?thesis by blast
```

qed

The following is the main theorem of this section. It states the correctness of the acquisition structure construction. For all non-empty hedges that are well-nested w.r.t. a list of lock-stacks with locks X, the existence of a schedule starting with locks X is equivalent to the conistency of the hedge's acquisition history w.r.t. X.

lemma *acqh-correct'*:
  fixes *h*::('*P*,'T,'*L*,'*X*) *lex-hedge*
  shows ⟦*wn-h h μ*⟧ ⟹
  (∃ *w*. *lsched h* (*locks-μ μ*) *w*) ⟷
    (∃ *l u e*. *ash h* = *Some* (*l*, *u*, *e*) ∧ *cons-as* (*l*, *u*, *e*) (*locks-μ μ*)
)
  using *acqh-sound acqh-precise* **by** *blast*

theorem *acqh-correct*:
  fixes *h*::('*P*,'T,'*L*,'*X*) *lex-hedge*
  assumes *WN*: *wn-h h μ*
  shows (∃ *w*. *lsched h* (*locks-μ μ*) *w*) ⟷ *cons* (*ash h*) (*locks-μ μ*)
  using *WN*

  apply (*simp only*: *acqh-correct'*)
  apply (*cases ash h*)
  apply *simp*
  apply (*case-tac a*)
  apply (*case-tac b*)
  apply *simp*
  done

end

end

12 DPNs with Initial Configuration

theory *DPN-c0*
imports *WellNested*
begin

12.1 DPNs with Initial Configuration

In the following locale, we fix a DPN with an initial configuration, and a list of lock-stacks. We assume that the initial configuration is well-nested w.r.t. the list of lock-stacks.

This is the model we are able to analyze with our acquisition history based techniques, that assume well-nestedness.

Note that we – up to now – do not show that there exists a non-trivial instance of this locale. Such a proof would support the trust in that the model we formalize here is really the intended model.

locale *LDPN-c0* = *LDPN* +
  constrains Δ :: (*'P*,'T,*'L*,*'X*::*finite*) *ldpn*
  fixes *c0* :: (*'P*,'T) *conf* — Initial configuration
  fixes *μ0* :: *'X list list* — Locks held at the start configuration
  assumes *wellnested*: *wn-c* Δ *c0* *μ0* — Start configuration must be well-nested
begin

12.1.1 Reachable Configurations

definition *reachable* == { *c* . ∃ *w*. (*c0*,*w*,*c*)∈*dpntrc* Δ }
definition *reachablels* == { (*c*,*X*) . ∃ *w*. ((*c0*,*locks-μ μ0*),*w*,(*c*,*X*))∈*ldpntrc* Δ }

lemma *reachablels-subset*: (*c*,*X*)∈*reachablels* ⟹ *c*∈*reachable*
  by (*auto simp add*: *reachablels-def reachable-def intro*: *ldpntrc-subset*)

lemma *reachable-wn*:
  ⟦(*c*,*X*)∈*reachablels*; !!*μ*. ⟦*wn-c* Δ *c* *μ*; *X*=*locks-μ* *μ*⟧ ⟹ *P*⟧ ⟹ *P*
  apply (*unfold reachablels-def*)
  apply *simp*
  apply (*erule exE*)
  apply (*erule wnc-preserve*)
  apply (*rule wellnested*)
  apply *blast*
  done

lemma *reachablels-triv*[*simp*]: (*c0*, *locks-μ μ0*)∈*reachablels*
  by (*unfold reachablels-def*) (*auto intro*: *exI*[*of* - []])

end

end

13 Property Specifications

theory *Specification*
imports *DPN-c0 Semantics LockSem common/SublistOrder*
begin

We develop a formalism that allows a concise and readable notation for a class of properties that are checkable via cascaded predecessor computations.

A specification consists of a list of atoms, where each atom either restricts the current configuration or describes some step.

13.1 Specification Formulas

The base element of a property is an atom, that describes a step or restricts the current configuration

datatype ($'Q$,$'\Gamma$,$'L$,$'X$) *spec-atom* =
 — Restrict current configuration to be in a specified set
 SPEC-RESTRICT ($'Q$,$'\Gamma$) *conf set* |
 — Go forward one step, using a rule with labels from a specified set
 SPEC-STEP ($'L$,$'X$) *lockstep set* |
 — Go forward any number of steps, using rules with labels from a specified set
 SPEC-STEPS ($'L$,$'X$) *lockstep set*

A property is a list of atoms

types ($'Q$,$'\Gamma$,$'L$,$'X$) *spec* = ($'Q$,$'\Gamma$,$'L$,$'X$) *spec-atom list*

13.2 Semantics

The semantics of a property specification Φ w.r.t. the current DPN is modelled by a transition relation *spec-tr* Φ, that contains all pairs (c,c') of configurations, such that there is a path between c and c' satisfying the property.

context *LDPN*
begin
 fun *spec-tr* **where**
 spec-tr [] = *Id* |
 spec-tr (*SPEC-RESTRICT* C # Φ) = $\{(c,c')\ .\ (c,c')\in$ *spec-tr* $\Phi \wedge$ *fst* $c\in C\}$ |
 spec-tr (*SPEC-STEP* L # Φ) =
 $\{(c,c')\ .\ \exists\, l\in L.\ \exists\, ch.\ (c,l,ch)\in$ *ldpntr* $\Delta \wedge (ch,c')\in$ *spec-tr* $\Phi\}$ |
 spec-tr (*SPEC-STEPS* L # Φ) =
 $\{(c,c')\ .\ \exists\, ll\in$ *lists* $L.\ \exists\, ch.\ (c,ll,ch)\in$ *ldpntrc* $\Delta \wedge (ch,c')\in$ *spec-tr* $\Phi\}$
end

context *LDPN-c0*
begin

In most cases, it suffices to check whether there is a path matching the specification from the initial configuration.

definition *model-check-ref* Φ == $(c0,$*locks-*$\mu\ \mu0)\in$ *Domain* (*spec-tr* Φ)
end

13.3 Examples

In this section, we present two short examples to justify the usefulness of our property specifications.

13.3.1 Conflict analysis

Given two stack symbols u,v∈Γ, conflict analysis asks whether a configuration c is reachable that has a conflict between u and v.

