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*: $(c,[],c') \in trcl\ t \Longrightarrow (c=c')$
⟨*proof*⟩
lemma *trcl-empty-simp*[*simp*]: $(c,[],c') \in trcl\ t = (c=c')$
⟨*proof*⟩

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

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

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*⟩

lemma *trcl-unconcat*: $!!\ c\ .\ (c,w1@w2,c') \in trcl\ t$

$\Longrightarrow \exists\, ch\ .\ (c,w1,ch)\in trcl\ t \wedge (ch,w2,c')\in trcl\ t$
⟨*proof*⟩

2.2.1 Appending of elements to paths

lemma *trcl-rev-cons*: $[\![\ (c,w,ch)\in trcl\ T;\ (ch,e,c')\in T\ ]\!] \Longrightarrow (c,w@[e],c')\in trcl\ T$
⟨*proof*⟩
lemma *trcl-rev-uncons*: $(c,w@[e],c')\in trcl\ T$
$\Longrightarrow \exists\, ch.\ (c,w,ch)\in trcl\ T \wedge (ch,e,c')\in T$
⟨*proof*⟩
lemma *trcl-rev-uncons-cases*: $[\![$
$(c,w@[e],c')\in trcl\ T;$
$!!ch.\ [\![(c,w,ch)\in trcl\ T;\ (ch,e,c')\in T]\!] \Longrightarrow P$
$]\!] \Longrightarrow P$
⟨*proof*⟩

lemma *trcl-rev-induct*[*induct set*, *consumes 1*, *case-names empty snoc*]: $!!\ c'.\ [\![$
$(c,w,c')\in trcl\ S;$
$!!c.\ P\ c\ []\ c;$
$!!c\ w\ c'\ e\ c''.\ [\![\ (c,w,c')\in trcl\ S;\ (c',e,c'')\in S;\ P\ c\ w\ c'\ ]\!] \Longrightarrow P\ c\ (w@[e])\ c''$
$]\!] \Longrightarrow P\ c\ w\ c'$
⟨*proof*⟩
lemma *trcl-rev-cases*: $!!c\ c'.\ [\![$
$(c,w,c')\in trcl\ S;$
$[\![w=[];\ c=c']\!]\Longrightarrow P;$
$!!ch\ e\ wh.\ [\![\ w=wh@[e];\ (c,wh,ch)\in trcl\ S;\ (ch,e,c')\in S\ ]\!]\Longrightarrow P$
$]\!] \Longrightarrow P$
⟨*proof*⟩

lemma *trcl-cons2*: $[\![\ (c,e,ch)\in T;\ (ch,f,c')\in T\ ]\!] \Longrightarrow (c,[e,f],c')\in trcl\ T$
⟨*proof*⟩

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*: $!!A\ B.\ A\subseteq B \Longrightarrow trcl\ A\subseteq trcl\ B$
⟨*proof*⟩

lemma *trcl-inter-mono*: $x\in trcl\ (S\cap R) \Longrightarrow x\in trcl\ S \quad x\in trcl\ (S\cap R) \Longrightarrow x\in trcl\ R$
⟨*proof*⟩

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*]: ⟦
  (*c*,*w*,*c'*)∈*trcl S*;
  ⟦*c*=*c'*; *w*=[]⟧ ⟹ *P*;
  ⟦*c*∈*Domain S*; *c'*∈*Range* (*Range S*)⟧⟹*P*
 ⟧ ⟹ *P*
 ⟨*proof*⟩

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 *'Γ*, where each pushdown rule may spawn additional processes. Rules are labeled by elements of type *'L*

datatype (*'P*,*'Γ*,*'L*) *pushdown-rule* =
 NOSPAWN *'P* *'Γ* *'L* *'P* *'Γ list* (-,- ↪ - -,- *51*) |
 SPAWN *'P* *'Γ* *'L* *'P* *'Γ list* *'P* *'Γ list* (-,- ↪ - -,- ♯ -,- *51*)

notation *NOSPAWN* (-,- ↪ - -,- *51*)
notation *SPAWN* (-,- ↪ - -,- ♯ -,- *51*)

types (*'Q*,*'Γ*,*'L*) *dpn* = (*'Q*,*'Γ*,*'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*,*'Γ*,*'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* $\Rightarrow$ $('Q,'\Gamma)$ *pconf* $\Rightarrow$ $('Q,'\Gamma,'L)$ *ex-tree* $\Rightarrow$ $('Q,'\Gamma)$ *conf* $\Rightarrow$ *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* Δ $(p',w@r)$ t $c']\!] \Longrightarrow$
      tsem Δ $(p,\gamma\#r)$ (*NNOSPAWN* l t) c' |
  tsem-spawn:
    $[\![(p,\gamma \hookrightarrow_l ps,ws \sharp p',w) \in \Delta;$ *tsem* Δ (ps,ws) *ts cs*; *tsem* Δ $(p',w@r)$ t $c']\!] \Longrightarrow$
      tsem Δ $(p,\gamma\#r)$ (*NSPAWN* l *ts* t) $(cs@c')$

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

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

lemma *hsem-empty-h*[*simp*]:
  hsem Δ c [] $c' \longleftrightarrow c=[] \wedge c'=[]$
  ⟨*proof*⟩

lemma *hsem-length*: *hsem* Δ c h $c' \Longrightarrow$ *length* c = *length* h
  ⟨*proof*⟩

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

lemmas *hsem-cons-single* = *hsem-cons*[**where** *cf'*=[π'], *simplified*, *standard*]

lemma *hsem-conc*: ⟦*hsem* Δ *c1 h1 c1'*; *hsem* Δ *c2 h2 c2'*⟧ ⟹
hsem Δ (*c1*@*c2*) (*h1*@*h2*) (*c1'*@*c2'*)
⟨*proof*⟩

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* Δ π *t* []
⟨*proof*⟩
lemma *hsem-empty-simps1*[*simp*]:
hsem Δ [] *h c'* ⟷ (*h*=[] ∧ *c'*=[])
hsem Δ *c h* [] ⟷ (*c*=[] ∧ *h*=[])
⟨*proof*⟩

lemma *hsem-id*[*simp*, *intro*!]: *hsem* Δ *c* (*map NLEAF c*) *c*
⟨*proof*⟩
lemmas *hsem-id'*[*simp*, *intro*!] = *hsem-id*[*of* - π#*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*⟩

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*
⟨*proof*⟩

lemma *hsem-single*:
⟦*hsem* Δ [π] *h c'*; !!*t*. ⟦ *h*=[*t*]; *tsem* Δ π *t c'* ⟧ ⟹ *P* ⟧ ⟹ *P*
⟨*proof*⟩

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*
⟨*proof*⟩

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'*

$\rrbracket \Longrightarrow P$
$\rrbracket \Longrightarrow P$
⟨*proof*⟩

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

lemma *hsem-split-h'*: $[\![hsem\ \Delta\ c\ (h1@h2)\ c']\!] \Longrightarrow$
$\exists\, c1\ c2\ c1'\ c2'.\ c{=}c1@c2 \wedge c'{=}c1'@c2' \wedge$
$hsem\ \Delta\ c1\ h1\ c1' \wedge hsem\ \Delta\ c2\ h2\ c2'$
⟨*proof*⟩

lemma *hsem-split-h*:
$[\![\ hsem\ \Delta\ c\ (h1@h2)\ c';$
$!!c1\ c2\ c1'\ c2'.$
$[\![c{=}c1@c2;\ c'{=}c1'@c2';\ hsem\ \Delta\ c1\ h1\ c1';\ hsem\ \Delta\ c2\ h2\ c2']\!] \Longrightarrow P$
$\rrbracket \Longrightarrow P$
⟨*proof*⟩

lemma *hsem-single-h*:
$[\![hsem\ \Delta\ c\ [t]\ c';\ !!p.\ [\![\ c{=}[p];\ tsem\ \Delta\ p\ t\ c'\ ]\!] \Longrightarrow P\ ]\!] \Longrightarrow P$
⟨*proof*⟩

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

lemma *hsem-lel-h*: $[\![\ hsem\ \Delta\ c\ (h1@t\#h2)\ c';$
$!!c1\ p\ c2\ c1'\ ct'\ c2'.\ [\![$
$c{=}c1@p\#c2;\ c'{=}c1'@ct'@c2';$
$hsem\ \Delta\ c1\ h1\ c1';\ tsem\ \Delta\ p\ t\ ct';\ hsem\ \Delta\ c2\ h2\ c2'$
$\rrbracket \Longrightarrow P$
$\rrbracket \Longrightarrow P$
⟨*proof*⟩

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*)
⟨*proof*⟩

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*]
⟨*proof*⟩

lemma *final-hsem-nostep*: ⟦*final h*; *hsem* Δ *c h c'*⟧ ⟹ *c'*=*c*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *sched-rel-alt2*: ⟦(*h*,*ll*,*h'*)∈*trcl sched-rel*; *final h'*⟧ ⟹ *sched h ll*
⟨*proof*⟩

lemma *sched-alt*: *sched' h ll* ⟷ *sched h ll*
⟨*proof*⟩

We now show some basic facts about the scheduler.

lemma *sched-empty-seq*[*simp*]: *sched h* [] ⟷ *final h*
⟨*proof*⟩

lemma *sched-empty-hedge*[*simp*]: *sched* [] *ll* ⟷ *ll*=[]
⟨*proof*⟩

lemma *sched-empty-empty*[*simp*, *intro*!]: *sched* [] [] ⟨*proof*⟩

lemma *sched-final-simp*[*simp*]: *final h* ⟹ *sched h c* ⟷ *c*=[]
⟨*proof*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

