A Theory of Architectural Design Patterns

Diego Marmsoler

March 17, 2025

Abstract

The following document formalizes and verifies several architectural design patterns [1]. Each pattern specification is formalized in terms of a locale where the locale assumptions correspond to the assumptions which a pattern poses on an architecture. Thus, pattern specifications may build on top of each other by interpreting the corresponding locale. A pattern is verified using the framework provided by the AFP entry *Dynamic Architectures* [3].

Currently, the document consists of formalizations of 4 different patterns: the singleton, the publisher subscriber, the blackboard pattern, and the blockchain pattern. Thereby, the publisher component of the publisher subscriber pattern is modeled as an instance of the singleton pattern and the blackboard pattern is modeled as an instance of the publisher subscriber pattern.

In general, this entry provides the first steps towards an overall theory of architectural design patterns [2].

Contents

1 A Theory of Singletons

In the following, we formalize the specification of the singleton pattern as described in [4].

theory *Singleton*
imports *DynamicArchitectures.Dynamic-Architecture-Calculus*
begin

1.1 Singletons

In the following we formalize a variant of the Singleton pattern.

locale $singleton = dynamic\text{-}component\ cmp\ active$
  for $active :: 'id \Rightarrow cnf \Rightarrow bool$ (‹‖-‖-› [0,110]60)
  and $cmp :: 'id \Rightarrow cnf \Rightarrow 'cmp$ (‹σ-(-)› [0,110]60) +
assumes $alwaysActive$: $\bigwedge k.\ \exists\, id.\ \|id\|_k$
  and $unique$: $\exists\, id.\ \forall\, k.\ \forall\, id'.\ (\|id'\|_k \longrightarrow id = id')$
begin

1.1.1 Calculus Interpretation

$baIA$: $[\![\exists\, i{\geq}n.\ \|c\|_{t\ i};\ \varphi\ (\sigma_c t\ \langle c \rightarrow t\rangle_n)]\!] \Longrightarrow eval\ c\ t\ t'\ n\ [\varphi]_b$

$baIN1$: $[\![\exists\, i.\ \|c\|_{t\ i};\ \neg\ (\exists\, i{\geq}n.\ \|c\|_{t\ i});\ \varphi\ (t'\ (n - \langle c \wedge t\rangle - 1))]\!] \Longrightarrow eval\ c\ t\ t'\ n\ [\varphi]_b$

$baIN2$: $[\![\nexists\, i.\ \|c\|_{t\ i};\ \varphi\ (t'\ n)]\!] \Longrightarrow eval\ c\ t\ t'\ n\ [\varphi]_b$

1.1.2 Architectural Guarantees

definition $the\text{-}singleton \equiv THE\ id.\ \forall\, k.\ \forall\, id'.\ \|id'\|_k \longrightarrow id' = id$

theorem $ts\text{-}prop$:
 fixes $k{::}cnf$
 shows $\bigwedge id.\ \|id\|_k \Longrightarrow id = the\text{-}singleton$
  and $\|the\text{-}singleton\|_k$
⟨*proof*⟩
declare $ts\text{-}prop(2)[simp]$

lemma $lNact\text{-}active[simp]$:
 fixes $cid\ t\ n$
 shows $\langle the\text{-}singleton \Leftarrow t\rangle_n = n$
 ⟨*proof*⟩

lemma $lNxt\text{-}active[simp]$:
 fixes $cid\ t\ n$
 shows $\langle the\text{-}singleton \rightarrow t\rangle_n = n$

$\langle proof \rangle$

lemma *baI*[*intro*]:
  fixes $t\ n\ a$
  assumes $\varphi\ (\sigma_{\mathit{the\text{-}singleton}}(t\ n))$
  shows *eval the-singleton* $t\ t'\ n\ [\varphi]_b$ $\langle proof \rangle$

lemma *baE*[*elim*]:
  fixes $t\ n\ a$
  assumes *eval the-singleton* $t\ t'\ n\ [\varphi]_b$
  shows $\varphi\ (\sigma_{\mathit{the\text{-}singleton}}(t\ n))$ $\langle proof \rangle$

lemma *evtE*[*elim*]:
  fixes $t\ id\ n\ a$
  assumes *eval the-singleton* $t\ t'\ n\ (\Diamond_b\ \gamma)$
  shows $\exists\, n' \geq n.$ *eval the-singleton* $t\ t'\ n'\ \gamma$
$\langle proof \rangle$

lemma *globE*[*elim*]:
  fixes $t\ id\ n\ a$
  assumes *eval the-singleton* $t\ t'\ n\ (\Box_b\ \gamma)$
  shows $\forall\, n' \geq n.$ *eval the-singleton* $t\ t'\ n'\ \gamma$
$\langle proof \rangle$

lemma *untilI*[*intro*]:
  fixes $t{::}nat \Rightarrow cnf$
    and $t'{::}nat \Rightarrow {}'cmp$
    and $n{::}nat$
    and $n'{::}nat$
  assumes $n' \geq n$
    and *eval the-singleton* $t\ t'\ n'\ \gamma$
    and $\bigwedge n''.\ [\![n \leq n'';\ n'' < n]\!] \Longrightarrow$ *eval the-singleton* $t\ t'\ n''\ \gamma'$
  shows *eval the-singleton* $t\ t'\ n\ (\gamma'\ \mathfrak{U}_b\ \gamma)$
$\langle proof \rangle$

lemma *untilE*[*elim*]:
  fixes $t\ id\ n\ \gamma'\ \gamma$
  assumes *eval the-singleton* $t\ t'\ n\ (\gamma'\ \mathfrak{U}_b\ \gamma)$
  shows $\exists\, n' \geq n.$ *eval the-singleton* $t\ t'\ n'\ \gamma \wedge (\forall\, n'' \geq n.\ n'' < n' \longrightarrow$ *eval the-singleton* $t\ t'\ n''\ \gamma')$
$\langle proof \rangle$
end

end

2 A Theory of Publisher-Subscriber Architectures

In the following, we formalize the specification of the publisher subscriber pattern as described in [4].

theory *Publisher-Subscriber*
imports *Singleton*
begin

2.1 Subscriptions

datatype $'evt\ subscription = sub\ 'evt \mid unsub\ 'evt$

2.2 Publisher-Subscriber Architectures

locale *publisher-subscriber* =
  pb: *singleton pbactive pbcmp* +
  sb: *dynamic-component sbcmp sbactive*
    for $pbactive :: 'pid \Rightarrow cnf \Rightarrow bool$ (‹‖-‖-› [0,110]60)
    and $pbcmp :: 'pid \Rightarrow cnf \Rightarrow 'PB$ (‹σ-(-)› [0,110]60)
    and $sbactive :: 'sid \Rightarrow cnf \Rightarrow bool$ (‹‖-‖-› [0,110]60)
    and $sbcmp :: 'sid \Rightarrow cnf \Rightarrow 'SB$ (‹σ-(-)› [0,110]60) +
  fixes $pbsb :: 'PB \Rightarrow ('evt\ set)\ subscription\ set$
    and $pbnt :: 'PB \Rightarrow ('evt \times 'msg)$
    and $sbnt :: 'SB \Rightarrow ('evt \times 'msg)\ set$
    and $sbsb :: 'SB \Rightarrow ('evt\ set)\ subscription$
  assumes *conn1*: $\bigwedge k\ pid.\ \|pid\|_k$
      $\Longrightarrow pbsb\ (\sigma_{pid}(k)) = (\bigcup sid{\in}\{sid.\ \|sid\|_k\}.\ \{sbsb\ (\sigma_{sid}(k))\})$
    and *conn2*: $\bigwedge t\ n\ n''\ sid\ pid\ E\ e\ m.$
      $[\![t \in arch;\ \|pid\|_{t\ n};\ \|sid\|_{t\ n};\ sub\ E = sbsb\ (\sigma_{sid}(t\ n));\ n'' \geq n;\ e \in E;$
      $\nexists n'\ E'.\ n' \geq n \wedge n' \leq n'' \wedge \|sid\|_{t\ n'} \wedge$
        $unsub\ E' = sbsb\ (\sigma_{sid}(t\ n')) \wedge e \in E';$
      $(e,\ m) = pbnt\ (\sigma_{pid}(t\ n''));\ \|sid\|_{t\ n''}]\!]$
      $\Longrightarrow pbnt\ (\sigma_{pid}(t\ n'')) \in sbnt\ (\sigma_{sid}(t\ n''))$
begin

