A Theory of Architectural Design Patterns

Diego Marmsoler

March 1, 2018

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

locale *singleton* = *dynamic-component cmp active*
  for *active* :: *'id* ⇒ *cnf* ⇒ *bool* (‖-‖_ [*0*,*110*]*60*)
  and *cmp* :: *'id* ⇒ *cnf* ⇒ *'cmp* (*σ*_(-) [*0*,*110*]*60*) +
assumes *alwaysActive*: ⋀*k*. ∃ *id*. ‖*id*‖$_k$
  and *unique*: ∃ *id*. ∀ *k*. ∀ *id'*. (‖*id'*‖$_k$ ⟶ *id* = *id'*)
begin

definition *the-singleton* ≡ *THE id*. ∀ *k*. ∀ *id'*. ‖*id'*‖$_k$ ⟶ *id'* = *id*

lemma *the-unique*:
  fixes *k*::*cnf* **and** *id*::*'id*
  assumes ‖*id*‖$_k$
  shows *id* = *the-singleton*
⟨*proof*⟩

lemma *the-active*[*simp*]:
  fixes *k*
  shows ‖*the-singleton*‖$_k$
⟨*proof*⟩

lemma *lNact-active*[*simp*]:
  fixes *cid t n*
  shows ⟨*the-singleton* ← *t*⟩$_n$ = *n*
  ⟨*proof*⟩

lemma *lNxt-active*[*simp*]:
  fixes *cid t n*
  shows ⟨*the-singleton* → *t*⟩$_n$ = *n*
⟨*proof*⟩

lemma *assI*[*intro*]:
  fixes *t n a*
  assumes *φ* (*σ*$_{the\text{-}singleton}$(*t n*))
  shows *eval the-singleton t t' n* (*ass φ*) ⟨*proof*⟩

lemma *assE*[*elim*]:
fixes *t n a*
assumes *eval the-singleton t t′ n* (*ass φ*)
shows *φ* (*σ*$_{\textit{the-singleton}}$(*t n*)) ⟨*proof*⟩

lemma *evtE*[*elim*]:
fixes *t id n a*
assumes *eval the-singleton t t′ n* (*evt γ*)
shows ∃ *n′*≥*n*. *eval the-singleton t t′ n′ γ*
⟨*proof*⟩

lemma *globE*[*elim*]:
fixes *t id n a*
assumes *eval the-singleton t t′ n* (*glob γ*)
shows ∀ *n′*≥*n*. *eval the-singleton t t′ n′ γ*
⟨*proof*⟩

lemma *untilI*[*intro*]:
fixes *t*::*nat* ⇒ *cnf*
and *t′*::*nat* ⇒ *′cmp*
and *n*::*nat*
and *n′*::*nat*
assumes *n′*≥*n*
and *eval the-singleton t t′ n′ γ*
and ⋀*n″*. ⟦*n*≤*n″*; *n″*<*n′*⟧ ⟹ *eval the-singleton t t′ n″ γ′*
shows *eval the-singleton t t′ n* (*γ′* 𝔘 *γ*)
⟨*proof*⟩

lemma *untilE*[*elim*]:
fixes *t id n γ′ γ*
assumes *eval the-singleton t t′ n* (*until γ′ γ*)
shows ∃ *n′*≥*n*. *eval the-singleton t t′ n′ γ* ∧ (∀ *n″*≥*n*. *n″* < *n′* ⟶ *eval the-singleton t t′ n″ γ′*)
⟨*proof*⟩
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 $('id, 'evt)$ *subscription* = *sub* $'id$ $'evt$ | *unsub* $'id$ $'evt$