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* ⟹ *P*⟧ ⟹ *P*
⟨*proof*⟩

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*′)∈*dpntrc* Δ ⟹ ∃*h*. *hsem* Δ *c h c*′ ∧ *sched h ll*
⟨*proof*⟩

lemma *sched-correct2*: ⟦ *sched h ll*; *hsem* Δ *c h c*′ ⟧ ⟹ (*c*,*ll*,*c*′)∈*dpntrc* Δ
⟨*proof*⟩

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

theorem *sched-correct*: $(c,ll,c')\in dpntrc\ \Delta \longleftrightarrow (\exists\, h.\ hsem\ \Delta\ c\ h\ c' \wedge sched\ h\ ll)$
⟨*proof*⟩

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

lemma *obtain-schedule*:
$[\![\ hsem\ \Delta\ c\ h\ c';$
$\quad !!ll.\ [\![(c,ll,c')\in dpntrc\ \Delta;\ sched\ h\ ll\ ]\!] \Longrightarrow P$
$]\!] \Longrightarrow P$
⟨*proof*⟩

5 Predecessor Sets

Following [2], we define the set of immediate predecessors $pre\ \Delta\ C$ and predecessors $pre^*\ \Delta\ C$ 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 $pre\ \Delta\ C' == \{\ c\ .\ \exists\, l\ c'.\ c'\in C' \wedge (c,l,c') \in dpntr\ \Delta\ \}$
definition *pre-star* (pre^*) **where**
$pre^*\ \Delta\ C' == \{\ c\ .\ \exists\, ll\ c'.\ c'\in C' \wedge (c,ll,c') \in dpntrc\ \Delta\ \}$

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 $prehc\ \Delta\ H\ C' == \{\ c\ .\ \exists\, h\ c'.\ h\in H \wedge c'\in C' \wedge hsem\ \Delta\ c\ h\ c'\ \}$

lemma *prehcI*: $[\![h\in H;\ c'\in C';\ hsem\ \Delta\ c\ h\ c']\!] \Longrightarrow c\in prehc\ \Delta\ H\ C'$
⟨*proof*⟩

lemma *prehcE*:
$[\![c\in prehc\ \Delta\ H\ C';\ !!h\ c'.\ [\![\ h\in H;\ c'\in C';\ hsem\ \Delta\ c\ h\ c'\ ]\!] \Longrightarrow P\ ]\!] \Longrightarrow P$
⟨*proof*⟩

The hedge-constrained predecessor set is monotonic in the constraint

lemma *prehc-mono*: $H\subseteq H' \Longrightarrow prehc\ \Delta\ H\ C' \subseteq prehc\ \Delta\ H'\ C'$
⟨*proof*⟩

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

lemma *prehc-triv-is-pre-star*: *prehc* Δ *UNIV* $C' = pre^*\ \Delta\ C'$
⟨*proof*⟩

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

lemma *prehc-subset-pre-star*: *prehc* Δ *H* $C' \subseteq pre^*\ \Delta\ C'$
⟨*proof*⟩

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

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

lemma *HpreI-nospawn*:
⟦*final h1*; *final h2*; *final-t t'*⟧ ⟹ *h1@NNOSPAWN lab t'#h2* ∈ *Hpre*
⟨*proof*⟩

lemma *HpreI-spawn*:
⟦*final h1*; *final h2*; *final-t ts*; *final-t t'*⟧ ⟹ *h1@NSPAWN lab ts t'#h2* ∈ *Hpre*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

lemma *Hpre-length2*: ⟦*sched h ll*; *length ll = 1*⟧ ⟹ *h*∈*Hpre*
⟨*proof*⟩

theorem *Hpre-length*: *sched h ll* ⟹ *h*∈*Hpre* ⟷ *length ll = 1*
⟨*proof*⟩

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

lemma *Hpre-correct1*: *c*∈*prehc* Δ *Hpre* C' ⟹ *c*∈*pre* Δ C'

⟨*proof*⟩

lemma *Hpre-correct2*: $c \in pre\ \Delta\ C' \Longrightarrow c \in prehc\ \Delta\ Hpre\ C'$
⟨*proof*⟩

theorem *Hpre-correct*: $prehc\ \Delta\ Hpre = pre\ \Delta$
⟨*proof*⟩

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* :: $('Q,'\Gamma)$ *conf* ⇒ $('Q,'\Gamma)$ *cl* **where**
c2cl c == *concat* (*map pc2cl* c)

abbreviation *c2cl-abbrv* :: $('Q,'\Gamma)$ *conf* ⇒ $('Q,'\Gamma)$ *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,γ ↪$_l$ p',w)∈Δ ⟧ ⟹
(c1@[CTRL p, STACK γ]@c2,
l,
c1@CTRL p'#(map STACK w)@c2
) ∈ cltr Δ |
cltr-spawn:
⟦ (p,γ ↪$_l$ ps,ws ♯ p',w)∈Δ ⟧ ⟹
(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* ⟨*proof*⟩

6.2.1 Representation of Single Processes

lemma *pc2cl-not-empty*[*simp*]: *pc2cl π ≠ []* ⟨*proof*⟩

lemma *pc2cl-inj*[*simp*, *intro*!]: *inj pc2cl*
⟨*proof*⟩

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

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

lemma *pc2cl-surj*: ⟦*πl*∈*pclvalid*; !!*π*. *πl*=*pc2cl π* ⟹ *P*⟧ ⟹ *P*
⟨*proof*⟩

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 ≠ *c1*@*CTRL p*#*c2*
⟨*proof*⟩

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

lemma *map-stack-ne-cCc1* [*simp*]:
map STACK w ≠ *c*@*CTRL s*#*c′*
⟨*proof*⟩
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′* ⟷
w′=*w* ∧ *p′*=*p* ∧ *cl′*=*cl*
⟨*proof*⟩

lemma *map-stack-ne-pc2cl*[*simp*]:
map STACK w ≠ *c*@*pc2cl π*@*c′*
c@*pc2cl π*@*c′* ≠ *map STACK w*
⟨*proof*⟩

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′*
⟨*proof*⟩

lemma *c2cl-simps*[*simp*]:
c2cl [] = []
c2cl (*π*#*c*) = *pc2cl π* @ *c2cl c*
c2cl (*c1*@*c2*) = *c2cl c1* @ *c2cl c2*
⟨*proof*⟩

lemma *c2cl-empty*[*simp*]:
c2cl c = [] ⟷ *c*=[]
[] = *c2cl c* ⟷ *c*=[]
⟨*proof*⟩

lemma *c2cl-start-with-ctrl*[*simp*]:
c2cl c ≠ *STACK* γ#*cl*
STACK γ#*cl* ≠ *c2cl c*
⟨*proof*⟩

lemma *c2cl-start-with-ctrl-map*:
w≠[] ⟹ *c2cl c* ≠ *map STACK w*
w≠[] ⟹ *map STACK w* ≠ *c2cl c*
⟨*proof*⟩

lemma *map-stack-c2cl-eq-simps*[*simp*]:
map STACK w @ *c2cl c* = *map STACK w′* @ *c2cl c′* ⟷ *w*=*w′* ∧ *c2cl c*=*c2cl c′*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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

lemma *c2cl-inj*[*simp*, *intro*!]: *inj c2cl*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *c2cl-surj*: ⟦*cl*∈*clvalid*; !!*c*. *cl*=*c2cl c* ⟹ *P*⟧ ⟹ *P*

⟨*proof*⟩

6.2.3 Step Relation on List-Configurations

lemma *cltr-pres-valid*: $(cl,l,cl')\in cltr\ \Delta \Longrightarrow cl\in clvalid \longleftrightarrow cl'\in clvalid$
⟨*proof*⟩

lemma *dpntr-is-cltr*: $[\![(c,l,c')\in dpntr\ \Delta]\!] \Longrightarrow (c2cl\ c,l,c2cl\ c')\in cltr\ \Delta$
⟨*proof*⟩

lemma *cltr-is-dpntr*: $[\![\ (c2cl\ c,l,c2cl\ c')\in cltr\ \Delta\ ]\!] \Longrightarrow (c,l,c')\in dpntr\ \Delta$
⟨*proof*⟩

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

theorem *cltr-eq-dpntr*: $(c2cl\ c,l,c2cl\ c')\in cltr\ \Delta \longleftrightarrow (c,l,c')\in dpntr\ \Delta$
⟨*proof*⟩

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

lemma *cltr2dpntr-fwd*:
$[\![\ (c2cl\ c,l,cl')\in cltr\ \Delta;$
$\quad !!c'.\ [\![cl'=c2cl\ c';\ (c,l,c')\in dpntr\ \Delta]\!] \Longrightarrow P$
$\ ]\!] \Longrightarrow P$
⟨*proof*⟩

lemma *cltr2dpntr-bwd*:
$[\![\ (cl,l,c2cl\ c')\in cltr\ \Delta;$
$\quad !!c.\ [\![cl=c2cl\ c;\ (c,l,c')\in dpntr\ \Delta]\!] \Longrightarrow P$
$\ ]\!] \Longrightarrow P$
⟨*proof*⟩