2.2.1 Calculus Interpretation

pb.baIA: $[\![\exists\ i{\geq}n.\ \|c\|_{t\ i};\ \varphi\ (\sigma_c t\ (pb.nxtAct\ c\ t\ n))]\!] \Longrightarrow pb.eval\ c\ t\ t'\ n\ [\varphi]_b$

sb.baIA: $[\![\exists\ i{\geq}n.\ \|c\|_{t\ i};\ \varphi\ (\sigma_c t\ (sb.nxtAct\ c\ t\ n))]\!] \Longrightarrow sb.eval\ c\ t\ t'\ n\ [\varphi]_b$

2.2.2 Results from Singleton

abbreviation $the\text{-}pb :: 'pid$ **where**
$the\text{-}pb \equiv pb.the\text{-}singleton$

pb.ts-prop (1): $\|id\|_k \Longrightarrow id = the\text{-}pb$

pb.ts-prop (2): $\|the\text{-}pb\|_k$

2.2.3 Architectural Guarantees

The following theorem ensures that a subscriber indeed receives all messages associated with an event for which he is subscribed.

theorem *msgDelivery*:
  fixes $t\ n\ n''$ **and** $sid{::}'sid$ **and** $E\ e\ m$
  assumes $t \in arch$
    and $\|sid\|_{t\ n}$
    and $sub\ E = sbsb\ (\sigma_{sid}(t\ n))$
    and $n'' \geq n$
    and $\nexists n'\ E'.\ n' \geq n \wedge n' \leq n'' \wedge \|sid\|_{t\ n'} \wedge unsub\ E' = sbsb(\sigma_{sid}(t\ n'))$
        $\wedge\ e \in E'$
    and $e \in E$

and $(e,m) = pbnt\ (\sigma_{the\text{-}pb}(t\ n''))$
and $\|sid\|_{t\ n''}$
shows $(e,m) \in sbnt\ (\sigma_{sid}(t\ n''))$
⟨*proof*⟩

Since a publisher is actually a singleton, we can provide an alternative version of constraint *conn1*.

lemma *conn1A*:
fixes k
shows $pbsb\ (\sigma_{the\text{-}pb}(k)) = (\bigcup sid{\in}\{sid.\ \|sid\|_k\}.\ \{sbsb\ (\sigma_{sid}(k))\})$
⟨*proof*⟩
end

end

3 A Theory of Blackboard Architectures

In the following, we formalize the specification of the blackboard pattern as described in [4].

theory *Blackboard*
imports *Publisher-Subscriber*
begin

3.1 Problems and Solutions

Blackboards work with problems and solutions for them.

typedecl *PROB*
consts $sb :: (PROB \times PROB)\ set$
axiomatization where *sbWF*: *wf sb*
typedecl *SOL*
consts $solve :: PROB \Rightarrow SOL$

3.2 Blackboard Architectures

In the following, we describe the locale for the blackboard pattern.

locale *blackboard = publisher-subscriber bbactive bbcmp ksactive kscmp bbrp bbcs kscs ksrp*
for $bbactive :: 'bid \Rightarrow cnf \Rightarrow bool$ (‹‖-‖-› [0,110]60)
and $bbcmp :: 'bid \Rightarrow cnf \Rightarrow 'BB$ (‹σ-(-)› [0,110]60)
and $ksactive :: 'kid \Rightarrow cnf \Rightarrow bool$ (‹‖-‖-› [0,110]60)
and $kscmp :: 'kid \Rightarrow cnf \Rightarrow 'KS$ (‹σ-(-)› [0,110]60)
and $bbrp :: 'BB \Rightarrow (PROB\ set)\ subscription\ set$
and $bbcs :: 'BB \Rightarrow (PROB \times SOL)$
and $kscs :: 'KS \Rightarrow (PROB \times SOL)\ set$
and $ksrp :: 'KS \Rightarrow (PROB\ set)\ subscription\ +$
fixes $bbns :: 'BB \Rightarrow (PROB \times SOL)\ set$
and $ksns :: 'KS \Rightarrow (PROB \times SOL)$
and $bbop :: 'BB \Rightarrow PROB$
and $ksop :: 'KS \Rightarrow PROB\ set$
and $prob :: 'kid \Rightarrow PROB$
assumes
ks1: $\forall p.\ \exists ks.\ p{=}prob\ ks$ — Component Parameter
— Assertions about component behavior.
and *bhvbb1*: $\bigwedge t\ t'\ bId\ p\ s.\ [\![t \in arch]\!] \Longrightarrow pb.eval\ bId\ t\ t'\ 0$

$(\Box_b\ ([\lambda bb.\ (p,s){\in}bbns\ bb]_b$
$\longrightarrow^b\ (\Diamond_b\ [\lambda bb.\ (p,s) = bbcs\ bb]_b)))$
and *bhvbb2*: $\bigwedge t\ t'\ bId\ P\ q.\ [\![t{\in}arch]\!] \Longrightarrow pb.eval\ bId\ t\ t'\ 0$
$(\Box_b\ ([\lambda bb.\ sub\ P \in bbrp\ bb \wedge q \in P]_b \longrightarrow^b$
$(\Diamond_b\ [\lambda bb.\ q = bbop\ bb]_b)))$
and *bhvbb3*: $\bigwedge t\ t'\ bId\ p\ .\ [\![t{\in}arch]\!] \Longrightarrow pb.eval\ bId\ t\ t'\ 0$
$(\Box_b\ ([\lambda bb.\ p = bbop(bb)]_b \longrightarrow^b$
$([\lambda bb.\ p{=}bbop(bb)]_b\ \mathfrak{W}_b\ [\lambda bb.\ (p,solve(p)) = bbcs(bb)]_b)))$
and *bhvks1*: $\bigwedge t\ t'\ kId\ p\ P.\ [\![t{\in}arch;\ p = prob\ kId]\!] \Longrightarrow sb.eval\ kId\ t\ t'\ 0$
$(\Box_b\ ([\lambda ks.\ sub\ P = ksrp\ ks]_b \wedge^b$
$(\forall_b\ q.\ ((sb.pred\ (q{\in}P)) \longrightarrow^b (\Diamond_b\ ([\lambda ks.\ (q,solve(q)) \in kscs\ ks]_b))))$
$\longrightarrow^b\ (\Diamond_b\ [\lambda ks.\ (p,\ solve\ p) = ksns\ ks]_b)))$
and *bhvks2*: $\bigwedge t\ t'\ kId\ p\ P\ q.\ [\![t \in arch;p = prob\ kId]\!] \Longrightarrow sb.eval\ kId\ t\ t'\ 0$
$(\Box_b\ [\lambda ks.\ sub\ P = ksrp\ ks \wedge q \in P \longrightarrow (q,p) \in sb]_b)$
and *bhvks3*: $\bigwedge t\ t'\ kId\ p.\ [\![t{\in}arch;p = prob\ kId]\!] \Longrightarrow sb.eval\ kId\ t\ t'\ 0$
$(\Box_b\ ([\lambda ks.\ p{\in}ksop\ ks]_b \longrightarrow^b (\Diamond_b\ [\lambda ks.\ (\exists\,P.\ sub\ P = ksrp\ ks)]_b)))$
and *bhvks4*: $\bigwedge t\ t'\ kId\ p\ P.\ [\![t{\in}arch;\ p{\in}P]\!] \Longrightarrow sb.eval\ kId\ t\ t'\ 0$
$(\Box_b\ ([\lambda ks.\ sub\ P = ksrp\ ks]_b \longrightarrow^b$
$((\neg^b\ (\exists_b\ P'.\ (sb.pred\ (p{\in}P') \wedge^b [\lambda ks.\ unsub\ P' = ksrp\ ks]_b)))\ \mathfrak{W}_b$
$[\lambda ks.\ (p,solve\ p) \in kscs\ ks]_b)))$