A configuration has a conflict between u and v, iff it contains a process with top stack symbol u and another (different) process with top stack symbol v.

context *LDPN-c0*
begin

atUV u v is the set of configurations that have a conflict between u and v.

definition *atUV-ordered u v* == { *c*. ∃ *q r q′ r′*. [(*q*,*u*#*r*),(*q′*,*v*#*r′*)] ≤ *c* }
definition *atUV u v* == (*atUV-ordered u v*) ∪ (*atUV-ordered v u*)

The following property specification describes all executions reaching a conflict:

definition *conflict-spec u v* ==
[*SPEC-STEPS UNIV*, *SPEC-RESTRICT* (*atUV u v*)]

The following definition is a direct definition of a conflict between u and v being reachable from an initial configuration [(*qmain*,[*γmain*])]:

definition *has-conflict-ref u v* == ∃ (*c*,*X*)∈*reachablels*. *c* ∈ *atUV u v*

The next lemma shows that the direct definition of a conflict matches the property specification:

lemma *has-conflict-ref u v* ⟷ *model-check-ref* (*conflict-spec u v*)
by (*unfold model-check-ref-def conflict-spec-def has-conflict-ref-def Domain-def reachablels-def*)
auto

end

13.3.2 Bitvector analysis

Given a set of generator labels *G*::′*L set*, a set of killer labels *K*::′*L set* and a stack symbol *u*::′Γ, bitvector analysis asks whether there is a path to a configuration that has process being at u, such that the path executes a generator rule, and after that no killer rule is executed.

context *LDPN-c0*
begin

For a stack symbol, u∈Γ, the set *atU u* is the set of all configurations that have a process with u at the top of the stack.

definition *atU u* == { *c* . ∃ *q r*. (*q*,*u*#*r*)∈*set c* }

The following property specification describes all paths that lead to u and have the bit set:

definition *bitvector-fwd-spec G K u* ==
  [*SPEC-STEPS UNIV*,
    SPEC-STEP G,
    SPEC-STEPS (*UNIV*−*K*),
    SPEC-RESTRICT (*atU u*)
  ]

The following is the direct definition of bitvector analysis:

definition *bitvector-fwd-ref G K u* ==
  ∃ *c1 X1 lg c2 X2 ll c3 X3 q r*.
    (*c1*,*X1*)∈*reachablels* ∧
    ((*c1*,*X1*),*lg*,(*c2*,*X2*))∈*ldpntr* Δ ∧
    lg∈*G* ∧
    ((*c2*,*X2*),*ll*,(*c3*,*X3*))∈*ldpntrc* Δ ∧
    ll∈*lists* (*UNIV*−*K*) ∧
    (*q*,*u*#*r*)∈*set c3*

This lemma shows that the direct definition matches the property specification:

lemma *bitvector-fwd-ref G K u* ⟷
      model-check-ref (*bitvector-fwd-spec G K u*)
  by (*unfold model-check-ref-def bitvector-fwd-spec-def*
        bitvector-fwd-ref-def Domain-def atU-def reachablels-def)
    fastsimp

end
end

14 Hedge Constraints for Acquisition Histories

theory *As-hc*
imports *Acqh WellNested DPN-c0 Specification*
begin

This theory formulates the set of execution hedges that have a lock-sensitive schedule, and shows how to use hedge-constrained predecessor set computations to compute property specifications based on cascaded predecessor sets.

14.1 Locks Encoded in Control State

For this section, we make the assumption that the set of locks is encoded in the control state of the DPN. We formalize this by means of a locale.

locale *EncodedLDPN* = *LDPN* +
  — The states of the DPN are tuples of some states *'P* and sets of locks:
  constrains Δ :: (*'P*×*'X set*,'Γ,*'L*,*'X*::*finite*) *ldpn*

constrains *c0* :: ('*P*×'*X set*,'T) *conf*
constrains μ*0* :: '*X list list*
— A step of the DPN transforms the locks as expected:
assumes *encoding-correct-nospawn*:
 $((p,X),\gamma \hookrightarrow_l (p',X'),w) \in \Delta \Longrightarrow$ *lock-valid X l X'*
assumes *encoding-correct-spawn1*:
 $((p,X),\gamma \hookrightarrow_l (ps,Xs),ws \sharp (p',X'),w) \in \Delta \Longrightarrow$ *lock-valid X l X'*

— A freshly spawned process initially owns no locks:
assumes *encoding-correct-spawn2*:
 $((p,X),\gamma \hookrightarrow_l (ps,Xs),ws \sharp (p',X'),w) \in \Delta \Longrightarrow Xs=\{\}$
begin

lemmas *encoding-correct-spawn* = *encoding-correct-spawn1 encoding-correct-spawn2*
lemmas *encoding-correct* = *encoding-correct-nospawn encoding-correct-spawn*

lemma *encoding-correct-nospawn'*:
 $(p,\gamma \hookrightarrow_l p',w) \in \Delta \Longrightarrow$ *lock-valid* (*snd p*) *l* (*snd p'*)
 by (*cases p*, *cases p'*) (*auto intro*: *encoding-correct-nospawn*)

lemma *encoding-correct-spawn'*:
 assumes *A*: $(p,\gamma \hookrightarrow_l ps,ws \sharp p',w) \in \Delta$
 shows *lock-valid* (*snd p*) *l* (*snd p'*) *snd ps*={}
 using *A encoding-correct-spawn* **by** (*cases p*, *cases p'*, *cases ps*, *force*)+

lemma *encoding-correct-spawn2'*:
 $(p,\gamma \hookrightarrow_l ps,ws \sharp p',w) \in \Delta \Longrightarrow$ *snd ps* = {}
 using *encoding-correct-spawn* **by** (*cases p*, *cases p'*, *cases ps*, *force*)+

lemma *ec-preserve-singlestep*:
 assumes
  A: ((*c*,*locks*-μ μ),*l*,(*c'*,*X'*))∈*ldpntr* Δ *wn-c* Δ *c* μ
   map (*snd*∘*fst*) *c* = *map set* μ **and**
  C: !!μ'. ⟦ *wn-c* Δ *c'* μ'; *X'*=*locks*-μ μ';