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

lemma *cltr-is-dpntrc*:
$(c2cl\ c,l,c2cl\ c')\in trcl\ (cltr\ \Delta) \Longrightarrow (c,l,c')\in dpntrc\ \Delta$
⟨*proof*⟩

lemma *dpntrc-is-cltr*:
$(c,l,c')\in dpntrc\ \Delta \Longrightarrow (c2cl\ c,l,c2cl\ c')\in trcl\ (cltr\ \Delta)$
⟨*proof*⟩

theorem *cltr-eq-dpntrc*:
$(c2cl\ c,l,c2cl\ c')\in trcl\ (cltr\ \Delta) \longleftrightarrow (c,l,c')\in dpntrc\ \Delta$
⟨*proof*⟩

lemma *cltrc-pres-valid*:
$(cl,w,cl')\in trcl\ (cltr\ \Delta) \Longrightarrow cl\in clvalid \longleftrightarrow cl'\in clvalid$
⟨*proof*⟩

lemma *cltr2dpntrc-fwd*:
⟦ (*c2cl c*,*l*,*cl′*)∈*trcl* (*cltr* Δ);
!!*c′*. ⟦*cl′*=*c2cl c′*; (*c*,*l*,*c′*)∈*dpntrc* Δ⟧ ⟹ *P*
⟧ ⟹ *P*
⟨*proof*⟩

lemma *cltr2dpntrc-bwd*:
⟦ (*cl*,*l*,*c2cl c′*)∈*trcl* (*cltr* Δ);
!!*c*. ⟦*cl*=*c2cl c*; (*c*,*l*,*c′*)∈*dpntrc* Δ⟧ ⟹ *P*
⟧ ⟹ *P*
⟨*proof*⟩

6.3 Predecessor Sets on List-Semantics

We also define predecessor sets for the list-semantics:

definition *precl* (*pre*$_{cl}$) **where**
pre$_{cl}$ Δ *C′* == { *c* . ∃ *l c′*. *c′*∈*C′* ∧ (*c*,*l*,*c′*) ∈ *cltr* Δ }
definition *precl-star* (*pre*$^*_{cl}$) **where**
pre$^*_{cl}$ Δ *C′* == { *c* . ∃ *ll c′*. *c′*∈*C′* ∧ (*c*,*ll*,*c′*) ∈ *trcl* (*cltr* Δ) }

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

lemma *precl-is-pre*: *pre*$_{cl}$ Δ (*c2cl*‘*C*) = *c2cl*‘(*pre* Δ *C*)
⟨*proof*⟩

lemma *precl-star-is-pre-star*: *pre*$^*_{cl}$ Δ (*c2cl*‘*C*) = *c2cl*‘(*pre** Δ *C*)
⟨*proof*⟩

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 $('S,'P,'\Gamma,'L)$ *ha-rule* =
  HAR-LEAF $'S$ $'P$ $'\Gamma$ *list set* |
  HAR-NOSPAWN $'S$ $'L$ $'S$ |
  HAR-SPAWN $'S$ $'L$ $'S$ $'S$

types $('S,'P,'\Gamma,'L)$ *ha* = $'S$ *list set* × $('S,'P,'\Gamma,'L)$ *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 $W \in H$; $w \in W$ ⟧ ⟹ *lab* H (*NLEAF* (p,w)) s |
  lab-nospawn:
    ⟦ *HAR-NOSPAWN* s l $s' \in H$; *lab* H t s' ⟧ ⟹ *lab* H (*NNOSPAWN* l t) s |
  lab-spawn:
    ⟦ *HAR-SPAWN* s l *ss* $s' \in H$; *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 $(t\#h)$ $(s\#\sigma)$

lemma *labh-empty*[*simp*]:
  labh H [] σ ⟷ σ=[]
  labh H h [] ⟷ h=[]
  ⟨*proof*⟩

lemma *labh-length*: *labh* H h σ ⟹ *length* h = *length* σ
  ⟨*proof*⟩

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* $\Rightarrow$ $('S,'P,'\Gamma,'L)$ *ha-rule set* $\Rightarrow$ $('P\times'S,'\Gamma,'L)$ *dpn*
  for Δ H **where**
  xdpn-nospawn:
    $[\![(p,\gamma \hookrightarrow_l p',w)\in\Delta;$ *HAR-NOSPAWN s l s'*$\in H]\!] \Longrightarrow$
      $((p,s),\gamma \hookrightarrow_l (p',s'),w) \in$ *xdpn* Δ H |
  xdpn-spawn:
    $[\![(p,\gamma \hookrightarrow_l ps,ws \sharp p',w)\in\Delta;$ *HAR-SPAWN s l ss s'*$\in H]\!] \Longrightarrow$
      $((p,s),\gamma \hookrightarrow_l (ps,ss),ws \sharp (p',s'),w)\in$*xdpn* Δ 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*×'*S*,'T) *conf* ⇒ ('*P*,'T) *conf* **where**
proj-c1 cx == *map* (λ((*p*,*s*),*w*). (*p*,*w*)) *cx*
abbreviation
proj-c2 :: ('*P*×'*S*,'T) *conf* ⇒ '*S list* **where**
proj-c2 cx == *map* (λ((*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*×'*S*,'T,'*L*) *ex-tree* ⇒ ('*P*,'T,'*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*,'T) *conf set* ⇒ ('*P*,'T) *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*
⟨*proof*⟩

lemma *proj-cxs2*[*simp*]: *length c = length σ ⟹ proj-c2 (cxs c σ) = σ*
⟨*proof*⟩

lemma *cxs-proj*[*simp*]: *cxs (proj-c1 cx) (proj-c2 cx) = cx*
⟨*proof*⟩

lemma *xdpnc-proj*: *cx ∈ xdpnC C H ⟹ proj-c1 cx ∈ C*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

lemma *xdpn-correct2*:
⟦ *hsem Δ c h c′; labh H h σ* ⟧ ⟹
∃ *hx cx′. hsem (xdpn Δ H) (cxs c σ) hx cx′ ∧*

$cx' \in xdpnCL\ H\ \wedge$
$(map\ proj\text{-}t1\ hx) = h\ \wedge$
$proj\text{-}c1\ cx' = c'$
⟨*proof*⟩

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*:
$prehc\ \Delta\ (langh\ (H0,H))\ C' = projH\ H0\ (\ pre^*\ (xdpn\ \Delta\ H)\ (xdpnC\ C'\ H)\ )$
⟨*proof*⟩

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**
$fp\text{-}cl1\ (CTRL\ (p,s)) = CTRL\ p\ |$
$fp\text{-}cl1\ (STACK\ \gamma) = STACK\ \gamma$

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 $\{[STACK\ \gamma]\ |\gamma.\ True\}$ 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*
⟨*proof*⟩

lemma *fp-cl1-stack-id*[*simp*]: *fp-cl1 s* = *STACK* γ ⟷ *s*=*STACK* γ
⟨*proof*⟩

lemma *fp-cl1-eq-map-stack*[*simp*]:
map fp-cl1 la = *map STACK w* ⟷ *la*=*map STACK w*
⟨*proof*⟩

lemma *star-STACK*[*simplified*,*simp*]:
star {[*STACK* γ] | γ. *True*} = {*map STACK w* | *w*. *True*}
⟨*proof*⟩

lemma *proj-c1-effective*: *c2cl* (*proj-c1 c*) = *map fp-cl1* (*c2cl c*)
⟨*proof*⟩

lemma *fp-inv-subst2I*[*intro*!, *simp*]:
CTRL (*p*,*s*)#*map STACK w* ∈ *fp-inv-subst2 s*
⟨*proof*⟩

lemma *fp-inv-subst2E*:
⟦*cl*∈*fp-inv-subst2 s*; !!*p w*. *cl*=*CTRL* (*p*,*s*)#*map STACK w* ⟹ *P*⟧ ⟹ *P*
⟨*proof*⟩

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*)
⟨*proof*⟩

Correctness of the list-level operation:

theorem *projH-effective*: *c2cl* ‘ *projH H0 Cx* = *projH-cl H0* (*c2cl* ‘ *Cx*)
⟨*proof*⟩

lemma *c2cl-empty-rev*: [] = *c2cl* [] ⟨*proof*⟩

theorem *xdpnCL-effective*: *c2cl* ‘ (*xdpnCL H*) = *xdpnCL-cl H*
⟨*proof*⟩

lemma *inv-proj-c1-effective*:
c2cl ‘ { *cx* . *proj-c1 cx* ∈ *C* } = *lang-inv-proj fp-cl1* (*c2cl* ‘ *C*)
⟨*proof*⟩

theorem *xdpnC-effective*: *c2cl* ‘ (*xdpnC C H*) = *xdpnC-cl* (*c2cl* ‘ *C*) *H*
⟨*proof*⟩

theorem *prehc-effective*:
c2cl ‘ prehc Δ (*langh* (*H0*,*H*)) *C′* = *prehc-cl* Δ (*H0*,*H*) (*c2cl ‘ C′*)
⟨*proof*⟩