— Assertions about component activation.
and *actks*:
$\bigwedge t\ n\ kid\ p.\ [\![t \in arch;\ \|kid\|_{t\ n};\ p{=}prob\ kid;\ p{\in}ksop\ (\sigma_{kid}(t\ n))]\!]$
$\Longrightarrow (\exists\,n'{\geq}n.\ \|kid\|_{t\ n'} \wedge (p,\ solve\ p) = ksns\ (\sigma_{kid}(t\ n')) \wedge$
$(\forall\,n''{\geq}n.\ n''{<}n' \longrightarrow \|kid\|_{t\ n''}))$
$\vee\ (\forall\,n'{\geq}n.\ (\|kid\|_{t\ n'} \wedge (\neg(p,\ solve\ p) = ksns\ (\sigma_{kid}(t\ n')))))$

— Assertions about connections.
and *conn1*: $\bigwedge k\ bid.\ \|bid\|_k$
$\Longrightarrow bbns\ (\sigma_{bid}(k)) = (\bigcup kid{\in}\{kid.\ \|kid\|_k\}.\ \{ksns\ (\sigma_{kid}(k))\})$
and *conn2*: $\bigwedge k\ kid.\ \|kid\|_k$
$\Longrightarrow ksop\ (\sigma_{kid}(k)) = (\bigcup bid{\in}\{bid.\ \|bid\|_k\}.\ \{bbop\ (\sigma_{bid}(k))\})$
begin
notation *sb.lNAct* (‹⟨- ⇐ -⟩-›)
notation *sb.nxtAct* (‹⟨- → -⟩-›)
notation *pb.lNAct* (‹⟨- ⇐ -⟩-›)
notation *pb.nxtAct* (‹⟨- → -⟩-›)

3.2.1 Calculus Interpretation

pb.baIA: $[\![\exists\,i{\geq}n.\ \|c\|_{t\ i};\ \varphi\ (\sigma_c t\ \langle c \rightarrow t\rangle_n)]\!] \Longrightarrow pb.eval\ c\ t\ t'\ n\ [\varphi]_b$

sb.baIA: $[\![\exists\,i{\geq}n.\ \|c\|_{t\ i};\ \varphi\ (\sigma_c t\ \langle c \rightarrow t\rangle_n)]\!] \Longrightarrow sb.eval\ c\ t\ t'\ n\ [\varphi]_b$

3.2.2 Results from Singleton

abbreviation *the-bb* ≡ *the-pb*

pb.ts-prop (*1*): $\|id\|_k \Longrightarrow id = the\text{-}bb$

pb.ts-prop (*2*): $\|the\text{-}bb\|_k$

3.2.3 Results from Publisher Subscriber

msgDelivery: $[\![t \in arch;\ \|sid\|_{t\ n};\ sub\ E = ksrp\ (\sigma_{sid} t\ n);\ n \leq n'';\ \nexists n'\ E'.\ n \leq n' \wedge n' \leq n'' \wedge \|sid\|_{t\ n'} \wedge unsub\ E' = ksrp\ (\sigma_{sid} t\ n') \wedge e \in E';\ e \in E;\ (e,\ m) = bbcs\ (\sigma_{the\text{-}bb} t\ n'');\ \|sid\|_{t\ n''}]\!] \Longrightarrow (e,\ m) \in kscs\ (\sigma_{sid} t\ n'')$

lemma *conn2-bb*:
  fixes k **and** $kid::'kid$
  assumes $\|kid\|_k$
  shows $bbop\ (\sigma_{the\text{-}bb}(k)) \in ksop\ (\sigma_{kid}(k))$
$\langle proof \rangle$

3.2.4 Knowledge Sources

In the following we introduce an abbreviation for knowledge sources which are able to solve a specific problem.

definition sKs:: $PROB \Rightarrow 'kid$ **where**
  $sKs\ p \equiv (SOME\ kid.\ p = prob\ kid)$

lemma *sks-prob*:
  $p = prob\ (sKs\ p)$
$\langle proof \rangle$

3.2.5 Architectural Guarantees

The following theorem verifies that a problem is eventually solved by the pattern even if no knowledge source exist which can solve the problem on its own. It assumes, however, that for every open sub problem, a corresponding knowledge source able to solve the problem will be eventually activated.

lemma *pSolved-Ind*:
  fixes t **and** $t'::nat \Rightarrow 'BB$ **and** p **and** $t''::nat \Rightarrow 'KS$
  assumes $t \in arch$ **and**
    $\forall n.\ (\exists n' \geq n.\ \|sKs\ (bbop(\sigma_{the\text{-}bb}(t\ n)))\|_{t\ n'})$
  shows
    $\forall n.\ (\exists P.\ sub\ P \in bbrp(\sigma_{the\text{-}bb}(t\ n)) \wedge p \in P) \longrightarrow$
      $(\exists m \geq n.\ (p, solve(p)) = bbcs\ (\sigma_{the\text{-}bb}(t\ m)))$
— The proof is by well-founded induction over the subproblem relation *sb*
$\langle proof \rangle$

theorem *pSolved*:
  fixes t **and** $t'::nat \Rightarrow 'BB$ **and** $t''::nat \Rightarrow 'KS$
  assumes $t \in arch$ **and**
    $\forall n.\ (\exists n' \geq n.\ \|sKs\ (bbop(\sigma_{the\text{-}bb}(t\ n)))\|_{t\ n'})$
  shows
    $\forall n.\ (\forall P.\ (sub\ P \in bbrp(\sigma_{the\text{-}bb}(t\ n))$
      $\longrightarrow (\forall p \in P.\ (\exists m \geq n.\ (p, solve(p)) = bbcs\ (\sigma_{the\text{-}bb}(t\ m))))))$
  $\langle proof \rangle$
end

end

4 Some Auxiliary Results

theory *Auxiliary* **imports** *Main*

begin

lemma *disjE3*: $P \lor Q \lor R \Longrightarrow (P \Longrightarrow S) \Longrightarrow (Q \Longrightarrow S) \Longrightarrow (R \Longrightarrow S) \Longrightarrow S$ $\langle proof \rangle$

lemma *ge-induct*[*consumes 1*, *case-names step*]:
  fixes $i{::}nat$ **and** $j{::}nat$ **and** $P{::}nat \Rightarrow bool$
  shows $i \leq j \Longrightarrow (\bigwedge n.\ i \leq n \Longrightarrow ((\forall m \geq i.\ \ m{<}n \longrightarrow P\ m) \Longrightarrow P\ n)) \Longrightarrow P\ j$
$\langle proof \rangle$

lemma *my-induct*[*consumes 1*, *case-names base step*]:
  fixes $P{::}nat{\Rightarrow}bool$
assumes *less*: $i \leq j$
    and *base*: $P\ j$
    and *step*: $\bigwedge n.\ i \leq n \Longrightarrow n < j \Longrightarrow (\forall n'{>}n.\ n'{\leq}j \longrightarrow P\ n') \Longrightarrow P\ n$
  shows $P\ i$
$\langle proof \rangle$

lemma *Greatest-ex-le-nat*: **assumes** $\exists k.\ P\ k \land (\forall k'.\ P\ k' \longrightarrow k' \leq k)$ **shows** $\neg(\exists n'{>}Greatest\ P.\ P\ n')$
  $\langle proof \rangle$

lemma *cardEx*: **assumes** *finite A* **and** *finite B* **and** $card\ A > card\ B$ **shows** $\exists x{\in}A.\ \neg\ x{\in}B$
$\langle proof \rangle$

lemma *cardshift*: $card\ \{i{::}nat.\ i{>}n \land i \leq n' \land p\ (n'' + i)\} = card\ \{i.\ i{>}(n + n'') \land i \leq (n' + n'') \land$
$p\ i\}$
$\langle proof \rangle$