     map (*snd*∘*fst*) *c'* = *map set* μ'
    ⟧ ⟹ *P*
 shows *P*
proof −
 from *A* **have**
  TR: (*c*,*l*,*c'*)∈*dpntr* Δ **and**
  LV: *lock-valid* (*locks*-μ μ) *l X'*
  by (*auto simp add*: *ldpntr-def*)
 from *TR* **show** *?thesis* **proof** (*cases rule*: *dpntr.cases*)
  case (*dpntr-no-spawn p* γ - *p' w c1 r c2*)
  hence
   FMT[*simp*]: *c* = *c1* @ (*p*, γ # *r*) # *c2* *c'* = *c1* @ (*p'*, *w* @ *r*) # *c2* **and**
   R: $(p,\gamma \hookrightarrow_l p',w) \in \Delta$
   by *auto*

from *wn-c-split-aux*[*OF A*(*2*) *FMT*(*1*)] **obtain** *μ1 xs μ2* **where**
[*simp*]: *μ = μ1 @ xs # μ2* **and**
WNS: *wn-π Δ (p, γ # r) xs* *wn-c Δ c1 μ1* *wn-c Δ c2 μ2* **and**
DISJ: *locks-μ μ1 ∩ set xs = {}* *locks-μ μ1 ∩ locks-μ μ2 = {}*
set xs ∩ locks-μ μ2 = {}
.
from *A*(*3*) *wn-c-length*[*OF WNS*(*2*)] *wn-c-length*[*OF WNS*(*3*)] **have**
ECS: *map (snd∘fst) c1 = map set μ1* *snd p = set xs*
map (snd∘fst) c2 = map set μ2
by *auto*
obtain *xs′* **where**
wn-π Δ (p′,w@r) xs′ *X′=(locks-μ (μ1@xs′#μ2))*
locks-μ μ1 ∩ set xs′ = {} *set xs′ ∩ locks-μ μ2 = {}* *snd p′ = set xs′*
proof (*cases l*)
case *LNone*[*simp*]
from *DISJ LV encoding-correct-nospawn′*[*OF R*] *ECS*(*2*) **show** *?thesis*
by (*rule-tac that*[*OF wn-π-none*[*OF R*[*simplified*] *WNS*(*1*)]]) *simp-all*
next
case (*LAcq x*)[*simp*]
from *that*[*OF wn-π-acq*[*OF R*[*simplified*] *WNS*(*1*)]] *LV DISJ*
encoding-correct-nospawn′[*OF R*] *ECS*(*2*)
show *?thesis* **by** *auto*
next
case (*LRel x*)[*simp*]
from *wn-π-rel*[*OF R*[*simplified*] *WNS*(*1*)] **obtain** *xs′* **where**
[*simp*]: *xs=x#xs′* **and**
1: *x∉set xs′* **and**
2: *wn-π Δ (p′,w@r) xs′*
.
from *1 LV DISJ encoding-correct-nospawn′*[*OF R*] *ECS*(*2*) **show** *?thesis*
by (*rule-tac that*[*OF 2*]) *auto*
qed
with *WNS*(*2*,*3*) *DISJ*(*2*) *ECS*(*1*,*3*) **show** *P*
by (*rule-tac μ′=μ1@xs′#μ2* **in** *C*) (*auto intro*!: *wn-c-appendI wn-c-prependI*)
next
case (*dpntr-spawn p γ - ps ws p′ w c1 r c2*) **hence**
FMT[*simp*]: *c = c1 @ (p, γ # r) # c2*
c′ = c1 @ (ps, ws) # (p′, w @ r) # c2 **and**
R: $(p,\gamma \hookrightarrow_l ps,ws \sharp p',w) \in \Delta$
by *auto*
from *R* **obtain** *nlab* **where** [*simp*]: *l=LNone nlab* **by** (*cases l*) *auto*
from *wn-c-split-aux*[*OF A*(*2*) *FMT*(*1*)] **obtain** *μ1 xs μ2* **where**
[*simp*]: *μ = μ1 @ xs # μ2* **and**
WNS: *wn-π Δ (p, γ # r) xs* *wn-c Δ c1 μ1* *wn-c Δ c2 μ2* **and**
DISJ: *locks-μ μ1 ∩ set xs = {}* *locks-μ μ1 ∩ locks-μ μ2 = {}*
set xs ∩ locks-μ μ2 = {}
.
from *A*(*3*) *wn-c-length*[*OF WNS*(*2*)] *wn-c-length*[*OF WNS*(*3*)] **have**
ECS: *map (snd∘fst) c1 = map set μ1* *snd p = set xs*

map (*snd*∘*fst*) *c2* = *map set* μ*2*
    by *auto*
  from *wn*-π-*spawn1*[*OF R WNS*(*1*)] *wn*-π-*spawn2*[*OF R WNS*(*1*)]
      WNS(*2*,*3*) *DISJ*
  have *wn-c* Δ *c*′ (μ*1*@[]#*xs*#μ*2*)
    by (*auto intro*!: *wn-c-appendI wn-c-prependI*)
  thus *?thesis*
    using *LV encoding-correct-spawn*′[*OF R*] *ECS*
    by (*rule-tac* μ′=μ*1*@[]#*xs*#μ*2* **in** *C*) *auto*
  qed
qed

lemma *ec-preserve*:
  assumes
    A: ((*c*,*locks*-μ μ),*ll*,(*c*′,*X*′))∈*ldpntrc* Δ *wn-c* Δ *c* μ
        map (*snd*∘*fst*) *c* = *map set* μ **and**
    C: !!μ′. ⟦*X*′=*locks*-μ μ′; *wn-c* Δ *c*′ μ′; *map* (*snd*∘*fst*) *c*′ = *map set* μ′⟧ ⟹ *P*
  shows *P*
proof −
  {
    fix *c X* μ *ll c*′ *X*′ *P*
    assume
      A: ((*c*,*X*),*ll*,(*c*′,*X*′))∈*ldpntrc* Δ *wn-c* Δ *c* μ
          map (*snd*∘*fst*) *c* = *map set* μ *X*=*locks*-μ μ **and**
      C: !!μ′. ⟦ *X*′=*locks*-μ μ′; *wn-c* Δ *c*′ μ′;
            map (*snd*∘*fst*) *c*′ = *map set* μ′
          ⟧ ⟹ *P*
    hence *P*
    proof (*induct arbitrary*: μ *P rule*: *trcl-pair-induct*)
      case *empty* **thus** *?case* **by** *auto*
    next
      case (*cons c x l ch Xh ll c*′ *X*′ μ *P*) **note** [*simp*]=‹*x*=*locks*-μ μ›
      from *ec-preserve-singlestep*[*OF cons.hyps*(*1*)[*simplified*] *cons.prems*(*1*,*2*)]
      obtain μ′ **where**
        P: *wn-c* Δ *ch* μ′   *map* (*snd* ∘ *fst*) *ch* = *map set* μ′   *Xh*=*locks*-μ μ′
        .
      from *cons.hyps*(*3*)[*OF P*] *cons.prems*(*4*) **show** *?case* **by** *blast*
    qed
  } **with** *A C* **show** *?thesis* **by** *blast*
qed

The following abbreviates the locks owned by a configuration:

abbreviation *locks-c c* == *list-collect-set* (*snd*∘*fst*) *c*

lemma *locks*-μ-*mapset*: *locks*-μ μ = $\bigcup$ *set* (*map set* μ)
  by (*auto simp add*: *list-collect-set-as-map*)

lemma *locks-c-mapset*: *locks-c c* = $\bigcup$ *set* (*map* (*snd*∘*fst*) *c*)
  by (*auto simp add*: *list-collect-set-as-map*)

end

locale *EncodedLDPN-c0* = *EncodedLDPN* + *LDPN-c0* +
— The states of the DPN are tuples of some states $'P$ and sets of locks:
constrains Δ :: $('P \times 'X\ set, 'T, 'L, 'X::finite)$ *ldpn*
constrains *c0* :: $('P \times 'X\ set, 'T)$ *conf*
constrains $\mu 0$ :: *'X list list*

— The locks encoded in the initial configuration correspond to the locks in the initial list of lock-stacks:
assumes *encoding-correct-start*:
map (*snd*∘*fst*) *c0* = *map set* $\mu 0$

begin

Reachable configurations are well-nested w.r.t. a lock-stack corresponding to the locks encoded in the control states of the processes

lemma *reachable-ec*:
⟦ (*c*,*X*)∈*reachablels*;
!!μ. ⟦*wn-c* Δ *c* μ; *X*=*locks-*μ μ; *map* (*snd*∘*fst*) *c* = *map set* μ⟧ ⟹ *P*
⟧ ⟹ *P*
apply (*unfold reachablels-def*)
apply *simp*
apply (*erule exE*)
apply (*erule ec-preserve*)
apply (*rule wellnested*)
apply (*rule encoding-correct-start*)
apply *blast*
done

Due to our assumptions, a reachable configuration always encodes the locks that are also used by the lock-sensitive semantics.

theorem *reachable-locks*: (*c*,*X*)∈*reachablels* ⟹ *locks-c c* = *X*
by (*erule reachable-ec*) (*auto simp add*: *locks-*μ*-mapset locks-c-mapset*)

14.2 Characterizing Schedulable Execution Hedges

In order to characterize schedulable execution hedges, we have to first characterize the locks allocated at the roots of an execution hedge. This can be done by deriving the locks at the roots from the control states annotated at the leafs.

fun *lock-eff* :: $('L,'X)$ *lockstep* ⇒ *'X set* ⇒ *'X set* **where**
lock-eff (*LNone nlab*) *X* = *X* |
lock-eff (*LAcq x*) *X* = *insert x X* |
lock-eff (*LRel x*) *X* = *X* − {*x*}

fun *lock-eff-inv* :: ('*L*,'*X*) *lockstep* ⇒ '*X set* ⇒ '*X set* **where**
lock-eff-inv (*LNone nlab*) *X* = *X* |
lock-eff-inv (*LAcq x*) *X* = *X* − {*x*} |
lock-eff-inv (*LRel x*) *X* = *insert x X*

fun *rlocks-t* :: ('*P*×'*X set*,'Γ,'*L*,'*X*) *lex-tree* ⇒ '*X set* **where**
rlocks-t (*NLEAF* π) = (*case* π *of* ((*p*,*X*),*w*) ⇒ *X*) |
rlocks-t (*NNOSPAWN l t*) = *lock-eff-inv l* (*rlocks-t t*) |
rlocks-t (*NSPAWN l ts t*) = *lock-eff-inv l* (*rlocks-t t*)

abbreviation *rlocks-h* :: ('*P*×'*X set*,'Γ,'*L*,'*X*) *lex-hedge* ⇒ '*X set list* **where**
rlocks-h h == *map rlocks-t h*

lemma *tsem-locks*: *tsem* Δ π *t c*' ⟹ *snd* (*fst* π) = *rlocks-t t*
apply (*induct rule*: *tsem.induct*)
apply *auto* [*1*]
apply (*drule encoding-correct-nospawn*')
apply (*case-tac l*)
apply (*auto*) [*3*]
apply (*drule encoding-correct-spawn*')
apply (*case-tac l*)
apply (*auto*) [*3*]
done

lemma *hsem-locks*: *hsem* Δ *c h c*' ⟹ *map* (*snd*∘*fst*) *c* = *rlocks-h h*
by (*induct rule*: *hsem.induct*) (*auto dest*: *tsem-locks*)

Next, we have to characterize the execution hedges with consistent acquisition histories w.r.t. the set of allocated locks.

definition *Hls h* == *cons* (*ash h*) (⋃*set* (*rlocks-h h*))

theorem *reachable-hls-char*:
assumes *A*: (*c*,*X*)∈*reachablels*   *hsem* Δ *c h c*'
shows (∃ *w*. *lsched h X w*) ⟷ *Hls h*
proof −
from *reachable-ec*[*OF A*(*1*)] **obtain** μ **where**
[*simp*]: *X* = *locks-*μ μ **and**
EC: *wn-c* Δ *c* μ   *map* (*snd* ∘ *fst*) *c* = *map set* μ
.
from *EC*(*1*) *A*(*2*) **have** *WNH*: *wn-h h* μ
by (*auto simp add*: *wnc-eq-wnch wn-c-h-def*)
have (∃ *w*. *lsched h X w*) ⟷ (∃ *w*. *lsched h* (*locks-*μ μ) *w*) **by** *simp*
also from *acqh-correct*[*OF WNH*] **have** ... = *cons* (*ash h*) (*locks-*μ μ) **.**
also have (*locks-*μ μ) = ⋃*set* (*rlocks-h h*)
by (*simp only*: *hsem-locks*[*OF A*(*2*)] *locks-*μ*-mapset EC*(*2*)[*symmetric*])
finally show *?thesis* **by** (*unfold Hls-def*)
qed

Now we can put it all together and show correctness of lock-sensitive predecessor computation

lemma *lsprestar1*:
assumes
REACH:$(c,X)\in$*reachablels* **and**
PRE: $c\in$*prehc* Δ *Hls* C'
shows $\exists\, c'\in C'.\ \exists\, ll\ X'.\ ((c,X),ll,(c',X'))\in$*ldpntrc* Δ
proof −
from *PRE* **obtain** $h\ c'$ **where** *A*: $c'\in C'$ $h\in$*Hls* *hsem* Δ $c\ h\ c'$
by (*auto elim*: *prehcE*)
from *reachable-hls-char*[*OF REACH A*(*3*)] *A*(*2*) **obtain** *ll* **where**
B: *lsched h X ll*
by (*auto simp add*: *mem-def*)
from *lsched-correct2*[*OF B A*(*3*)] *A*(*1*) **show** *?thesis* **by** *blast*
qed

lemma *lsprestar2*:
assumes
REACH:$(c,X)\in$*reachablels* **and**
MEM: $c'\in C'$ **and**
PATH: $((c,X),ll,(c',X'))\in$*ldpntrc* Δ
shows $c\in$*prehc* Δ *Hls* C'
proof −
from *lsched-correct1*[*OF PATH*] **obtain** *h* **where**
A: *hsem* Δ $c\ h\ c'$ *lsched h X ll*
by *blast*
from *reachable-hls-char*[*OF REACH A*(*1*)] *A*(*2*) **have** *B*: *Hls h* **by** *blast*
from *prehcI*[*OF - MEM A*(*1*)] *B* **show** *?thesis* **by** (*auto simp add*: *mem-def*)
qed

theorem *lsprestar*:
assumes *REACH*:$(c,X)\in$*reachablels*
shows $c\in$*prehc* Δ *Hls* $C' \longleftrightarrow (\exists\, c'\in C'.\ \exists\, ll\ X'.\ ((c,X),ll,(c',X'))\in$*ldpntrc* $\Delta)$
using *REACH lsprestar1 lsprestar2* **by** *blast*

14.3 Checking Specifications Using *prehc* Δ *Hls*

We now show that we can use our construction to check for property specifications (cf. Specification.thy).

We first have to construct a hedge-constraint for execution hedges that contain a restricted set of labels.