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*,*′*Γ,*′L*,*′X*) *ldpn* = (*′P*,*′*Γ,(*′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*,*′*Γ,*′L*,*′X*::*finite*) *ldpn*
assumes
spawn-no-locks: ⟦(*p*,γ ↪$_a$ *ps*,*ws* ♯ *p′*,*w*) ∈ Δ; !!*l*. *a*=*LNone l* ⟹ *P*⟧ ⟹ *P*

begin
lemma *snl-simps*[*simp*, *intro*!]:
$(p,\gamma \hookrightarrow_{LAcq\ x}\ ps,ws \sharp\ p',w) \notin \Delta$
$(p,\gamma \hookrightarrow_{LRel\ x}\ ps,ws \sharp\ p',w) \notin \Delta$
⟨*proof*⟩

lemma *X-finite*: *finite* (*UNIV*::'*X set*) ⟨*proof*⟩
end

9.2 Interleaving Semantics

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

inductive *lock-valid* :: '*X set* ⇒ ('*L*,'*X*) *lockstep* ⇒ '*X set* ⇒ *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'* ⟷ *X*=*X'*
lock-valid X (*LAcq x*) *X'* ⟷ *X'*=*insert x X* ∧ *x*∉*X*
lock-valid X (*LRel x*) *X'* ⟷ *X*=*insert x X'* ∧ *x*∉*X'*
⟨*proof*⟩

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
⟨*proof*⟩
lemma *ldpntrc-subset*: ((*c*,*X*),*w*,(*c'*,*X'*))∈*ldpntrc* Δ ⟹ (*c*,*w*,*c'*)∈*dpntrc* Δ
⟨*proof*⟩

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 $('Q,'\mathrm{T},'L,'X)$ *lex-tree* $= ('Q,\mathrm{T},('L,'X)$ *lockstep*$)$ *ex-tree*
types $('Q,'\mathrm{T},'L,'X)$ *lex-hedge* $= ('Q,\mathrm{T},('L,'X)$ *lockstep*$)$ *ex-hedge*

— The definition of the lock-sensitive scheduler is straightforward:
inductive *lsched*
:: $('Q,\mathrm{T},'L,'X)$ *lex-hedge* $\Rightarrow$ $'X$ *set* $\Rightarrow$ $('L,'X)$ *lockstep list* $\Rightarrow$ *bool*
where
lsched-final: *final* $h \Longrightarrow$ *lsched* $h\ X\ [] \mid$
lsched-nospawn:
$[\![$*lsched* $(h1@t\#h2)\ X'\ w$; *lock-valid* $X\ l\ X']\!] \Longrightarrow$
lsched $(h1@(NNOSPAWN\ l\ t)\#h2)\ X\ (l\#w) \mid$
lsched-spawn:
$[\![$*lsched* $(h1@ts\#t\#h2)\ X'\ w$; *lock-valid* $X\ l\ X']\!] \Longrightarrow$
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* $h\ ll$
⟨*proof*⟩

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*⟩
lemma *lsched-correct2*:
$[\![\ lsched\ h\ X\ ll;\ hsem\ \Delta\ c\ h\ c'\ ]\!] \Longrightarrow \exists X'.\ ((c,X),ll,(c',X')) \in ldpntrc\ \Delta$
⟨*proof*⟩

theorem *lsched-correct*:
$(\exists X'.\ ((c,X),ll,(c',X')) \in ldpntrc\ \Delta) \longleftrightarrow (\exists h.\ hsem\ \Delta\ c\ h\ c' \wedge lsched\ h\ X\ ll)$
⟨*proof*⟩

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*
:: $('P,'T,'L,'X)$ *ldpn* $\Rightarrow$ $('P,'T)$ *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*,γ#*r*) *True l* |
ppairs-genenv: ⟦(*q*,γ ↪$_a$ *qs*,*ws* ♯ *q*′,*w*) ∈ Δ; *ppairs* Δ (*qs*,*ws*) *x l*⟧ ⟹
ppairs Δ (*q*,γ#*r*) *True l*

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

fun *wn-p* :: (′*L*,′*X*) *lockstep list* ⇒ ′*X list* ⇒ *bool* **where**
wn-p [] μ = *distinct* μ |
wn-p (*LAcq x*#*l*) μ ⟷ *wn-p l* (*x*#μ) |
wn-p (*LRel x*#*l*) μ ⟷ (∃ μ′. μ=*x*#μ′ ∧ *x*∉*set* μ′ ∧ *wn-p l* μ′) |
wn-p (-#*l*) μ ⟷ *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*) ⇒
∀ *l*. (*ppairs* Δ (*p*,*w*) *False l* ⟶ *wn-p l* μ) ∧
(*ppairs* Δ (*p*,*w*) *True l* ⟶ *wn-p l* [])

Introduction and elimination rules for *wn-π*

lemma *wn-πI*:
⟦
!!*l*. *ppairs* Δ (*q*,*w*) *False l* ⟹ *wn-p l* μ;
!!*l*. *ppairs* Δ (*q*,*w*) *True l* ⟹ *wn-p l* []
⟧ ⟹ *wn-π* Δ (*q*,*w*) μ
⟨*proof*⟩

lemma *wn-πE*:
⟦*wn-π* Δ (*q*,*w*) μ;
⟦
!!*l*. *ppairs* Δ (*q*,*w*) *False l* ⟹ *wn-p l* μ;
!!*l*. *ppairs* Δ (*q*,*w*) *True l* ⟹ *wn-p l* []
⟧ ⟹ *P*
⟧ ⟹ *P*
⟨*proof*⟩

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* μ ⟹ *distinct* μ
⟨*proof*⟩

lemma *wn-π-distinct*: *wn-π* Δ π μ ⟹ *distinct* μ
⟨*proof*⟩

Well-nestedness is preserved by steps:

lemma *wn-π-none*:
⟦ (*q*,γ ↪$_{(LNone\ l)}$ *q*′,*w*)∈Δ; *wn-π* Δ (*q*,γ#*r*) μ ⟧ ⟹ *wn-π* Δ (*q*′,*w*@*r*) μ
⟨*proof*⟩
lemma (**in** *LDPN*) *wn-π-spawn1*:
⟦ (*q*,γ ↪$_a$ *qs*,*ws* ♯ *q*′,*w*)∈Δ; *wn-π* Δ (*q*,γ#*r*) μ ⟧ ⟹ *wn-π* Δ (*q*′,*w*@*r*) μ

⟨*proof*⟩
lemma *wn-π-spawn2*:
⟦ (*q*,γ ↪$_a$ *qs*,*ws* ♯ *q′*,*w*)∈Δ; *wn-π* Δ (*q*,γ#*r*) μ ⟧ ⟹ *wn-π* Δ (*qs*,*ws*) []
⟨*proof*⟩
lemma *wn-π-acq*:
⟦ (*q*,γ ↪$_{LAcq\ x}$ *q′*,*w*)∈Δ; *wn-π* Δ (*q*,γ#*r*) μ ⟧ ⟹ *wn-π* Δ (*q′*,*w*@*r*) (*x*#μ)
⟨*proof*⟩
lemma *wn-π-rel*:
assumes *A*: (*q*,γ ↪$_{LRel\ x}$ *q′*,*w*)∈Δ *wn-π* Δ (*q*,γ#*r*) μ **and**
C: !!μ′. ⟦μ=*x*#μ′; *x*∉*set* μ′; *wn-π* Δ (*q′*,*w*@*r*) μ′ ⟧ ⟹ *P*
shows *P*
⟨*proof*⟩

lemma (**in** *LDPN*) *wn-π-preserve*:
⟦ (*q*,γ ↪$_l$ *q′*,*w*)∈Δ; *wn-π* Δ (*q*,γ#*r*) *xs*;
!!*xs′*. *wn-π* Δ (*q′*,*w*@*r*) *xs′* ⟹ *P*
⟧ ⟹ *P*

⟦ (*q*,γ ↪$_l$ *qs*,*ws* ♯ *q′*,*w*)∈Δ; *wn-π* Δ (*q*,γ#*r*) *xs*;
!!*xs′*. ⟦ *wn-π* Δ (*q′*,*w*@*r*) *xs′*; *wn-π* Δ (*qs*,*ws*) [] ⟧ ⟹ *P*
⟧ ⟹ *P*
⟨*proof*⟩

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 [] [] ⟷ *True* |
wn-c Δ (π#*c*) (*xs*#μ) ⟷
wn-c Δ *c* μ ∧ *set xs* ∩ *locks-μ* μ = {} ∧ *wn-π* Δ π *xs* |
wn-c Δ - - ⟷ *False*