end

5 Relative Frequency LTL

theory *RF-LTL*
  imports *Main HOL−Library.Sublist Auxiliary DynamicArchitectures.Dynamic-Architecture-Calculus*
begin

type-synonym $'s\ seq = nat \Rightarrow 's$

abbreviation $ccard\ n\ n'\ p \equiv card\ \{i.\ i{>}n \land i \leq n' \land p\ i\}$

lemma *ccard-same*:
  assumes $\neg\ p\ (Suc\ n')$
  shows $ccard\ n\ n'\ p = ccard\ n\ (Suc\ n')\ p$
$\langle proof \rangle$

lemma *ccard-zero*[*simp*]:
  fixes $n{::}nat$
  shows $ccard\ n\ n\ p = 0$
  $\langle proof \rangle$

lemma *ccard-inc*:
  assumes $p\ (Suc\ n')$
    and $n' \geq n$
  shows $ccard\ n\ (Suc\ n')\ p = Suc\ (ccard\ n\ n'\ p)$
$\langle proof \rangle$

lemma *ccard-mono*:
  assumes $n' \geq n$
  shows $n'' \geq n' \Longrightarrow \mathit{ccard}\ n\ (n''::\mathit{nat})\ p \geq \mathit{ccard}\ n\ n'\ p$
$\langle \mathit{proof} \rangle$

lemma *ccard-ub*[*simp*]:
  $\mathit{ccard}\ n\ n'\ p \leq \mathit{Suc}\ n' - n$
$\langle \mathit{proof} \rangle$

lemma *ccard-sum*:
  fixes $n::\mathit{nat}$
  assumes $n' \geq n''$
    and $n'' \geq n$
  shows $\mathit{ccard}\ n\ n'\ P = \mathit{ccard}\ n\ n''\ P + \mathit{ccard}\ n''\ n'\ P$
$\langle \mathit{proof} \rangle$

lemma *ccard-ex*:
  fixes $n::\mathit{nat}$
  shows $c \geq 1 \Longrightarrow c < \mathit{ccard}\ n\ n''\ P \Longrightarrow \exists\, n' < n''.\ n' > n \wedge \mathit{ccard}\ n\ n'\ P = c$
$\langle \mathit{proof} \rangle$

lemma *ccard-freq*:
  assumes $(n'::\mathit{nat}) \geq n$
    and $\mathit{ccard}\ n\ n'\ P > \mathit{ccard}\ n\ n'\ Q + \mathit{cnf}$
  shows $\exists\, n'\ n''.\ \mathit{ccard}\ n'\ n''\ P > \mathit{cnf} \wedge \mathit{ccard}\ n'\ n''\ Q \leq \mathit{cnf}$
$\langle \mathit{proof} \rangle$

locale *honest* =
  fixes $\mathit{bc}::\ ('a\ \mathit{list})\ \mathit{seq}$
    and $n::\mathit{nat}$
  assumes *growth*: $n' \neq 0 \Longrightarrow n' \leq n \Longrightarrow \mathit{bc}\ n' = \mathit{bc}\ (n'-1) \vee (\exists\, b.\ \mathit{bc}\ n' = \mathit{bc}\ (n' - 1)\ @\ b)$
begin
end

locale *dishonest* =
  fixes $\mathit{bc}::\ ('a\ \mathit{list})\ \mathit{seq}$
    and $\mathit{mining}::\mathit{bool}\ \mathit{seq}$
  assumes *growth*: $\bigwedge n::\mathit{nat}.\ \mathit{prefix}\ (\mathit{bc}\ (\mathit{Suc}\ n))\ (\mathit{bc}\ n) \vee (\exists\, b::'a.\ \mathit{bc}\ (\mathit{Suc}\ n) = \mathit{bc}\ n\ @\ [b]) \wedge \mathit{mining}$
$(\mathit{Suc}\ n)$
begin

lemma *prefix-save*:
  assumes $\mathit{prefix}\ \mathit{sbc}\ (\mathit{bc}\ n')$
    and $\forall\, n''' > n'.\ n''' \leq n'' \longrightarrow \mathit{length}\ (\mathit{bc}\ n''') \geq \mathit{length}\ \mathit{sbc}$
  shows $n'' \geq n' \Longrightarrow \mathit{prefix}\ \mathit{sbc}\ (\mathit{bc}\ n'')$
$\langle \mathit{proof} \rangle$

theorem *prefix-length*:
  assumes $\mathit{prefix}\ \mathit{sbc}\ (\mathit{bc}\ n')$ **and** $\neg\ \mathit{prefix}\ \mathit{sbc}\ (\mathit{bc}\ n'')$ **and** $n' \leq n''$
  shows $\exists\, n''' > n'.\ n''' \leq n'' \wedge \mathit{length}\ (\mathit{bc}\ n''') < \mathit{length}\ \mathit{sbc}$
$\langle \mathit{proof} \rangle$

theorem *grow-mining*:
  assumes $\mathit{length}\ (\mathit{bc}\ n) < \mathit{length}\ (\mathit{bc}\ (\mathit{Suc}\ n))$
  shows $\mathit{mining}\ (\mathit{Suc}\ n)$

$\langle proof \rangle$

lemma *length-suc-length*:
$length\ (bc\ (Suc\ n)) \leq Suc\ (length\ (bc\ n))$
$\langle proof \rangle$

end

locale *dishonest-growth* =
fixes bc:: *nat seq*
and *mining*:: $nat \Rightarrow bool$
assumes *as1*: $\bigwedge n::nat.\ bc\ (Suc\ n) \leq Suc\ (bc\ n)$
and *as2*: $\bigwedge n::nat.\ bc\ (Suc\ n) > bc\ n \Longrightarrow mining\ (Suc\ n)$
begin

end

sublocale *dishonest* $\subseteq$ *dishonest-growth* $\lambda n.\ length\ (bc\ n)$ $\langle proof \rangle$

context *dishonest-growth*
begin
theorem *ccard-diff-lgth*:
$n' \geq n \Longrightarrow ccard\ n\ n'\ (\lambda n.\ mining\ n) \geq (bc\ n' - bc\ n)$
$\langle proof \rangle$
end

locale *honest-growth* =
fixes bc:: *nat seq*
and *mining*:: $nat \Rightarrow bool$
and *init*:: *nat*
assumes *as1*: $\bigwedge n::nat.\ bc\ (Suc\ n) \geq bc\ n$
and *as2*: $\bigwedge n::nat.\ mining\ (Suc\ n) \Longrightarrow bc\ (Suc\ n) > bc\ n$
begin
lemma *grow-mono*: $n' \geq n \Longrightarrow bc\ n' \geq bc\ n$
$\langle proof \rangle$

theorem *ccard-diff-lgth*:
shows $n' \geq n \Longrightarrow bc\ n' - bc\ n \geq ccard\ n\ n'\ (\lambda n.\ mining\ n)$
$\langle proof \rangle$
end

locale *bounded-growth* = *hg*: *honest-growth hbc hmining* + *dg*: *dishonest-growth dbc dmining*
for *hbc*:: *nat seq*
and *dbc*:: *nat seq*
and *hmining*:: $nat \Rightarrow bool$
and *dmining*:: $nat \Rightarrow bool$
and *sbc*::*nat*
and *cnf*::*nat* +
assumes *fair*: $\bigwedge n\ n'.\ ccard\ n\ n'\ (\lambda n.\ dmining\ n) > cnf \Longrightarrow ccard\ n\ n'\ (\lambda n.\ hmining\ n) > cnf$
and *a2*: $hbc\ 0 \geq sbc+cnf$
and *a3*: $dbc\ 0 < sbc$
begin

theorem *hn-upper-bound*: **shows** $dbc\ n < hbc\ n$
$\langle proof \rangle$

end

end

6 Blockchain Architectures

theory *Blockchain* **imports** *Auxiliary DynamicArchitectures.Dynamic-Architecture-Calculus RF-LTL*
begin