fun *isLab* :: $('L,'X)$ *lockstep set* $\Rightarrow$ $('Q,'\Gamma,'L,'X)$ *lex-tree* $\Rightarrow$ *bool* **where**
isLab L (*NLEAF* π) $\longleftrightarrow$ *True* |
isLab L (*NNOSPAWN l t*) $\longleftrightarrow$ $l\in L \wedge$ *isLab L t* |
isLab L (*NSPAWN l ts t*) $\longleftrightarrow$ $l\in L \wedge$ *isLab L ts* $\wedge$ *isLab L t*

abbreviation *HLab L* == { *h* . *list-all* (*isLab L*) *h*}

lemma *final-h-is-lab*[*simp*]: *final h* $\Longrightarrow$ *list-all* (*isLab L*) *h*
  apply (*induct h*)
  apply *simp*
  apply (*case-tac a*)
  apply *auto*
  done

lemma *HLab-correct*: *sched h ll* $\Longrightarrow$ *h*$\in$*HLab L* $\longleftrightarrow$ *ll*$\in$*lists L*
  by (*induct rule*: *sched.induct*) (*auto simp add*: *lists.Nil*)

lemmas *HLab-correct'* = *HLab-correct*[*OF lsched-is-sched*]

Then we can show how to check property specifications using *prehc*.

fun *mc-pre* :: ($'P\times'X$ *set*,$'\Gamma$,$'L$,$'X$) *spec* $\Rightarrow$ ($'P\times'X$ *set*,$'\Gamma$) *conf set* **where**
  mc-pre [] = *UNIV* |
  mc-pre (*SPEC-RESTRICT C* # Φ) = *C* $\cap$ *mc-pre* Φ |
  mc-pre (*SPEC-STEP L* # Φ) = *prehc* Δ (*Hls* $\cap$ *Hpre* $\cap$ *HLab L*) (*mc-pre* Φ) |
  mc-pre (*SPEC-STEPS L* # Φ) = *prehc* Δ (*Hls* $\cap$ *HLab L*) (*mc-pre* Φ)

lemma *mc-pre-correct-aux*:
  (*c*,*X*)$\in$*reachablels* $\Longrightarrow$ *c*$\in$*mc-pre* Φ $\longleftrightarrow$ (*c*,*X*)$\in$*Domain* (*spec-tr* Φ)
proof (*induct* Φ *arbitrary*: *c X*)
  case *Nil* **thus** *?case* **by** *auto*
next
  case (*Cons A* Φ)
  show *?case* **proof** (*cases A*)
    case (*SPEC-RESTRICT C*) **with** *Cons* **show** *?thesis* **by** *auto*
  next
    case (*SPEC-STEP L*)[*simp*]
    show *?thesis* **proof** (*auto simp add*: *prehc-def*)
      case (*goal1 h c'*)
    from *reachable-hls-char*[*OF Cons.prems goal1*(*5*)] *goal1*(*1*) **obtain** *w* **where**

        LS: *lsched h X w* **by** (*fastsimp simp add*: *mem-def*)
      from *Hpre-length1*[*OF goal1*(*2*) *lsched-is-sched*[*OF LS*]] **have**
        LEN: *length w* = *1* **.**
      from *HLab-correct'*[*OF LS*] *goal1*(*3*) **have** *IL*: *w*$\in$*lists L* **by** *simp*
      from *lsched-correct2*[*OF LS goal1*(*5*)] **obtain** *X'* **where**
        P: ((*c*, *X*), *w*, (*c'*, *X'*)) $\in$ *ldpntrc* Δ
        ..
      with *LEN IL* **obtain** *a* **where**
        [*simp*]: *w*=[*a*] **and**
          P1: *a*$\in$*L*   ((*c*, *X*), *a*, (*c'*, *X'*)) $\in$ *ldpntr* Δ
        by (*cases w*) *auto*
      from *P Cons.prems* **have** *P2*: (*c'*,*X'*)$\in$*reachablels*
        by (*unfold reachablels-def*) (*auto dest*: *trcl-concat trcl-one-elem*)
      from *Cons.hyps*[*OF P2*] *goal1*(*4*) **have**
        (*c'*, *X'*) $\in$ *Domain* (*LDPN.spec-tr* Δ Φ)

by *simp*
thus *?case* **using** *P1* **by** *force*
next
case (*goal2 c' X' l ch Xh*)
from *goal2*(*2*) *Cons.prems* **have** *REACH*: (*ch*,*Xh*)∈*reachablels*
by (*unfold reachablels-def*) (*auto dest*: *trcl-concat trcl-one-elem*)
from *Cons.hyps*[*OF REACH*] *goal2*(*3*) **have** *IHAPP*: *ch*∈*mc-pre* Φ **by** *auto*
from *lsched-correct1*[*OF trcl-one-elem*[*OF goal2*(*2*)]] **obtain** *h* **where**
H: *hsem* Δ *c h ch* *lsched h X* [*l*]
by *blast*
from *Hpre-length2*[*OF lsched-is-sched*[*OF H*(*2*)]] **have**
HPRE: *h*∈*Hpre*
by *simp*
from *reachable-hls-char*[*OF Cons.prems H*(*1*)] *H*(*2*) **have**
HLS: *h*∈*Hls*
by (*auto simp add*: *mem-def*)
from *HLab-correct'*[*OF H*(*2*), *of L*] *goal2*(*1*) **have**
list-all (*isLab L*) *h*
by *auto*
with *HLS HPRE IHAPP H*(*1*) **show** *?case* **by** *blast*
qed
next
case (*SPEC-STEPS L*)[*simp*]
show *?thesis* **proof** (*auto simp add*: *prehc-def*)
case (*goal1 h c'*)
from *reachable-hls-char*[*OF Cons.prems goal1*(*4*)] *goal1*(*1*) **obtain** *w* **where**

LS: *lsched h X w*
by (*fastsimp simp add*: *mem-def*)
from *HLab-correct'*[*OF LS*] *goal1*(*2*) **have** *IL*: *w*∈*lists L* **by** *simp*
from *lsched-correct2*[*OF LS goal1*(*4*)] **obtain** *X'* **where**
P: ((*c*, *X*), *w*, (*c'*, *X'*)) ∈ *ldpntrc* Δ **..**
from *P Cons.prems* **have** *P2*: (*c'*,*X'*)∈*reachablels*
by (*unfold reachablels-def*) (*auto dest*: *trcl-concat*)
from *Cons.hyps*[*OF P2*] *goal1*(*3*)
have (*c'*, *X'*) ∈ *Domain* (*LDPN.spec-tr* Δ Φ) **by** *simp*
thus *?case* **using** *IL P* **by** *force*
next
case (*goal2 c' X' ll ch Xh*)
from *goal2*(*2*) *Cons.prems* **have** *REACH*: (*ch*,*Xh*)∈*reachablels*
by (*unfold reachablels-def*) (*auto dest*: *trcl-concat*)
from *Cons.hyps*[*OF REACH*] *goal2*(*3*) **have** *IHAPP*: *ch*∈*mc-pre* Φ **by** *auto*
from *lsched-correct1*[*OF goal2*(*2*)] **obtain** *h* **where**
H: *hsem* Δ *c h ch* *lsched h X ll*
by *blast*
from *reachable-hls-char*[*OF Cons.prems H*(*1*)] *H*(*2*) **have** *HLS*: *h*∈*Hls*
by (*auto simp add*: *mem-def*)
from *HLab-correct'*[*OF H*(*2*), *of L*] *goal2*(*1*)
have *list-all* (*isLab L*) *h* **by** *auto*