10.2.1 Auxilliary Lemmas about *wn-c*

lemma *wn-c-simps*[*simp*]:
wn-c Δ *c* [] ⟷ *c*=[]
wn-c Δ [] μ ⟷ μ=[]
⟨*proof*⟩

lemma *wn-c-length*: *wn-c* Δ *c* μ ⟹ *length c* = *length* μ
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *wn-c-append-c-helper*:
assumes
A: *wn-c* Δ *c* μ *c1*@*c2*=*c* **and**
C: !!μ*1* μ*2*. ⟦ μ=μ*1*@μ*2* ∧ *wn-c* Δ *c1* μ*1* ∧ *wn-c* Δ *c2* μ*2* ∧
    locks-μ μ*1* ∩ *locks-μ* μ*2* = {}
  ⟧ ⟹ *P*
shows *P*
⟨*proof*⟩

lemma *wn-c-append-c*:
⟦*wn-c* Δ (*c1*@*c2*) μ;
  !!μ*1* μ*2*. ⟦ μ=μ*1*@μ*2* ∧ *wn-c* Δ *c1* μ*1* ∧ *wn-c* Δ *c2* μ*2* ∧
    locks-μ μ*1* ∩ *locks-μ* μ*2* = {}⟧ ⟹ *P*
⟧ ⟹ *P*
⟨*proof*⟩

lemma *wn-c-append-μ-helper*:
assumes
A: *wn-c* Δ *c* μ μ*1*@μ*2*=μ **and**
C: !!*c1* *c2*. ⟦ *c*=*c1*@*c2* ∧ *wn-c* Δ *c1* μ*1* ∧ *wn-c* Δ *c2* μ*2* ∧
    locks-μ μ*1* ∩ *locks-μ* μ*2* = {}⟧ ⟹ *P*
shows *P*
⟨*proof*⟩

lemma *wn-c-append-μ*:
⟦*wn-c* Δ *c* (μ*1*@μ*2*);
  !!*c1* *c2*. ⟦ *c*=*c1*@*c2* ∧ *wn-c* Δ *c1* μ*1* ∧ *wn-c* Δ *c2* μ*2* ∧
    locks-μ μ*1* ∩ *locks-μ* μ*2* = {}⟧ ⟹ *P*
⟧ ⟹ *P*
⟨*proof*⟩

lemma *wn-c-appendI*:
⟦*wn-c* Δ *c1* μ*1*; *wn-c* Δ *c2* μ*2*; *locks-μ* μ*1* ∩ *locks-μ* μ*2* = {}⟧ ⟹
  wn-c Δ (*c1*@*c2*) (μ*1*@μ*2*)

⟨*proof*⟩

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)
⟨*proof*⟩

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

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*
⟨*proof*⟩

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*⟩

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*⟩

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*,'T,'*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*,'T,'*L*,'*X*) *lex-tree* ⇒ '*X list* ⇒ *bool* **where**

distinct $\mu \Longrightarrow$ *wn-t* (*NLEAF* π) μ |
wn-t t $\mu \Longrightarrow$ *wn-t* (*NNOSPAWN* (*LNone* l) t) μ |
$[\![$*wn-t* t μ; *wn-t ts* [] $]\!] \Longrightarrow$ *wn-t* (*NSPAWN* (*LNone* l) *ts* t) μ |
$[\![$*wn-t* t ($x\#\mu$); $x\notin$*set* $\mu]\!] \Longrightarrow$ *wn-t* (*NNOSPAWN* (*LAcq* x) t) μ |
$[\![$*wn-t* t μ; $x\notin$*set* $\mu]\!] \Longrightarrow$ *wn-t* (*NNOSPAWN* (*LRel* x) t) ($x\#\mu$)

inductive *lock-valid-xs* **where**
distinct xs $\Longrightarrow$ *lock-valid-xs* (*LNone* l) *xs xs* |
$[\![$*distinct xs*; $x\notin$*set xs*$]\!] \Longrightarrow$ *lock-valid-xs* (*LRel* x) ($x\#$*xs*) *xs* |
$[\![$*distinct xs*; $x\notin$*set xs*$]\!] \Longrightarrow$ *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
⟨*proof*⟩

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

lemma *wnt-distinct*: *wn-t* t $\mu \Longrightarrow$ *distinct* μ
⟨*proof*⟩
lemma *wnt-distinct'*: *wn-t'* t *ms* $\Longrightarrow$ *distinct ms*
⟨*proof*⟩

lemma *all-t-wnt-distinct*: $\forall\, t\ c'$. *tsem* Δ (q,w) t $c' \longrightarrow$ *wn-t* t $\mu \Longrightarrow$ *distinct* μ
⟨*proof*⟩

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. μ=$s_1 \ldots s_n$ 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 ($xs\#\mu$) ⟷ *cons-μ* $\mu \wedge$ *distinct xs* $\wedge$ *set xs* $\cap$ *locks-μ* $\mu = \{\}$

A hedge h=$t_1 \ldots t_n$ is well-nested w.r.t. a list μ=$s_1 \ldots s_n$ 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 ($t\#h$) ($xs\#\mu$) ⟷ *wn-h h* $\mu \wedge$ *set xs* $\cap$ *locks-μ* $\mu = \{\} \wedge$ *wn-t'* t *xs* |
wn-h - - ⟷ *False*

lemma *cons-μ-append*[*simp*]:
cons-μ ($\mu 1 @ \mu 2$) ⟷ *cons-μ* $\mu 1 \wedge$ *cons-μ* $\mu 2 \wedge$ *locks-μ* $\mu 1 \cap$ *locks-μ* $\mu 2 = \{\}$
⟨*proof*⟩

10.4.1 Auxilliary Lemmas about *wn-h*

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

wn-h h [] ⟷ *h*=[]
wn-h [] *μ* ⟷ *μ*=[]
⟨*proof*⟩

lemma *wn-h-length*: *wn-h h μ* ⟹ *length h* = *length μ*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *wn-h-append-h-helper*:
assumes
A: *wn-h h μ* *h1*@*h2*=*h* **and**
C: !!*μ1 μ2*. ⟦ *μ*=*μ1*@*μ2* ∧ *wn-h h1 μ1* ∧ *wn-h h2 μ2* ∧
locks-μ μ1 ∩ *locks-μ μ2* = {}⟧ ⟹ *P*
shows *P*
⟨*proof*⟩

lemma *wn-h-append-h*:
⟦*wn-h* (*h1*@*h2*) *μ*;
!!*μ1 μ2*. ⟦ *μ*=*μ1*@*μ2* ∧ *wn-h h1 μ1* ∧ *wn-h h2 μ2* ∧
locks-μ μ1 ∩ *locks-μ μ2* = {}⟧ ⟹ *P*
⟧ ⟹ *P*
⟨*proof*⟩

lemma *wn-h-append-μ-helper*:
assumes
A: *wn-h h μ* *μ1*@*μ2*=*μ* **and**
C: !!*h1 h2*. ⟦ *h*=*h1*@*h2* ∧ *wn-h h1 μ1* ∧ *wn-h h2 μ2* ∧
locks-μ μ1 ∩ *locks-μ μ2* = {}⟧ ⟹ *P*
shows *P*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *wn-h-appendI*:
⟦*wn-h h1 μ1*; *wn-h h2 μ2*; *locks-μ μ1 ∩ locks-μ μ2 = {}*⟧ ⟹
wn-h (*h1@h2*) (*μ1@μ2*)
⟨*proof*⟩

lemma *wn-h-prependI*:
⟦*wn-t′ t xs*; *wn-h h μ*; *set xs ∩ locks-μ μ = {}*⟧ ⟹ *wn-h* (*t#h*) (*xs#μ*)
⟨*proof*⟩

lemma *wn-h-singlehE*: ⟦*wn-h* [*t*] *μ*; !!*xs*. ⟦*μ=*[*xs*]; *wn-t′ t xs*⟧ ⟹ *P* ⟧ ⟹ *P*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩
lemma *wn-c-hI*[*intro?*]: ⟦!!*h c′*. *hsem Δ c h c′* ⟹ *wn-h h μ* ⟧ ⟹ *wn-c-h Δ c μ*
⟨*proof*⟩

lemma *wn-π-t-distinct*: *wn-π-t Δ π μ* ⟹ *distinct μ*
⟨*proof*⟩

lemma *wn-c-h-prepend1*: **assumes** *A*: *wn-c-h* Δ $(\pi\#c)$ $(xs\#\mu)$
shows *wn-π-t* Δ π *xs* *wn-c-h* Δ *c* μ *set xs* $\cap$ *locks-μ* $\mu = \{\}$
⟨*proof*⟩

lemma *wn-c-h-prepend2*:
$[\![$*wn-π-t* Δ π *xs*; *wn-c-h* Δ *c* μ; *set xs* $\cap$ *locks-μ* $\mu = \{\}]\!] \Longrightarrow$
wn-c-h Δ $(\pi\#c)$ $(xs\#\mu)$
⟨*proof*⟩

lemma *wn-c-h-prepend*[*simp*]:
wn-c-h Δ $(\pi\#c)$ $(xs\#\mu) \longleftrightarrow$
wn-π-t Δ π *xs* $\wedge$ *wn-c-h* Δ *c* μ $\wedge$ *set xs* $\cap$ *locks-μ* $\mu = \{\}$
⟨*proof*⟩

lemma *wn-c-h-empty*[*simp*]: *wn-c-h* Δ *c* [] $\longleftrightarrow$ (*c*=[]) ⟨*proof*⟩

lemma *wn-c-h-prepend-c*:
$[\![$*wn-c-h* Δ $(\pi\#c)$ μ;
!!*xs* μ'. $[\![\mu$=*xs*#μ'; *wn-π-t* Δ π *xs*; *wn-c-h* Δ *c* μ';
set xs $\cap$ *locks-μ* $\mu' = \{\}$ $]\!] \Longrightarrow P$
$]\!] \Longrightarrow P$
⟨*proof*⟩

lemma *wn-c-h-simps*[*simp*]: *wn-c-h* Δ [] μ $\longleftrightarrow$ (μ=[])
⟨*proof*⟩

lemma (**in** *LDPN*) *wnπ2wnt*: $[\![$*tsem* Δ (q,w) *t c'*; *wn-π* Δ (q,w) $\mu]\!] \Longrightarrow$ *wn-t t* μ
⟨*proof*⟩

lemma (**in** *LDPN*) *wnt2wnp*:
$[\![$*ppairs* Δ (q,w) *en l*; $\forall$ *t c'*. *tsem* Δ (q,w) *t c'* $\longrightarrow$ *wn-t t* $\mu]\!] \Longrightarrow$
($\neg$*en* $\longrightarrow$ *wn-p l* μ) $\wedge$ (*en* $\longrightarrow$ *wn-p l* [])
⟨*proof*⟩

theorem (**in** *LDPN*) *wnπ-eq-wnπt*: *wn-π* Δ π μ $\longleftrightarrow$ *wn-π-t* Δ π μ ⟨*proof*⟩

theorem (**in** *LDPN*) *wnc-eq-wnch*: *wn-c* Δ *c* μ $\longleftrightarrow$ *wn-c-h* Δ *c* μ
⟨*proof*⟩