6.1 Blockchains

A blockchain itself is modeled as a simple list.

type-synonym $'a\ BC = 'a\ list$

abbreviation *max-cond*:: $('a\ BC)\ set \Rightarrow 'a\ BC \Rightarrow bool$
  where *max-cond* $B\ b \equiv b \in B \wedge (\forall\, b'{\in}B.\ length\ b' \leq length\ b)$

no-syntax
  -MAX1 :: $pttrns \Rightarrow 'b \Rightarrow 'b$ (‹(‹*indent=3 notation=*‹*binder MAX*››*MAX* -./ -)› [*0*, *10*] *10*)
  -MAX :: $pttrn \Rightarrow 'a\ set \Rightarrow 'b \Rightarrow 'b$ (‹(‹*indent=3 notation=*‹*binder MAX*››*MAX* -∈-./ -)› [*0*, *0*, *10*] *10*)

definition *MAX*:: $('a\ BC)\ set \Rightarrow 'a\ BC$
  where $MAX\ B = (SOME\ b.\ max\text{-}cond\ B\ b)$

lemma *max-ex*:
  fixes $XS{::}('a\ BC)\ set$
  assumes $XS \neq \{\}$
    and *finite XS*
  shows $\exists\, xs{\in}XS.\ (\forall\, ys{\in}XS.\ length\ ys \leq length\ xs)$
⟨*proof*⟩

lemma *max-prop*:
  fixes $XS{::}('a\ BC)\ set$
  assumes $XS \neq \{\}$
    and *finite XS*
  shows $MAX\ XS \in XS$
    and $\forall\, b'{\in}XS.\ length\ b' \leq length\ (MAX\ XS)$
⟨*proof*⟩

lemma *max-less*:
  fixes $b{::}'a\ BC$ **and** $b'{::}'a\ BC$ **and** $B{::}('a\ BC)\ set$
  assumes $b{\in}B$
    and *finite B*
    and $length\ b > length\ b'$
  shows $length\ (MAX\ B) > length\ b'$
⟨*proof*⟩

6.2 Blockchain Architectures

In the following we describe the locale for blockchain architectures.

locale *Blockchain* = *dynamic-component cmp active*
  for *active* :: $'nid \Rightarrow cnf \Rightarrow bool$ (‹‖-‖-› [*0*,*110*]*60*)

and *cmp* :: *'nid* ⇒ *cnf* ⇒ *'ND* (‹$\sigma_{\_}$(-)› *[0,110]60*) +
fixes *pin* :: *'ND* ⇒ (*'nid BC*) *set*
and *pout* :: *'ND* ⇒ *'nid BC*
and *bc* :: *'ND* ⇒ *'nid BC*
and *mining* :: *'ND* ⇒ *bool*
and *honest* :: *'nid* ⇒ *bool*
and *actHn* :: *cnf* ⇒ *'nid set*
and *actDn* :: *cnf* ⇒ *'nid set*
and *PoW*:: *trace* ⇒ *nat* ⇒ *nat*
and *hmining*:: *trace* ⇒ *nat* ⇒ *bool*
and *dmining*:: *trace* ⇒ *nat* ⇒ *bool*
and *cb*:: *nat*
defines *actHn k* ≡ {*nid*. $\|nid\|_k$ ∧ *honest nid*}
and *actDn k* ≡ {*nid*. $\|nid\|_k$ ∧ ¬ *honest nid*}
and *PoW t n* ≡ (*LEAST x*. ∀ *nid*∈*actHn* (*t n*). *length* (*bc* (σ_{nid}(*t n*))) ≤ *x*)
and *hmining t* ≡ (λ*n*. ∃ *nid*∈*actHn* (*t n*). *mining* (σ_{nid}(*t n*)))
and *dmining t* ≡ (λ*n*. ∃ *nid*∈*actDn* (*t n*). *mining* (σ_{nid}(*t n*)))
assumes *consensus*: ⋀*nid t t' bc'*::(*'nid BC*). ⟦*honest nid*⟧ ⟹ *eval nid t t' 0*
($\Box_b$ ([λ*nd*. *bc'* = (*if* (∃ *b*∈*pin nd*. *length b* > *length* (*bc nd*)) *then* (*MAX* (*pin nd*)) *else* (*bc nd*))]$_b$
$\longrightarrow^b$ $\bigcirc_b$ [λ*nd*.(¬ *mining nd* ∧ *bc nd* = *bc'* ∨ *mining nd* ∧ (∃ *b*. *bc nd* = *bc'* @ [*b*]))]$_b$))
and *attacker*: ⋀*nid t t' bc'*. ⟦¬ *honest nid*⟧ ⟹ *eval nid t t' 0*
($\Box_b$ ([λ*nd*. *bc'* = (*SOME b*. *b* ∈ (*pin nd* ∪ {*bc nd*}))]$_b$ $\longrightarrow^b$
$\bigcirc_b$ [λ*nd*.(¬ *mining nd* ∧ *prefix* (*bc nd*) *bc'* ∨ *mining nd* ∧ (∃ *b*. *bc nd* = *bc'* @ [*b*]))]$_b$))
and *forward*: ⋀*nid t t'*. *eval nid t t' 0* ($\Box_b$ [λ*nd*. *pout nd* = *bc nd*]$_b$)
— At each time point a node will forward its blockchain to the network
and *init*: ⋀*nid t t'*. *eval nid t t' 0* [λ*nd*. *bc nd*=[]]$_b$
and *conn*: ⋀*k nid*. ⟦$\|nid\|_k$; *honest nid*⟧
⟹ *pin* (*cmp nid k*) = (⋃ *nid'*∈*actHn k*. {*pout* (*cmp nid' k*)})
and *act*: ⋀*t n*::*nat*. *finite* {*nid*::*'nid*. $\|nid\|_{t\ n}$}
and *actHn*: ⋀*t n*::*nat*. ∃ *nid*. *honest nid* ∧ $\|nid\|_{t\ n}$ ∧ $\|nid\|_{t\ (Suc\ n)}$
and *fair*: ⋀*n n'*. *ccard n n'* (*dmining t*) > *cb* ⟹ *ccard n n'* (*hmining t*) > *cb*
and *closed*: ⋀*t nid b n*::*nat*. ⟦$\|nid\|_{t\ n}$; *b* ∈ *pin* (σ_{nid}(*t n*))⟧ ⟹ ∃ *nid'*. $\|nid'\|_{t\ n}$ ∧ *bc* ($\sigma_{nid'}$(*t n*))
= *b*
and *mine*: ⋀*t nid n*::*nat*. ⟦*honest nid*; $\|nid\|_{t\ (Suc\ n)}$; *mining* (σ_{nid}(*t* (*Suc n*)))⟧ ⟹ $\|nid\|_{t\ n}$
begin

lemma *init-model*:
assumes ¬ (∃ *n'*. *latestAct-cond nid t n n'*)
and $\|nid\|_{t\ n}$
shows *bc* (σ_{nid}*t n*) = []
⟨*proof*⟩

lemma *fwd-bc*:
fixes *nid* **and** *t*::*nat* ⇒ *cnf* **and** *t'*::*nat* ⇒ *'ND*
assumes $\|nid\|_{t\ n}$
shows *pout* (σ_{nid}*t n*) = *bc* (σ_{nid}*t n*)
⟨*proof*⟩

lemma *finite-input*:
fixes *t n nid*
assumes $\|nid\|_{t\ n}$
defines *dep nid'* ≡ *pout* ($\sigma_{nid'}$(*t n*))
shows *finite* (*pin* (*cmp nid* (*t n*)))
⟨*proof*⟩

lemma *nempty-input*:
  fixes t n nid
  assumes $\|nid\|_{t\ n}$
    and *honest nid*
  shows $pin\ (cmp\ nid\ (t\ n)) \neq \{\}$ ⟨*proof*⟩

lemma *onlyone*:
  assumes $\exists\, n' \geq n.\ \|tid\|_{t\ n'}$
    and $\exists\, n' < n.\ \|tid\|_{t\ n'}$
  shows $\exists !i.\ \langle tid \leftarrow t \rangle_n \leq i \wedge i < \langle tid \rightarrow t \rangle_n \wedge \|tid\|_{t\ i}$
⟨*proof*⟩