```
      with HLS IHAPP H(1) show ?case by blast
    qed
  qed
qed

theorem mc-pre-correct: c0∈mc-pre Φ ⟷ model-check-ref Φ
  using mc-pre-correct-aux[of c0 locks-μ μ0 Φ, simplified]
  by (unfold model-check-ref-def)

end

end
```

15 Monitors (aka Block-Structured Locks)

```
theory Monitors
imports LockSem WellNested As-hc
begin
```

We model monitors by binding locks to stack symbols, and making some restrictions on rules:

- A rule labeled by *LNone* must not change the allocated locks, nor must it push or pop stack symbols associated with locks.
- An acquisition rule must be a rule that pushes a stack-symbol with the acquired lock, and does not change the locks of the stacl-symbol at the bottom.
- A release rule must be a rule that pops a stack-symbol with the released lock.

One purpose of this theory is, that it gives strong evidence that our model is not too restrictive. This is done by defining an introduction rule for encoded DPNs with initial configurations that only depends on local properties of the rules and the initial configuration.

— Lock-stack encoded into stack

```
definition lstackm-s :: ('Γ ⇀ 'X) ⇒ 'Γ ⇒ 'X list where
  lstackm-s mon γ = (case mon γ of None ⇒ [] | Some x ⇒ [x])

lemma lstackm-s-simps[simp]:
  mon γ = None ⟹ lstackm-s mon γ = []
  mon γ = Some x ⟹ lstackm-s mon γ = [x]
  by (auto simp add: lstackm-s-def)
```

fun *lstackm* :: ($'\Gamma \rightharpoonup 'X$) $\Rightarrow$ $'\Gamma$ *list* $\Rightarrow$ $'X$ *list* **where**
lstackm mon [] = [] |
lstackm mon (γ#*s*) = *lstackm-s mon* γ @ *lstackm mon s*

lemma *lstackm-conc*[*simp*]:
lstackm mon (*s*@*s'*) = *lstackm mon s* @ *lstackm mon s'*
by (*induct s*) *auto*

lemma *lstack-spawn-empty*[*simp*]:
⟦ ($\forall\gamma s\in$*set w. mon* γs=*None*) ⟧ $\Longrightarrow$ *lstackm mon w* = []
by (*induct w*) (*auto*)

locale *MDPN* = *EncodedLDPN* +
constrains
Δ :: ($'P\times'X$ *set*,$'\Gamma$,$'L$,$'X$::*finite*) *ldpn*
fixes *mon* :: $'\Gamma \Rightarrow 'X$ *option* — Maps stack symbols to associated monitors

assumes
locks-lnone-pop-nospawn:
(p,$\gamma \hookrightarrow_{LNone\ a}$ p',[])$\in\Delta$ $\Longrightarrow$ *mon* γ = *None* **and**
locks-lnone-pop-spawn:
(p,$\gamma \hookrightarrow_{l}$ *ps*,*ws* $\sharp$ p',[])$\in\Delta$ $\Longrightarrow$ *mon* γ = *None* **and**
locks-lnone-nospawn:
(p,$\gamma \hookrightarrow_{LNone\ a}$ p',*w*@[γ'])$\in\Delta$ $\Longrightarrow$ *mon* γ' = *mon* γ $\wedge$
($\forall\gamma s\in$*set w. mon* γs=*None*) **and**
locks-lnone-spawn:
(p,$\gamma \hookrightarrow_{l}$ *ps*,*ws* $\sharp$ p',*w*@[γ'])$\in\Delta$ $\Longrightarrow$ *mon* γ' = *mon* γ $\wedge$
($\forall\gamma s\in$*set w. mon* γs=*None*) **and**
locks-spawn:
(p,$\gamma \hookrightarrow_{l}$ *ps*,*ws* $\sharp$ p',*w*)$\in\Delta$ $\Longrightarrow$ ($\forall\gamma s\in$*set ws. mon* γs=*None*) **and**
locks-acquire:
⟦ (p,$\gamma \hookrightarrow_{LAcq\ x}$ p',*w*)$\in\Delta$;
!!*w'* $\gamma 2$ $\gamma 1$. ⟦ *w*=*w'*@[$\gamma 1$,$\gamma 2$]; *mon* $\gamma 2$ = *mon* γ; *mon* $\gamma 1$ = *Some x*;
($\forall\gamma s\in$*set w'. mon* γs=*None*)
⟧ $\Longrightarrow$ *P*
⟧ $\Longrightarrow$ *P* **and**
locks-release:
(p,$\gamma \hookrightarrow_{LRel\ x}$ p',*w*)$\in\Delta$ $\Longrightarrow$ *w*=[] $\wedge$ *mon* γ = *Some x*

begin

abbreviation *lstack-s* == *lstackm-s mon*
abbreviation *lstack* == *lstackm mon*

lemma *lstack-lnone-nospawn*:
⟦(p,$\gamma \hookrightarrow_{LNone\ a}$ p',*w*)$\in\Delta$⟧ $\Longrightarrow$ *lstack* (γ#*r*) = *lstack* (*w*@*r*)
apply (*cases w rule*: *rev-cases*)
apply *simp*

apply (*drule locks-lnone-pop-nospawn*)
apply (*simp*)
apply (*simp*)
apply (*drule locks-lnone-nospawn*)
apply (*cases mon* γ)
apply (*simp-all*)
done

lemma *lstack-lnone-spawn*:
$[\![(p,\gamma \hookrightarrow_a ps,ws \sharp p',w) \in \Delta]\!] \Longrightarrow lstack\ (\gamma \# r) = lstack\ (w @ r)$
apply (*cases w rule*: *rev-cases*)
apply *simp*
apply (*drule locks-lnone-pop-spawn*)
apply (*simp*)
apply (*simp*)
apply (*drule locks-lnone-spawn*)
apply (*cases mon* γ)
apply (*simp-all*)
done

lemma *well-nested-t*:
assumes *CONS*: *distinct* (*lstack* (*snd* π))
assumes *H*: *tsem* Δ π *t* c'
assumes *COINC*: *snd* (*fst* π) = *set* (*lstack* (*snd* π))
shows *wn-t'* *t* (*lstack* (*snd* π))
using *H CONS COINC*
proof (*induct rule*: *tsem.induct*)
case *tsem-leaf* **thus** *?case* **by** (*auto intro*: *wn-t.intros*)
next
case (*tsem-spawn p* γ *l ps ws* p' *w ts cs r t* c')
from *spawn-no-locks*[*OF tsem-spawn.hyps*(*1*)] **obtain** *la* **where**
[*simp*]: *l*=*LNone la*
by *auto*
from *locks-spawn*[*OF tsem-spawn.hyps*(*1*)] **have**
[*simp*]: *lstack ws* = []
by (*simp add*: *lstack-spawn-empty*)
from *encoding-correct-spawn2'*[*OF tsem-spawn.hyps*(*1*)] **have**
[*simp*]: *snd ps* = {} **.**
from *tsem-spawn.hyps*(*3*) **have**
IHAPP1: *wn-t'* *ts* (*lstack* (*snd* (*ps*, *ws*)))
by *simp*
moreover
from *lstack-lnone-spawn*[*OF tsem-spawn.hyps*(*1*)] **have**
LSF[*simplified*, *simp*]: *lstack* (γ # *r*) = *lstack* (*w* @ *r*) **.**
moreover from *encoding-correct-spawn'*[*OF tsem-spawn.hyps*(*1*)] **have**
[*simp*]: *snd p* = *snd* p'
by *simp*
from *tsem-spawn.prems tsem-spawn.hyps*(*5*) *LSF* **have**

IHAPP2: *wn-t′ t* (*lstack* (*w*@*r*))
by *simp*
ultimately show *?case* **by** *simp*
next
case (*tsem-nospawn p* γ *l p′ w r t c′*)
show *?case*
proof (*cases l*)
case (*LNone la*)[*simp*]
from *lstack-lnone-nospawn tsem-nospawn.hyps*(*1*) **have**
[*simplified*, *simp*]: *lstack* (γ#*r*) = *lstack* (*w*@*r*)
by *simp*
moreover from *encoding-correct-nospawn′*[*OF tsem-nospawn.hyps*(*1*)] **have**
[*simp*]: *snd p* = *snd p′*
by *simp*
from *tsem-nospawn.prems tsem-nospawn.hyps*(*3*) **have**
IHAPP: *wn-t′ t* (*lstack* (*w*@*r*))
by *simp*
thus *?thesis* **by** *simp*
next
case (*LAcq x*)[*simp*]
from *tsem-nospawn.hyps*(*1*)[*simplified*] **show** *?thesis*
proof (*cases rule*: *locks-acquire*[*consumes 1*, *case-names C*])
case (*C w′* γ*2* γ*1*)
note [*simp*] = *C*(*1*)
from *C*(*4*) **have** [*simp*]: *lstack w′* = [] **by** *simp*
from *C*(*3*) **have** [*simp*]: *lstack-s* γ*1* = [*x*] **by** *simp*
from *C*(*2*) **have** [*simp*]: *lstack-s* γ*2* = *lstack-s* γ
by (*cases mon* γ) *simp-all*

from *encoding-correct-nospawn′*[*OF tsem-nospawn.hyps*(*1*)] **have**
XNSP: *x*∉*snd p* **and**
SP′F[*simp*]: *snd p′* = *insert x* (*snd p*)
by *auto*
from *tsem-nospawn.prems*(*2*) *XNSP* **have**
XNIS: *x*∉*set* (*lstack* (γ#*r*))
by *simp*
from *XNIS*[*simplified*] *tsem-nospawn.prems*(*1*)[*simplified*] **have**
P1: *distinct* (*lstack* (*w*@*r*))
by (*simp*)
from *tsem-nospawn.prems*(*2*)[*simplified*] *tsem-nospawn.hyps P1* **have**
IHAPP: *wn-t′ t* (*lstack* (*w*@*r*))
by *simp*
thus *?thesis* **using** *XNIS* **by** *simp*
qed
next
case (*LRel x*)[*simp*]
from *tsem-nospawn.hyps*(*1*)[*simplified*] *locks-release* **have**
[*simp*]: *w*=[]   *mon* γ = *Some x*
by *auto*