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*$]\!] \Longrightarrow$
$\exists$ *xs'*. *X'*=*set xs'* $\wedge$ *lock-valid-xs l xs xs'* $\wedge$ *wn-t' t xs'*
⟨*proof*⟩

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′*
⟨*proof*⟩

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*⟩

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′*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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* ⟨*proof*⟩

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*
)
)
⟨*proof*⟩

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*
⟨*proof*⟩

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

lemma *as-comp-assoc*: (*x*‖*y*)‖*z* = *x*‖*y*‖*z*
⟨*proof*⟩

interpretation *as-comp-acz*: *ACIZ*[*op* ‖ *Some* (*empty*,{},*empty*) *None*]
⟨*proof*⟩

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*
⟨*proof*⟩

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 == λ*y*. *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,'T,'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,'T,'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
  ⟨proof⟩

lemma l-add-use-empty[simp]: l-add-use empty X = empty
  ⟨proof⟩

lemma l-add-use-eq-empty[simp]: l-add-use f X = empty ⟷ f=empty
  ⟨proof⟩

lemma l-add-use-add[simp]:
  l-add-use (l++l') X = l-add-use l X ++ l-add-use l' X
  ⟨proof⟩

lemma l-add-use-le: l ≤ l-add-use l X
  ⟨proof⟩
```

lemma *l-remove-add*[*simp*]: *l-remove* (*l1*++*l2*) *m* = *l-remove l1 m* ++ *l-remove l2 m*
⟨*proof*⟩

lemma *l-remove-no-eff*[*simp*]: *x*∉*dom l* ⟹ *l-remove l x* = *l*
⟨*proof*⟩

lemma *l-remove-dom*[*simp*]: *dom* (*l-remove l x*) = *dom l* − {*x*}
⟨*proof*⟩

lemma *l-remove-app*[*simp*]:
l-remove l x x = *None*
x≠*x'* ⟹ *l-remove l x x'* = *l x'*
⟨*proof*⟩

lemma *l-remove-eq-empty*: *l-remove l x* = *empty* ⟹ *dom l* ⊆ {*x*}
⟨*proof*⟩

lemma *l-remove-le-l* [*simp*]: *l-remove l x* ≤ *l*
⟨*proof*⟩

lemma *as-ran-e-le-u*: *as t* = *Some* (*l*,*u*,*e*) ⟹ ⋃*ran e* ⊆ *u*
⟨*proof*⟩

lemma *ash-le-u*: *ash h* = *Some* (*l*,*u*,*e*) ⟹ ⋃*ran e* ⊆ *u*
⟨*proof*⟩

lemma *ash-final*[*simp*]: *final h* ⟹ *ash h*=*Some* (*empty*,{},*empty*)
⟨*proof*⟩

lemma *ash-append*[*simp*]: *ash* (*h1*@*h2*) = *ash h1* ‖ *ash h2*
⟨*proof*⟩

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*)
⟨*proof*⟩

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 ¬∃ 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 $x \in dom\ e \wedge x' \in the\ (e\ x)$

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 $x' \in dom\ l \wedge x \in the\ (l\ x')$

definition *agraph* :: $'X\ ah \Rightarrow ('X \times 'X)\ set$ **where**
agraph e == { (x,x') . $x \in dom\ e \wedge x' \in the\ (e\ x)$ }
definition *rgraph* :: $'X\ ah \Rightarrow ('X \times 'X)\ set$ **where**
rgraph l == { (x,x') . $x' \in dom\ l \wedge x \in the\ (l\ x')$ }

lemma *agraph-alt*: *agraph e* = { (x,x') . $\exists X'.\ e\ x = Some\ X' \wedge x' \in X'$}
⟨*proof*⟩
lemma *rgraph-alt*: *rgraph l* = { (x,x') . $\exists X.\ l\ x' = Some\ X \wedge x \in X$ }
⟨*proof*⟩

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* = $(rgraph\ h)^{-1}$
⟨*proof*⟩

lemma *agraph-add-union*:
$[\![dom\ e \cap dom\ e' = \{\}]\!] \Longrightarrow agraph\ (e{+}{+}e') = agraph\ e \cup agraph\ e'$
⟨*proof*⟩

lemma *rgraph-add-union*:
$[\![dom\ l \cap dom\ l' = \{\}]\!] \Longrightarrow rgraph\ (l{+}{+}l') = rgraph\ l \cup rgraph\ l'$
⟨*proof*⟩

lemma *agraph-domain-simp*[*simp*]:
Domain (*agraph h*) = *dom h* − { x . $h\ x = Some\ \{\}$ }
⟨*proof*⟩

lemma *agraph-range-simp*[*simp*]: *Range* (*agraph h*) = $\bigcup ran\ h$
⟨*proof*⟩

lemma *rgraph-domain-simp*[*simp*]: *Domain* (*rgraph h*) = $\bigcup ran\ h$
⟨*proof*⟩

lemma *rgraph-range-simp*[*simp*]:

Range (*rgraph h*) = *dom h* − { *x* . *h x* = *Some* {} }
⟨*proof*⟩

lemma *graph-empty*[*simp*]:
agraph empty = {}
rgraph empty = {}
⟨*proof*⟩

lemma *rgraph-add-use*: *rgraph* (*l-add-use l X*) = *rgraph l* ∪ *X*×*dom l*
⟨*proof*⟩

lemma *rgraph-remove*: *rgraph* (*l-remove l x*) = *rgraph l* − *UNIV*×{*x*}
⟨*proof*⟩

lemma *rgraph-upd*: *x*∉*dom l* ⟹ *rgraph* (*l*(*x*↦*X*)) = *rgraph l* ∪ *X*×{*x*}
⟨*proof*⟩

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

lemma *agraph-upd*: *x*∉*dom e* ⟹ *agraph* (*e*(*x*↦*X*)) = *agraph e* ∪ {*x*}×*X*
⟨*proof*⟩

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

lemma *rgraph-mono*: *l*≤*l*′ ⟹ *rgraph l* ⊆ *rgraph l*′
⟨*proof*⟩

lemma *agraph-mono*: *e*≤*e*′ ⟹ *agraph e* ⊆ *agraph e*′
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *cons-h-empty*[*simp*]: *cons-h empty*
⟨*proof*⟩

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*′

⟨*proof*⟩

lemma *cons-h-antimono*: ⟦*l*≤*l'*; *cons-h l'*⟧ ⟹ *cons-h l*
⟨*proof*⟩

lemma *cons-h-update*:
assumes *A*: *cons-h h* *X*∩*insert x* (*dom h*) = {}
shows *cons-h* (*h*(*x*↦*X*))
⟨*proof*⟩

lemma *cons-h-update2*:
assumes *A*: *cons-h h* *x*∉*dom h* *x*∉*X* *x*∉⋃ *ran h*
shows *cons-h* (*h*(*x*↦*X*))
⟨*proof*⟩

lemma *cons-h-remove*: *cons-h l* ⟹ *cons-h* (*l-remove l m*)
⟨*proof*⟩

lemma *cons-h-add-use*: ⟦*m*∉*dom l*; *cons-h l*⟧ ⟹ *cons-h* (*l-add-use l* {*m*})
⟨*proof*⟩

lemma *cons-h-add-remove*: *cons-h l* ⟹ *cons-h* (*l-add-use* (*l-remove l m*) {*m*})
⟨*proof*⟩

lemma *cons-h-add-remove-partial*:
⟦ *m*∉*dom l1*; *cons-h* (*l1*++*l2*) ⟧ ⟹
cons-h (*l1* ++ *l-add-use* (*l-remove l2 m*) {*m*})
⟨*proof*⟩

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

fun *cons-as* :: *'X as* ⇒ *'X set* ⇒ *bool* **where**
cons-as (*l*,*u*,*e*) *ξ* ⟷
u∩(*ξ*−*dom l*) = {} ∧ *dom e* ∩ (*ξ*−*dom l*) = {} ∧ *cons-h l* ∧ *cons-h e*

lemma *cons-as-antimono*: ⟦*cons-as h ξ*; *ξ'*⊆*ξ*⟧ ⟹ *cons-as h ξ'*
⟨*proof*⟩

fun *cons* **where**
cons None X = *False* |
cons (*Some* (*l*,*u*,*e*)) *X* = *cons-as* (*l*,*u*,*e*) *X*

11.3.1 Minimal Elements

lemma *finite-acyclic-wf*: ⟦*finite r*; *acyclic r*⟧ ⟹ *wf r*
⟨*proof*⟩

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* = {})
⟨*proof*⟩

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*⟩

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*≠*empty* *cons-h e*
shows ∃ *m*. *ah-min e m*
⟨*proof*⟩

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-t' t μ*; *as t* = *Some* (*l*, *u*, *e*)⟧ ⟹ *dom l* ⊆ *set μ*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟧ ⟹

$\exists \mu 1\ \mu 2.\ \mu = \mu 1 @ x \# \mu 2 \wedge set\ \mu 1 \subseteq dom\ l \wedge$
$(\forall x' \in set\ \mu 1.\ l\ x' \leq l\ x \wedge$
$(case\ l\ x'\ of\ None \Rightarrow True \mid Some\ lx' \Rightarrow x \notin lx' \wedge x' \notin lx')$
$)$
⟨*proof*⟩

lemma *wn-t-dom-l-stack'*: $[\![wn\text{-}t'\ t\ \mu;\ as\ t = Some\ (l,u,e);\ x \in dom\ l]\!] \Longrightarrow$
$\exists \mu 1\ \mu 2.\ \mu = \mu 1 @ x \# \mu 2 \wedge set\ \mu 1 \subseteq dom\ l \wedge$
$(\forall x' \in set\ \mu 1.\ l\ x' \leq l\ x \wedge x \notin the\ (l\ x') \wedge x' \notin the\ (l\ x'))$
⟨*proof*⟩

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*:
$[\![\ lsched\ h\ X\ w;\ wn\text{-}h\ h\ \mu;\ X = locks\text{-}\mu\ \mu\ ]\!] \Longrightarrow$
$\exists l\ u\ e.\ ash\ h = Some\ (l,u,e) \wedge cons\text{-}as\ (l,u,e)\ (locks\text{-}\mu\ \mu)$
— 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*⟩
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 $[] = (0::nat) \mid$
list-size' f $(a\#l) = f\ a +$ *list-size' f l*

fun *size-t* **where**
size-t $(NLEAF\ \pi) = Suc\ 0 \mid$

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*
⟨*proof*⟩

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*=[]
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *rootlocks-split*:
⟦ *x*∈*rootlocks h*; !!*h1 t h2*. *h*=*h1*@*NNOSPAWN* (*LAcq x*) *t*#*h2* ⟹ *P* ⟧ ⟹ *P*

⟨*proof*⟩

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*) ⟧ ⟹
l x = *Some Xu*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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=⋃*ran e*; *l*=*empty*;
e = e'(*x*↦*u*); *x*∉*dom e'*
⟧ ⟹ *P*
shows *P*
⟨*proof*⟩

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*⟩

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 $\Longrightarrow$ *P*
!!*h1 lab ts t h2*. *h*=*h1*@*NSPAWN lab ts t*#*h2* $\Longrightarrow$ *P*
!!*h1 t nlab h2*. *h*=*h1*@*NNOSPAWN* (*LNone nlab*) *t*#*h2* $\Longrightarrow$ *P*