6.2.1 Component Behavior

lemma *bhv-hn-ex*:
  fixes t **and** $t'::nat \Rightarrow 'ND$ **and** tid
  assumes *honest tid*
    and $\exists\, n' \geq n.\ \|tid\|_{t\ n'}$
    and $\exists\, n' < n.\ \|tid\|_{t\ n'}$
    and $\exists\, b \in pin\ (\sigma_{tid} t\ \langle tid \leftarrow t \rangle_n).\ length\ b > length\ (bc\ (\sigma_{tid} t\ \langle tid \leftarrow t \rangle_n))$
  shows $\neg\ mining\ (\sigma_{tid} t\ \langle tid \rightarrow t \rangle_n) \wedge bc\ (\sigma_{tid} t\ \langle tid \rightarrow t \rangle_n) =$
    $Blockchain.MAX\ (pin\ (\sigma_{tid} t\ \langle tid \leftarrow t \rangle_n)) \vee mining\ (\sigma_{tid} t\ \langle tid \rightarrow t \rangle_n) \wedge$
    $(\exists\, b.\ bc\ (\sigma_{tid} t\ \langle tid \rightarrow t \rangle_n) = Blockchain.MAX\ (pin\ (\sigma_{tid} t\ \langle tid \leftarrow t \rangle_n))\ @\ [b])$
⟨*proof*⟩

lemma *bhv-hn-in*:
  fixes t **and** $t'::nat \Rightarrow 'ND$ **and** tid
  assumes *honest tid*
    and $\exists\, n' \geq n.\ \|tid\|_{t\ n'}$
    and $\exists\, n' < n.\ \|tid\|_{t\ n'}$
    and $\neg\ (\exists\, b \in pin\ (\sigma_{tid} t\ \langle tid \leftarrow t \rangle_n).\ length\ b > length\ (bc\ (\sigma_{tid} t\ \langle tid \leftarrow t \rangle_n)))$
  shows $\neg\ mining\ (\sigma_{tid} t\ \langle tid \rightarrow t \rangle_n) \wedge bc\ (\sigma_{tid} t\ \langle tid \rightarrow t \rangle_n) = bc\ (\sigma_{tid} t\ \langle tid \leftarrow t \rangle_n) \vee$
    $mining\ (\sigma_{tid} t\ \langle tid \rightarrow t \rangle_n) \wedge (\exists\, b.\ bc\ (\sigma_{tid} t\ \langle tid \rightarrow t \rangle_n) = bc\ (\sigma_{tid} t\ \langle tid \leftarrow t \rangle_n)\ @\ [b])$
⟨*proof*⟩

lemma *bhv-hn-context*:
  assumes *honest tid*
      and $\|tid\|_{t\ n}$
      and $\exists\, n' < n.\ \|tid\|_{t\ n'}$
    shows $\exists\, nid'.\ \|nid'\|_{t\ \langle tid \leftarrow t \rangle_n} \wedge (mining\ (\sigma_{tid} t\ n) \wedge (\exists\, b.\ bc\ (\sigma_{tid} t\ n) = bc\ (\sigma_{nid'} t\ \langle tid \leftarrow t \rangle_n)\ @$
$[b]) \vee$
      $\neg\ mining\ (\sigma_{tid} t\ n) \wedge bc\ (\sigma_{tid} t\ n) = bc\ (\sigma_{nid'} t\ \langle tid \leftarrow t \rangle_n))$
⟨*proof*⟩

lemma *bhv-dn*:
  fixes t **and** $t'::nat \Rightarrow 'ND$ **and** uid
  assumes $\neg$ *honest uid*
    and $\exists\, n' \geq n.\ \|uid\|_{t\ n'}$
    and $\exists\, n' < n.\ \|uid\|_{t\ n'}$
  shows $\neg\ mining\ (\sigma_{uid} t\ \langle uid \rightarrow t \rangle_n) \wedge prefix\ (bc\ (\sigma_{uid} t\ \langle uid \rightarrow t \rangle_n))\ (SOME\ b.\ b \in pin\ (\sigma_{uid} t\ \langle uid$
$\leftarrow t \rangle_n) \cup \{bc\ (\sigma_{uid} t\ \langle uid \leftarrow t \rangle_n)\})$
      $\vee\ mining\ (\sigma_{uid} t\ \langle uid \rightarrow t \rangle_n) \wedge (\exists\, b.\ bc\ (\sigma_{uid} t\ \langle uid \rightarrow t \rangle_n) = (SOME\ b.\ b \in pin\ (\sigma_{uid} t\ \langle uid \leftarrow$
$t \rangle_n) \cup \{bc\ (\sigma_{uid} t\ \langle uid \leftarrow t \rangle_n)\})\ @\ [b])$
⟨*proof*⟩

lemma *bhv-dn-context*:

assumes $\neg$ *honest uid*
and $\|uid\|_{t\ n}$
and $\exists\, n'{<}n.\ \|uid\|_{t\ n'}$
shows $\exists\, nid'.\ \|nid'\|_{t\ \langle uid \leftarrow t\rangle_n} \wedge (mining\ (\sigma_{uid} t\ n) \wedge (\exists\, b.\ prefix\ (bc\ (\sigma_{uid} t\ n))\ (bc\ (\sigma_{nid'} t\ \langle uid \leftarrow t\rangle_n)\ @\ [b]))$
$\vee \neg\ mining\ (\sigma_{uid} t\ n) \wedge prefix\ (bc\ (\sigma_{uid} t\ n))\ (bc\ (\sigma_{nid'} t\ \langle uid \leftarrow t\rangle_n)))$
$\langle proof \rangle$

6.2.2 Maximal Honest Blockchains

abbreviation *mbc-cond*:: $trace \Rightarrow nat \Rightarrow 'nid \Rightarrow bool$
where *mbc-cond t n nid* $\equiv nid{\in}actHn\ (t\ n) \wedge (\forall\, nid'{\in}actHn\ (t\ n).\ length\ (bc\ (\sigma_{nid'}(t\ n))) \leq length\ (bc\ (\sigma_{nid}(t\ n))))$

lemma *mbc-ex*:
fixes *t n*
shows $\exists\, x.$ *mbc-cond t n x*
$\langle proof \rangle$

definition *MBC*:: $trace \Rightarrow nat \Rightarrow 'nid$
where *MBC t n* = (*SOME b. mbc-cond t n b*)

lemma *mbc-prop*[*simp*]:
shows *mbc-cond t n* (*MBC t n*)
$\langle proof \rangle$

6.2.3 Honest Proof of Work

An important construction is the maximal proof of work available in the honest community. The construction was already introduces in the locale itself since it was used to express some of the locale assumptions.