```
      from encoding-correct-nospawn'[OF tsem-nospawn.hyps(1)] have
        XNSP: x∉snd p' and SPF[simp]: snd p' = snd p − {x}
        by auto
      from tsem-nospawn.prems(1)[simplified] have
        P1: distinct (lstack (w@r))
        by (simp)
      from tsem-nospawn.prems have P2: snd p' = set (lstack (w @ r)) by simp
      from tsem-nospawn.hyps P1 P2 have IHAPP: wn-t' t (lstack (w@r)) by simp
      thus ?thesis using tsem-nospawn.prems(1) by simp
    qed
  qed

  lemma well-nested-h:
    assumes CONS: cons-μ (map (lstack ∘ snd) c)
    assumes H: hsem Δ c h c'
    assumes COINC: map (snd∘fst) c = map (set∘lstack∘snd) c
    shows wn-h h (map (lstack ∘ snd) c)
    using H CONS COINC
    by (induct rule: hsem.induct) (auto intro: well-nested-t)


  theorem well-nested:
    assumes CONS: cons-μ (map (lstack ∘ snd) c)
    assumes COINC: map (snd∘fst) c = map (set∘lstack∘snd) c
    shows wn-c Δ c (map (lstack ∘ snd) c)
    apply (simp add: wnc-eq-wnch)
    apply (unfold wn-c-h-def)
    apply (blast intro: well-nested-h[OF CONS - COINC])
    done
```

This theorem can be used to show that an MDPN along with a consistent start configuration is a DPN with well-nested lock usage, as described by the locale *EncodedLDPN-c0*.

```
  theorem EncodedLDPN-c0-intro[intro?]:
    assumes start-config-cons: cons-μ μ0
    assumes start-config-coinc: map (snd∘fst) c0 = map set μ0
    assumes start-config-match: map (lstack ∘ snd) c0 = μ0
    shows EncodedLDPN-c0 Δ c0 μ0
  proof
    from start-config-coinc start-config-match[symmetric] have
      map (snd∘fst) c0 = map set (map (lstack ∘ snd) c0)
      by simp
    also have ... = map (set ∘ lstack ∘ snd) c0 by (simp add: map-compose)
    finally show wn-c Δ c0 μ0
      using start-config-cons start-config-match by (blast intro: well-nested)
  qed (rule start-config-coinc)

end
```

theorem *EncodedLDPN-c0-intro-external*:
  assumes *MDPN*: *MDPN* Δ *mon*
  assumes *start-config-cons*: *cons-μ μ0*
  assumes *start-config-coinc*: *map* (*snd*∘*fst*) *c0* = *map set μ0*
  assumes *start-config-match*: *map* (*lstackm mon* ∘ *snd*) *c0* = *μ0*
  shows *EncodedLDPN-c0* Δ *c0 μ0*
proof −
  interpret *MDPN*[Δ *mon*] **using** *MDPN* **.**
  from *EncodedLDPN-c0-intro*[*OF start-config-cons start-config-coinc start-config-match*]
  show *?thesis* **.**
qed

15.1 Non-Trivial Instance of a Well-Nested DPN

In this section, we define a non-trivial Well-nested DPN by hand. This gives strong evidence that our model assumptions are not too restrictive.

We start by introducing some finite set of locks that we can use in our programs:

typedef *t-my-locks* = {*1..6*::*nat*} **by** *auto*

instance *t-my-locks*::*finite*
proof (*intro-classes*)
  have *Rep-t-my-locks* ‘ *UNIV* ⊆ *t-my-locks* **using** *Rep-t-my-locks* **by** *auto*
  moreover have *finite t-my-locks* **by** (*unfold t-my-locks-def*) *auto*
  ultimately show *finite* (*UNIV*::*t-my-locks set*)
    apply (*rule-tac f*=*Rep-t-my-locks* **in** *finite-imageD*)
    apply (*drule finite-subset*)
    apply *assumption+*
    apply (*rule injI*)
    apply (*simp add*: *Rep-t-my-locks-inject*)
    done
qed

definition *l1* :: *t-my-locks* **where** *l1* = *Abs-t-my-locks* (*1*::*nat*)
definition *l2* :: *t-my-locks* **where** *l2* = *Abs-t-my-locks* (*2*::*nat*)

lemma [*simp*, *intro*!]: *l1*≠*l2*   *l2*≠*l1*
  apply (*unfold l1-def l2-def*)
  apply (*auto simp add*: *Abs-t-my-locks-inject t-my-locks-def*)
  done

The following rules correspond to a by-hand translation of the (nonsense) program:

```
procedure p1:
  sync l1 {
```

```
      sync l2 {
        spawn p1
        spawn p2
      }
    }

  procedure p2:
    if ? then
      spawn p2
      call p2
    else
      sync l2 {
        sync l1 {
          spawn p1
        }
      }
```

definition *my*Δ :: (*nat* × *t-my-locks set*,*nat*,*unit*,*t-my-locks*) *ldpn* **where**
*my*Δ = {
$((0,\{\}),1 \hookrightarrow_{LAcq\ l1} (0,\{l1\}),[2,3]),$
$((0,\{l1\}),2 \hookrightarrow_{LAcq\ l2} (0,\{l1,l2\}),[4,5]),$
$((0,\{l1,l2\}),4 \hookrightarrow_{LNone\ ()} (0,\{\}),[1]\sharp(0,\{l1,l2\}),[6]),$
$((0,\{l1,l2\}),6 \hookrightarrow_{LNone\ ()} (0,\{\}),[11]\sharp(0,\{l1,l2\}),[7]),$
$((0,\{l1,l2\}),7 \hookrightarrow_{LRel\ l2} (0,\{l1\}),[]),$
$((0,\{l1\}),5 \hookrightarrow_{LRel\ l1} (0,\{\}),[]),$
$((0,\{\}),3 \hookrightarrow_{LNone\ ()} (0,\{\}),[]),$

$((0,\{\}),11 \hookrightarrow_{LNone\ ()} (0,\{\}),[11]\sharp(0,\{\}),[12]),$
$((0,\{\}),12 \hookrightarrow_{LNone\ ()} (0,\{\}),[11,13]),$
$((0,\{\}),11 \hookrightarrow_{LAcq\ l2} (0,\{l2\}),[14,13]),$
$((0,\{l2\}),14 \hookrightarrow_{LAcq\ l1} (0,\{l1,l2\}),[16,17]),$
$((0,\{l1,l2\}),16 \hookrightarrow_{LNone\ ()} (0,\{\}),[1]\sharp(0,\{l1,l2\}),[18]),$
$((0,\{l1,l2\}),18 \hookrightarrow_{LRel\ l1} (0,\{l2\}),[]),$
$((0,\{l2\}),17 \hookrightarrow_{LRel\ l2} (0,\{\}),[]),$
$((0,\{\}),13 \hookrightarrow_{LNone\ ()} (0,\{\}),[])$
}

definition *my-mon* :: *nat* ⇒ *t-my-locks option* **where**
my-mon s = (
if s=1 then None
else if s=2 then Some l1
else if s=3 then None
else if s=4 then Some l2
else if s=5 then Some l1

else if s=6 then Some l2
else if s=7 then Some l2
else if s=11 then None
else if s=12 then None
else if s=13 then None
else if s=14 then Some l2
else if s=15 then None
else if s=16 then Some l1
else if s=17 then Some l2
else if s=18 then Some l1
else None
)

It is straightforward to show that this is an MDPN

interpretation *MDPN*[*my*Δ *my-mon*]
apply (*unfold-locales*)
apply (*unfold my*Δ*-def*)
apply *auto*
apply (*unfold my-mon-def*)
apply *simp-all*
apply *blast+*
done

And with the stuff proven above, we also get that this program is a well-nested LDPN w.r.t. the start configuration [((*0*::′*a*, {}), [*1*::′*c*])], which corresponds to starting with procedure `p1`.

interpretation *EncodedLDPN-c0*[*my*Δ [((*0*,{}),[*1*])] [[]]]
apply *rule*
apply *auto*
apply (*unfold lstackm-s-def my-mon-def*)
apply *simp*
done

end

16 Conclusion

We formalized a tree-based semantics for DPNs, where executions are modeled as hedges, that reflect the ordering of steps of each process and the causality due to process creation, but enforce no ordering between steps of processes running in parallel. We have shown how to efficiently compute predecessor sets of regular sets of configurations with tree-regular constraints on the execution hedges, by encoding a hedge-automaton into the DPN, thus reducing the problem to unconstrained predecessor set computation.

We have then formalized a generalization of acquisition histories to DPNs, and have shown its correctness. We have demonstrated how to use the generalized acquisistion histories to describe the set of execution hedges, that have a lock-sensitive schedule, as a regular set. Thus we could use the techniques for hedge-constrained predecessor set computation to also compute lock-sensitive, hedge-constrained predecessor sets. Finally, we have defined a class of properties that can be computed using cascaded predecessor computations, and have applied our techniques to decide those properties for DPNs.

16.1 Trusted Code Base

In this section we shortly characterize on what our formal proof depends, i.e. how to interpret the information contained in this formal proof and the fact that it is accepted by Isabelle.

First of all, you have to trust the theorem prover and its axiomatization of HOL, the ML-platform, the operating system software and the hardware it runs on. All this components are able to cause false theorems to be proven.

Next, most of the theorems proven here have some implicit and explicit assumptions. The most critical assumptions are the assumptions of the locales, namely *DPN*, *LDPN*, *LDPN_c0*, and *encodedLDPN*. It is not formally provebn that these assumptions make sense, and the locales really admit useful models. In Section 15 we give an example for a non-trivial DPN and formally prove that it satisfies our assumptions. This gives some evidence that our assumptions are not too restrictive.

The next crucial point – already discussed in the introduction – is, that we at some points claim that our methods are executable. However, we do not derive any executable code, and even if we did, the Isabelle code-generator can only guarantee *partial* correctness, i.e. correctness under the assumption of termination. At this point, the belief in the existence of executable methods depends on the belief in that the model-checking functions, i.e. the function *mc-pre* in *As-hc.thy* is effective for regular sets, and the result is a regular set again, such that we can check $c_0 \in \mathsf{mc} - \mathsf{pre}\Phi$ as required by Theorem *mc-pre-correct*, using the saturation algorithm of [2].

However, we prove some theorems that support this belief by showing how the required operations can be decomposed to operations that are well-known to be effective and to preserve regularity.

References

[1] A. Bouajjani, J. Esparza, S. Schwoon, and J. Strejcek. Reachability analysis of multithreaded software with asynchronous communication. In *Proc. of FSTTCS'05*, pages 348–359. Springer, 2005.

[2] A. Bouajjani, M. Müller-Olm, and T. Touili. Regular symbolic analysis of dynamic networks of pushdown systems. In *Proc. of CONCUR'05*. Springer, 2005.

[3] V. Kahlon and A. Gupta. An automata-theoretic approach for model checking threads for LTL properties. In *Proc. of LICS 2006*, pages 101–110. IEEE Computer Society, 2006.

[4] V. Kahlon, F. Ivancic, and A. Gupta. Reasoning about threads communicating via locks. In *Proc. of CAV 2005*, pages 505–518. Springer, 2005.