!!*h1 x t h2*. *h*=*h1*@*NNOSPAWN* (*LRel x*) *t*#*h2* $\Longrightarrow$ *P*
⟦ !!*Q t*. ⟦ *t*∈*set h*; !!*x t'*. *t*=*NNOSPAWN* (*LAcq x*) *t'* $\Longrightarrow$ *Q*;
!!*p w*. *t*=*NLEAF* (*p*,*w*) $\Longrightarrow$ *Q*
⟧ $\Longrightarrow$ *Q*;
!!*Q*. ⟦ !!*t' x*. *NNOSPAWN* (*LAcq x*) *t'* ∈ *set h* $\Longrightarrow$ *Q* ⟧ $\Longrightarrow$ *Q*
⟧ $\Longrightarrow$ *P*
shows *P*
⟨*proof*⟩

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* = {}
⟧ $\Longrightarrow$ *P*
⟧ $\Longrightarrow$ *P*
⟨*proof*⟩

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* = {}
⟧ $\Longrightarrow$ *P*
⟧ $\Longrightarrow$ *P*
⟨*proof*⟩

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*. ⟦

$l{=}l1{+}{+}l2$; $u{=}u1{\cup}u2$; $e{=}e1{+}{+}e2$; *as* t = *Some* $(l1,u1,e1)$;
ash h1 $\parallel$ *ash h2* = *Some* $(l2,u2,e2)$;
dom l1 $\cap$ *dom l2* = {}; *dom e1* $\cap$ *dom e2* = {}
$\rrbracket \Longrightarrow P$
shows P
⟨*proof*⟩

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: $!!\mu 1\ xs\ \mu 2\ l1\ u1\ e1\ l2\ u2\ e2.\ [\![$
$\mu{=}\mu 1@xs\#\mu 2$; $l{=}l1{+}{+}l2$; $u{=}u1{\cup}u2$; $e{=}e1{+}{+}e2$;
wn-t′ t xs; *wn-h h1* $\mu 1$; *wn-h h2* $\mu 2$;
as t = *Some* $(l1,u1,e1)$; *ash h1* $\parallel$ *ash h2* = *Some* $(l2,u2,e2)$;
locks-μ $\mu 1$ $\cap$ *set xs* = {}; *locks-*μ $\mu 1$ $\cap$ *locks-*μ $\mu 2$ = {};
set xs $\cap$ *locks-*μ $\mu 2$ = {}; *dom l1* $\cap$ *dom l2* = {}; *dom e1* $\cap$ *dom e2* = {}
$]\!] \Longrightarrow P$
shows P
⟨*proof*⟩

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 $h{::}('P,'\mathrm{T},'L,'X)$ *lex-hedge*
assumes *A*: *ash h*=*Some* (l,u,e) *cons-as* (l,u,e) (*locks-*μ μ) *wn-h h* μ
shows $\exists w.$ *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*⟩

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*)
)
⟨*proof*⟩

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*)
⟨*proof*⟩

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 :: $('P,'\mathrm{T},'L,'X::finite)$ *ldpn*
  fixes *c0* :: $('P,'\mathrm{T})$ *conf* — Initial configuration
  fixes $\mu 0$ :: *'X list list* — Locks held at the start configuration
  assumes *wellnested*: *wn-c* Δ *c0* $\mu 0$ — Start configuration must be well-nested
begin

12.1.1 Reachable Configurations

  definition *reachable* == $\{$ *c* . $\exists$ *w*. (*c0*,*w*,*c*)$\in$*dpntrc* Δ $\}$
  definition *reachablels* == $\{$ (*c*,*X*) . $\exists$ *w*. ((*c0*,*locks-*μ $\mu 0$),*w*,(*c*,*X*))$\in$*ldpntrc* Δ $\}$

  lemma *reachablels-subset*: (*c*,*X*)$\in$*reachablels* $\Longrightarrow$ *c*$\in$*reachable*
    ⟨*proof*⟩

  lemma *reachable-wn*:
    $[\![$(*c*,*X*)$\in$*reachablels*; !!μ. $[\![$*wn-c* Δ *c* μ; *X*=*locks-*μ $\mu]\!] \Longrightarrow P]\!] \Longrightarrow P$
    ⟨*proof*⟩

  lemma *reachablels-triv*[*simp*]: (*c0*, *locks-*μ $\mu 0$)$\in$*reachablels*
    ⟨*proof*⟩

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-*$\mu\ \mu 0)\in$*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 $u,v \in \Gamma$, 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 $atUV\text{-}ordered\ u\ v == \{\ c.\ \exists\, q\ r\ q'\ r'.\ [(q,u\#r),(q',v\#r')] \leq c\ \}$
definition $atUV\ u\ v == (atUV\text{-}ordered\ u\ v) \cup (atUV\text{-}ordered\ v\ u)$

The following property specification describes all executions reaching a conflict:

definition *conflict-spec u v ==*
$[SPEC\text{-}STEPS\ UNIV,\ SPEC\text{-}RESTRICT\ (atUV\ u\ v)]$

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

definition $has\text{-}conflict\text{-}ref\ u\ v == \exists (c,X) \in reachablels.\ c \in atUV\ u\ v$

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

lemma $has\text{-}conflict\text{-}ref\ u\ v \longleftrightarrow model\text{-}check\text{-}ref\ (conflict\text{-}spec\ u\ v)$
$\langle proof \rangle$

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 $u::'\Gamma$, 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, $u \in \Gamma$, the set $atU\ u$ is the set of all configurations that have a process with u at the top of the stack.

definition $atU\ u == \{\ c\ .\ \exists\, q\ r.\ (q,u\#r) \in set\ c\ \}$

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*)
⟨*proof*⟩

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*,'T,*'L*,*'X*::*finite*) *ldpn*
constrains *c0* :: (*'P*×*'X set*,'T) *conf*
constrains *μ0* :: *'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* $(snd\ p)\ l\ (snd\ p')$
⟨*proof*⟩

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

lemma *encoding-correct-spawn2'*:
$(p,\gamma \hookrightarrow_l ps,ws \sharp p',w) \in \Delta \Longrightarrow snd\ ps = \{\}$
⟨*proof*⟩

lemma *ec-preserve-singlestep*:
assumes
A: $((c,locks\text{-}\mu\ \mu),l,(c',X'))\in ldpntr\ \Delta$ $wn\text{-}c\ \Delta\ c\ \mu$
$map\ (snd\circ fst)\ c = map\ set\ \mu$ **and**
C: $!!\mu'.\ [\![\ wn\text{-}c\ \Delta\ c'\ \mu';\ X'=locks\text{-}\mu\ \mu';$
$map\ (snd\circ fst)\ c' = map\ set\ \mu'$
$]\!] \Longrightarrow P$
shows *P*
⟨*proof*⟩

lemma *ec-preserve*:
assumes
A: $((c,locks\text{-}\mu\ \mu),ll,(c',X'))\in ldpntrc\ \Delta$ $wn\text{-}c\ \Delta\ c\ \mu$
$map\ (snd\circ fst)\ c = map\ set\ \mu$ **and**
C: $!!\mu'.\ [\![X'=locks\text{-}\mu\ \mu';\ wn\text{-}c\ \Delta\ c'\ \mu';\ map\ (snd\circ fst)\ c' = map\ set\ \mu']\!] \Longrightarrow P$
shows *P*
⟨*proof*⟩

The following abbreviates the locks owned by a configuration:

abbreviation *locks-c c* == *list-collect-set* $(snd\circ fst)\ c$

lemma *locks-μ-mapset*: $locks\text{-}\mu\ \mu = \bigcup set\ (map\ set\ \mu)$

⟨*proof*⟩

lemma *locks-c-mapset*: *locks-c c* = ⋃ *set* (*map* (*snd*∘*fst*) *c*)
⟨*proof*⟩

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*×*'X set*,'T,*'L*,*'X*::*finite*) *ldpn*
constrains *c0* :: (*'P*×*'X set*,'T) *conf*
constrains *μ0* :: *'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 μ0*

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *hsem-locks*: *hsem* Δ *c h c'* ⟹ *map* (*snd*∘*fst*) *c* = *rlocks-h h*
⟨*proof*⟩

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*⟩

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

lemma *lsprestar1*:
assumes
REACH:(*c*,*X*)∈*reachablels* **and**
PRE: *c*∈*prehc* Δ *Hls C'*
shows ∃ *c'*∈*C'*. ∃ *ll X'*. ((*c*,*X*),*ll*,(*c'*,*X'*))∈*ldpntrc* Δ
⟨*proof*⟩

lemma *lsprestar2*:
assumes
REACH:(*c*,*X*)∈*reachablels* **and**
MEM: *c'*∈*C'* **and**
PATH: ((*c*,*X*),*ll*,(*c'*,*X'*))∈*ldpntrc* Δ
shows *c*∈*prehc* Δ *Hls C'*
⟨*proof*⟩

theorem *lsprestar*:
assumes *REACH*:(*c*,*X*)∈*reachablels*
shows *c*∈*prehc* Δ *Hls C'* ⟷ (∃ *c'*∈*C'*. ∃ *ll X'*. ((*c*,*X*),*ll*,(*c'*,*X'*))∈*ldpntrc* Δ)
⟨*proof*⟩

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* ⇒ ('*Q*,'T,'*L*,'*X*) *lex-tree* ⇒ *bool* **where**
isLab L (*NLEAF* π) ⟷ *True* |
isLab L (*NNOSPAWN l t*) ⟷ *l*∈*L* ∧ *isLab L t* |
isLab L (*NSPAWN l ts t*) ⟷ *l*∈*L* ∧ *isLab L ts* ∧ *isLab L t*

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

lemma *final-h-is-lab*[*simp*]: *final h* ⟹ *list-all* (*isLab L*) *h*
⟨*proof*⟩

lemma *HLab-correct*: *sched h ll* ⟹ *h*∈*HLab L* ⟷ *ll*∈*lists L*
⟨*proof*⟩

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

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

fun *mc-pre* :: ('*P*×'*X set*,'T,'*L*,'*X*) *spec* ⇒ ('*P*×'*X set*,'T) *conf set* **where**
mc-pre [] = *UNIV* |
mc-pre (*SPEC-RESTRICT C* # Φ) = *C* ∩ *mc-pre* Φ |
mc-pre (*SPEC-STEP L* # Φ) = *prehc* Δ (*Hls* ∩ *Hpre* ∩ *HLab L*) (*mc-pre* Φ) |
mc-pre (*SPEC-STEPS L* # Φ) = *prehc* Δ (*Hls* ∩ *HLab L*) (*mc-pre* Φ)

lemma *mc-pre-correct-aux*:
(*c*,*X*)∈*reachablels* ⟹ *c*∈*mc-pre* Φ ⟷ (*c*,*X*)∈*Domain* (*spec-tr* Φ)
⟨*proof*⟩

theorem *mc-pre-correct*: *c0*∈*mc-pre* Φ ⟷ *model-check-ref* Φ
⟨*proof*⟩

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]
  ⟨proof⟩


fun lstackm :: ('Γ ⇀ 'X) ⇒ 'Γ list ⇒ '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'
  ⟨proof⟩

lemma lstack-spawn-empty[simp]:
  ⟦ (∀γs∈set w. mon γs=None) ⟧ ⟹ lstackm mon w = []
  ⟨proof⟩


locale MDPN = EncodedLDPN +
  constrains
    Δ :: ('P×'X set,'Γ,'L,'X::finite) ldpn
  fixes mon :: 'Γ ⇒ 'X option — Maps stack symbols to associated monitors

  assumes
    locks-lnone-pop-nospawn:
      (p,γ ↪LNone a p',[])∈Δ ⟹ mon γ = None and
    locks-lnone-pop-spawn:
      (p,γ ↪l ps,ws ♯ p',[])∈Δ ⟹ mon γ = None and
    locks-lnone-nospawn:
```