abbreviation *pow-cond*:: $trace \Rightarrow nat \Rightarrow nat \Rightarrow bool$
where *pow-cond t n n'* $\equiv \forall\, nid{\in}actHn\ (t\ n).\ length\ (bc\ (\sigma_{nid}(t\ n))) \leq n'$

lemma *pow-ex*:
fixes *t n*
shows *pow-cond t n* $(length\ (bc\ (\sigma_{MBC\ t\ n}(t\ n))))$
and $\forall\, x'.$ *pow-cond t n x'* $\longrightarrow x' {\geq} length\ (bc\ (\sigma_{MBC\ t\ n}(t\ n)))$
$\langle proof \rangle$

lemma *pow-prop*:
pow-cond t n (*PoW t n*)
$\langle proof \rangle$

lemma *pow-eq*:
fixes *n*
assumes $\exists\, tid{\in}actHn\ (t\ n).\ length\ (bc\ (\sigma_{tid}(t\ n))) = x$
and $\forall\, tid{\in}actHn\ (t\ n).\ length\ (bc\ (\sigma_{tid}(t\ n))) \leq x$
shows *PoW t n* = x
$\langle proof \rangle$

lemma *pow-mbc*:
shows $length\ (bc\ (\sigma_{MBC\ t\ n} t\ n)) = PoW\ t\ n$
$\langle proof \rangle$

lemma *pow-less*:
  fixes $t\ n\ nid$
  assumes $pow\text{-}cond\ t\ n\ x$
  shows $PoW\ t\ n \leq x$
$\langle proof\rangle$

lemma *pow-le-max*:
  assumes $honest\ tid$
    and $\|tid\|_{t\ n}$
  shows $PoW\ t\ n \leq length\ (MAX\ (pin\ (\sigma_{tid} t\ n)))$
$\langle proof\rangle$

lemma *pow-ge-lgth*:
  assumes $honest\ tid$
    and $\|tid\|_{t\ n}$
  shows $length\ (bc\ (\sigma_{tid} t\ n)) \leq PoW\ t\ n$
$\langle proof\rangle$

lemma *pow-le-lgth*:
  assumes $honest\ tid$
    and $\|tid\|_{t\ n}$
    and $\neg(\exists\, b{\in}pin\ (\sigma_{tid} t\ n).\ length\ b > length\ (bc\ (\sigma_{tid} t\ n)))$
  shows $length\ (bc\ (\sigma_{tid} t\ n)) \geq PoW\ t\ n$
$\langle proof\rangle$

lemma *pow-mono*:
  shows $n'{\geq}n \Longrightarrow PoW\ t\ n' \geq PoW\ t\ n$
$\langle proof\rangle$

lemma *pow-equals*:
  assumes $PoW\ t\ n = PoW\ t\ n'$
  and $n'{\geq}n$
  and $n''{\geq}n$
  and $n''{\leq}n'$
shows $PoW\ t\ n = PoW\ t\ n''$ $\langle proof\rangle$

lemma *pow-mining-suc*:
    assumes $hmining\ t\ (Suc\ n)$
    shows $PoW\ t\ n < PoW\ t\ (Suc\ n)$
$\langle proof\rangle$

6.2.4 History

In the following we introduce an operator which extracts the development of a blockchain up to a time point n.

abbreviation $his\text{-}prop\ t\ n\ nid\ n'\ nid'\ x \equiv$
  $(\exists\, n.\ latestAct\text{-}cond\ nid'\ t\ n'\ n) \wedge \|snd\ x\|_{t\ (fst\ x)} \wedge fst\ x = \langle nid' \leftarrow t\rangle_{n'} \wedge$
  $(prefix\ (bc\ (\sigma_{nid'}(t\ n')))\ (bc\ (\sigma_{snd\ x}(t\ (fst\ x)))) \vee$
    $(\exists\, b.\ bc\ (\sigma_{nid'}(t\ n')) = (bc\ (\sigma_{snd\ x}(t\ (fst\ x))))\ @\ [b] \wedge mining\ (\sigma_{nid'}(t\ n'))))$

inductive-set
$his{::}\ trace \Rightarrow nat \Rightarrow 'nid \Rightarrow (nat \times 'nid)\ set$
  for $t{::}trace$ **and** $n{::}nat$ **and** $nid{::}'nid$
  where $[\![\|nid\|_{t\ n}]\!] \Longrightarrow (n,nid) \in his\ t\ n\ nid$
  | $[\![(n',nid') \in his\ t\ n\ nid;\ \exists\, x.\ his\text{-}prop\ t\ n\ nid\ n'\ nid'\ x]\!] \Longrightarrow (SOME\ x.\ his\text{-}prop\ t\ n\ nid\ n'\ nid'\ x) \in$

his t n nid

lemma *his-act*:
  assumes $(n',nid') \in$ *his t n nid*
  shows $\|nid'\|_{t\ n'}$
  ⟨*proof*⟩

In addition we also introduce an operator to obtain the predecessor of a blockchains development.

definition *hisPred*
  where *hisPred t n nid n'* $\equiv$ (*GREATEST n''*. $\exists$ *nid'*. $(n'',nid') \in$ *his t n nid* $\wedge$ $n'' < n'$)

lemma *hisPrev-prop*:
  assumes $\exists\, n''<n'$. $\exists$ *nid'*. $(n'',nid') \in$ *his t n nid*
  shows *hisPred t n nid n'* $< n'$ **and** $\exists$ *nid'*. (*hisPred t n nid n'*,*nid'*)$\in$ *his t n nid*
⟨*proof*⟩

lemma *hisPrev-nex-less*:
  assumes $\exists\, n''<n'$. $\exists$ *nid'*. $(n'',nid') \in$ *his t n nid*
  shows $\neg(\exists\, x \in$ *his t n nid*. *fst x* $< n'$ $\wedge$ *fst x*$>$*hisPred t n nid n'*)
⟨*proof*⟩

lemma *his-le*:
  assumes $x \in$ *his t n nid*
  shows *fst x*$\leq n$
⟨*proof*⟩

lemma *his-determ-base*:
  shows $(n,\ nid') \in$ *his t n nid* $\Longrightarrow$ *nid'*=*nid*
⟨*proof*⟩

lemma *hisPrev-same*:
  assumes $\exists\, n'<n''$. $\exists$ *nid'*. $(n',nid') \in$ *his t n nid*
  and $\exists\, n''<n'$. $\exists$ *nid'*. $(n'',nid') \in$ *his t n nid*
  and $(n',nid') \in$ *his t n nid*
  and $(n'',nid'') \in$ *his t n nid*
  and *hisPred t n nid n'*=*hisPred t n nid n''*
  shows $n'=n''$
⟨*proof*⟩

lemma *his-determ-ext*:
  shows $n'\leq n \Longrightarrow (\exists$ *nid'*. $(n',nid') \in$*his t n nid*) $\Longrightarrow$ ($\exists!$*nid'*. $(n',nid') \in$*his t n nid*) $\wedge$
    (($\exists\, n''<n'$. $\exists$ *nid'*. $(n'',nid') \in$ *his t n nid*) $\longrightarrow$ ($\exists\, x$. *his-prop t n nid n'* (*THE nid'*. $(n',nid') \in$*his t n nid*) *x*) $\wedge$
    (*hisPred t n nid n'*, (*SOME nid'*. (*hisPred t n nid n'*, *nid'*) $\in$ *his t n nid*)) = (*SOME x*. *his-prop t n nid n'* (*THE nid'*. $(n',nid') \in$*his t n nid*) *x*))
⟨*proof*⟩

corollary *his-determ-ex*:
  assumes $(n',nid') \in$*his t n nid*
  shows $\exists!$*nid'*. $(n',nid') \in$*his t n nid*
  ⟨*proof*⟩

corollary *his-determ*:
  assumes $(n',nid') \in$*his t n nid*

and $(n',nid'') \in his\ t\ n\ nid$
shows $nid'=nid''$ $\langle proof \rangle$

corollary *his-determ-the*:
assumes $(n',nid') \in his\ t\ n\ nid$
shows $(THE\ nid'.\ (n',\ nid') \in his\ t\ n\ nid) = nid'$
$\langle proof \rangle$

6.2.5 Blockchain Development

definition $devBC::trace \Rightarrow nat \Rightarrow 'nid \Rightarrow nat \Rightarrow 'nid\ option$
where $devBC\ t\ n\ nid\ n' \equiv$
$(if\ (\exists\, nid'.\ (n',\ nid') \in\ his\ t\ n\ nid)\ then\ (Some\ (THE\ nid'.\ (n',\ nid') \in his\ t\ n\ nid))$
$else\ Option.None)$

lemma *devBC-some*[*simp*]: **assumes** $\|nid\|_{t\ n}$ **shows** $devBC\ t\ n\ nid\ n = Some\ nid$
$\langle proof \rangle$

lemma *devBC-act*: **assumes** $\neg\ Option.is\text{-}none\ (devBC\ t\ n\ nid\ n')$ **shows** $\|the\ (devBC\ t\ n\ nid\ n')\|_{t\ n'}$
$\langle proof \rangle$

lemma *his-ex*:
assumes $\neg Option.is\text{-}none\ (devBC\ t\ n\ nid\ n')$
shows $\exists\, nid'.\ (n',nid') \in his\ t\ n\ nid$
$\langle proof \rangle$

lemma *devExt-nopt-leq*:
assumes $\neg Option.is\text{-}none\ (devBC\ t\ n\ nid\ n')$
shows $n' \leq n$
$\langle proof \rangle$