$(p,\gamma \hookrightarrow_{LNone\ a}\ p',w@[\gamma'])\in\Delta \Longrightarrow mon\ \gamma' = mon\ \gamma \wedge$
$(\forall \gamma s\in set\ w.\ mon\ \gamma s = None)$ **and**

locks-lnone-spawn:

$(p,\gamma \hookrightarrow_{l}\ ps,ws\ \sharp\ p',w@[\gamma'])\in\Delta \Longrightarrow mon\ \gamma' = mon\ \gamma \wedge$
$(\forall \gamma s\in set\ w.\ mon\ \gamma s = None)$ **and**

locks-spawn:

$(p,\gamma \hookrightarrow_{l}\ ps,ws\ \sharp\ p',w)\in\Delta \Longrightarrow (\forall \gamma s\in set\ ws.\ mon\ \gamma s = None)$ **and**

locks-acquire:

$[\![\ (p,\gamma \hookrightarrow_{LAcq\ x}\ p',w)\in\Delta;$
$!!w'\ \gamma 2\ \gamma 1.\ [\![\ w = w'@[\gamma 1,\gamma 2];\ mon\ \gamma 2 = mon\ \gamma;\ mon\ \gamma 1 = Some\ x;$
$(\forall \gamma s\in set\ w'.\ mon\ \gamma s = None)$
$]\!] \Longrightarrow P$
$]\!] \Longrightarrow P$ **and**

locks-release:

$(p,\gamma \hookrightarrow_{LRel\ x}\ p',w)\in\Delta \Longrightarrow w = [] \wedge mon\ \gamma = Some\ x$

begin

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

lemma *lstack-lnone-nospawn*:
$[\![(p,\gamma \hookrightarrow_{LNone\ a}\ p',w)\in\Delta]\!] \Longrightarrow lstack\ (\gamma\#r) = lstack\ (w@r)$
⟨*proof*⟩

lemma *lstack-lnone-spawn*:
$[\![(p,\gamma \hookrightarrow_{a}\ ps,ws\ \sharp\ p',w)\in\Delta]\!] \Longrightarrow lstack\ (\gamma\#r) = lstack\ (w@r)$
⟨*proof*⟩

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))
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)

⟨*proof*⟩

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*⟩

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*⟩

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*} ⟨*proof*⟩

instance *t-my-locks*::*finite*
⟨*proof*⟩

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*
⟨*proof*⟩

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 *myΔ* :: (*nat* × *t-my-locks set*,*nat*,*unit*,*t-my-locks*) *ldpn* **where**
myΔ = {
((0,{}),1 ↪$_{LAcq\ l1}$ (0,{l1}),[2,3]),
((0,{l1}),2 ↪$_{LAcq\ l2}$ (0,{l1,l2}),[4,5]),
((0,{l1,l2}),4 ↪$_{LNone\ ()}$ (0,{}),[1]♯(0,{l1,l2}),[6]),
((0,{l1,l2}),6 ↪$_{LNone\ ()}$ (0,{}),[11]♯(0,{l1,l2}),[7]),
((0,{l1,l2}),7 ↪$_{LRel\ l2}$ (0,{l1}),[]),
((0,{l1}),5 ↪$_{LRel\ l1}$ (0,{}),[]),
((0,{}),3 ↪$_{LNone\ ()}$ (0,{}),[]),

((0,{}),11 ↪$_{LNone\ ()}$ (0,{}),[11]♯(0,{}),[12]),
((0,{}),12 ↪$_{LNone\ ()}$ (0,{}),[11,13]),
((0,{}),11 ↪$_{LAcq\ l2}$ (0,{l2}),[14,13]),
((0,{l2}),14 ↪$_{LAcq\ l1}$ (0,{l1,l2}),[16,17]),
((0,{l1,l2}),16 ↪$_{LNone\ ()}$ (0,{}),[1]♯(0,{l1,l2}),[18]),
((0,{l1,l2}),18 ↪$_{LRel\ l1}$ (0,{l2}),[]),
((0,{l2}),17 ↪$_{LRel\ l2}$ (0,{}),[]),
((0,{}),13 ↪$_{LNone\ ()}$ (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*]
⟨*proof*⟩

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*])] [[]]]
⟨*proof*⟩

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.