An extended version of the development in which deactivations are filled with the last value.

function $devExt::trace \Rightarrow nat \Rightarrow 'nid \Rightarrow nat \Rightarrow nat \Rightarrow 'nid\ BC$
where $[\![\exists\, n'<n_s.\ \neg Option.is\text{-}none\ (devBC\ t\ n\ nid\ n');\ Option.is\text{-}none\ (devBC\ t\ n\ nid\ n_s)]\!] \Longrightarrow devExt\ t\ n\ nid\ n_s\ 0 = bc\ (\sigma_{the\ (devBC\ t\ n\ nid\ (GREATEST\ n'.\ n'<n_s\ \wedge\ \neg Option.is\text{-}none\ (devBC\ t\ n\ nid\ n')))}(t\ (GREATEST\ n'.\ n'<n_s\ \wedge\ \neg Option.is\text{-}none\ (devBC\ t\ n\ nid\ n'))))$
$|\ [\![\neg\ (\exists\, n'<n_s.\ \neg Option.is\text{-}none\ (devBC\ t\ n\ nid\ n'));\ Option.is\text{-}none\ (devBC\ t\ n\ nid\ n_s)]\!] \Longrightarrow devExt\ t\ n\ nid\ n_s\ 0 = []$
$|\ \neg\ Option.is\text{-}none\ (devBC\ t\ n\ nid\ n_s) \Longrightarrow devExt\ t\ n\ nid\ n_s\ 0 = bc\ (\sigma_{the\ (devBC\ t\ n\ nid\ n_s)}(t\ n_s))$
$|\ \neg\ Option.is\text{-}none\ (devBC\ t\ n\ nid\ (n_s + Suc\ n')) \Longrightarrow devExt\ t\ n\ nid\ n_s\ (Suc\ n') = bc\ (\sigma_{the\ (devBC\ t\ n\ nid\ (n_s\ +\ Suc\ n'))}(t\ (n_s + Suc\ n')))$
$|\ Option.is\text{-}none\ (devBC\ t\ n\ nid\ (n_s + Suc\ n')) \Longrightarrow devExt\ t\ n\ nid\ n_s\ (Suc\ n') = devExt\ t\ n\ nid\ n_s\ n'$
$\langle proof \rangle$
termination $\langle proof \rangle$

lemma *devExt-same*:
assumes $\forall\, n'''>n'.\ n''' \leq n'' \longrightarrow Option.is\text{-}none\ (devBC\ t\ n\ nid\ n''')$
and $n' \geq n_s$
and $n''' \leq n''$
shows $n''' \geq n' \Longrightarrow devExt\ t\ n\ nid\ n_s\ (n'''-n_s) = devExt\ t\ n\ nid\ n_s\ (n'-n_s)$
$\langle proof \rangle$

lemma *devExt-bc*[*simp*]:
assumes $\neg\ Option.is\text{-}none\ (devBC\ t\ n\ nid\ (n'+n''))$
shows $devExt\ t\ n\ nid\ n'\ n'' = bc\ (\sigma_{the\ (devBC\ t\ n\ nid\ (n'+n''))}(t\ (n'+n'')))$

⟨*proof*⟩

lemma *devExt-greatest*:
assumes $\exists\, n'''<n'+n''.\ \neg\ Option.is\text{-}none\ (devBC\ t\ n\ nid\ n''')$
and $Option.is\text{-}none\ (devBC\ t\ n\ nid\ (n'+n''))$ **and** $\neg\ n''=0$
shows $devExt\ t\ n\ nid\ n'\ n'' = bc\ (\sigma_{the\ (devBC\ t\ n\ nid\ (GREATEST\ n'''.\ n'''<(n'+n'')\ \wedge\ \neg Option.is\text{-}none\ (devBC\ t\ n\ nid\ n'''$
$(GREATEST\ n'''.\ n'''<(n'+n'')\ \wedge\ \neg Option.is\text{-}none\ (devBC\ t\ n\ nid\ n'''))))$
⟨*proof*⟩

lemma *devExt-shift*: $devExt\ t\ n\ nid\ (n'+n'')\ 0 = devExt\ t\ n\ nid\ n'\ n''$
⟨*proof*⟩

lemma *devExt-bc-geq*:
assumes $\neg\ Option.is\text{-}none\ (devBC\ t\ n\ nid\ n')$ **and** $n'\geq n_s$
shows $devExt\ t\ n\ nid\ n_s\ (n'-n_s) = bc\ (\sigma_{the\ (devBC\ t\ n\ nid\ n')}(t\ n'))$ (**is** *?LHS = ?RHS*)
⟨*proof*⟩

lemma *his-bc-empty*:
assumes $(n',nid')\in his\ t\ n\ nid$ **and** $\neg(\exists\, n''<n'.\ \exists\, nid''.\ (n'',nid'')\in his\ t\ n\ nid)$
shows $bc\ (\sigma_{nid'}(t\ n')) = []$
⟨*proof*⟩

lemma *devExt-devop*:
$prefix\ (devExt\ t\ n\ nid\ n_s\ (Suc\ n'))\ (devExt\ t\ n\ nid\ n_s\ n') \vee (\exists\, b.\ devExt\ t\ n\ nid\ n_s\ (Suc\ n') = devExt\ t\ n\ nid\ n_s\ n'\ @\ [b]) \wedge \neg\ Option.is\text{-}none\ (devBC\ t\ n\ nid\ (n_s + Suc\ n')) \wedge \|the\ (devBC\ t\ n\ nid\ (n_s + Suc\ n'))\|_{t\ (n_s\ +\ Suc\ n')} \wedge n_s + Suc\ n' \leq n \wedge mining\ (\sigma_{the\ (devBC\ t\ n\ nid\ (n_s\ +\ Suc\ n'))}(t\ (n_s + Suc\ n')))$
⟨*proof*⟩

abbreviation *devLgthBC* **where** $devLgthBC\ t\ n\ nid\ n_s \equiv (\lambda n'.\ length\ (devExt\ t\ n\ nid\ n_s\ n'))$

theorem *blockchain-save*:
fixes $t{::}nat\Rightarrow cnf$ **and** n_s **and** sbc **and** n
assumes $\forall\, nid.\ honest\ nid \longrightarrow prefix\ sbc\ (bc\ (\sigma_{nid}(t\ (\langle nid \rightarrow t\rangle_{n_s}))))$
and $\forall\, nid\in actDn\ (t\ n_s).\ length\ (bc\ (\sigma_{nid}(t\ n_s))) < length\ sbc$
and $PoW\ t\ n_s\geq length\ sbc + cb$
and $\forall\, n'<n_s.\ \forall\, nid.\ \|nid\|_{t\ n'} \longrightarrow length\ (bc\ (\sigma_{nid} t\ n')) < length\ sbc \vee prefix\ sbc\ (bc\ (\sigma_{nid}(t\ n')))$
and $n\geq n_s$
shows $\forall\, nid \in actHn\ (t\ n).\ prefix\ sbc\ (bc\ (\sigma_{nid}(t\ n)))$
⟨*proof*⟩

end

end

References

[1] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley West Sussex, England, 1996.

[2] Diego Marmsoler. Towards a theory of architectural styles. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*, pages 823–825. ACM, ACM Press, 2014.

[3] Diego Marmsoler. Dynamic architectures. *Archive of Formal Proofs*, July 2017. http://isa-afp.org/entries/DynamicArchitectures.html, Formal proof development.

[4] Diego Marmsoler. Hierarchical specication and verication of architecture design patterns. In *Fundamental Approaches to Software Engineering - 21th International Conference, FASE 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, 2018.