2.2 Publisher-Subscriber Architectures

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

notation *pb.imp* (**infixl** $\longrightarrow^p$ *10*)
notation *pb.or* (**infixl** $\vee^p$ *15*)
notation *pb.and* (**infixl** $\wedge^p$ *20*)
notation *pb.not* ($\neg^p$ - [*19*]*19*)
no-notation *pb.all* (**binder** $\forall_b$ *10*)
no-notation *pb.exists* (**binder** $\exists_b$ *10*)
notation *pb.all* (**binder** $\forall_p$ *10*)
notation *pb.exists* (**binder** $\exists_p$ *10*)

notation *sb.imp* (**infixl** $\longrightarrow^s$ *10*)
notation *sb.or* (**infixl** $\vee^s$ *15*)
notation *sb.and* (**infixl** $\wedge^s$ *20*)
notation *sb.not* ($\neg^s$ - [*19*]*19*)
no-notation *sb.all* (**binder** $\forall_b$ *10*)
no-notation *sb.exists* (**binder** $\exists_b$ *10*)
notation *sb.all* (**binder** $\forall_s$ *10*)
notation *sb.exists* (**binder** $\exists_s$ *10*)

abbreviation *the-publisher* :: $'pid$ **where**
the-publisher ≡ *pb.the-singleton*

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'' *sid* E e m
assumes $t \in$ *arch*
and *sbactive sid* $(t\ n)$
and *sub sid* E = *sbsb* (*sbcmp sid* $(t\ n)$)
and $n'' \geq n$
and $\nexists n'\ E'.\ n' \geq n \wedge n' \leq n'' \wedge$ *sbactive sid* $(t\ n') \wedge$ *unsub sid* E' = *sbsb*(*sbcmp sid* $(t\ n')$) $\wedge\ e \in E'$
and $e \in E$
and $(e,m) \in$ *pbnt* (*pbcmp the-publisher* $(t\ n'')$)
and *sbactive sid* $(t\ n'')$
shows $(e,m) \in$ *sbnt* (*sbcmp 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* (*pbcmp the-publisher* k) = $(\bigcup sid{\in}\{sid.\ sbactive\ sid\ k\}.\ \{sbsb\ (sbcmp\ sid\ k)\})$
⟨*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* × *PROB*) *set*
axiomatization where *sbWF*: *wf sb*
typedecl *SOL*
consts *solve*:: *PROB* ⇒ *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* ⇒ *cnf* ⇒ *bool* (‖-‖_ [0,110]60)
and *bbcmp* :: *'bid* ⇒ *cnf* ⇒ *'BB* (σ_(-) [0,110]60)
and *ksactive* :: *'kid* ⇒ *cnf* ⇒ *bool* (‖-‖_ [0,110]60)
and *kscmp* :: *'kid* ⇒ *cnf* ⇒ *'KS* (σ_(-) [0,110]60)

and *bbrp* :: $'BB \Rightarrow ('kid, PROB\ set)\ subscription\ set$
and *bbcs* :: $'BB \Rightarrow (PROB \times SOL)\ set$
and *kscs* :: $'KS \Rightarrow (PROB \times SOL)\ set$
and *ksrp* :: $'KS \Rightarrow ('kid, PROB\ set)\ subscription$ +
fixes *bbns* :: $'BB \Rightarrow (PROB \times SOL)\ set$
and *ksns* :: $'KS \Rightarrow (PROB \times SOL)\ set$
and *bbop* :: $'BB \Rightarrow PROB\ set$
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*: $\bigwedge t\ t'\ bId\ p\ s.\ [\![t \in arch]\!] \Longrightarrow pb.eval\ bId\ t\ t'\ 0$
$(pb.glob\ (pb.ass\ (\lambda bb.\ (p,s){\in}bbns\ bb)$
$\longrightarrow^p (pb.evt\ (pb.ass\ (\lambda bb.\ (p,s) \in bbcs\ bb)))))$
and *bhvbb2*: $\bigwedge t\ t'\ bId\ kId\ P\ q.\ [\![t{\in}arch]\!] \Longrightarrow pb.eval\ bId\ t\ t'\ 0$
$(pb.glob\ (pb.ass\ (\lambda bb.\ sub\ kId\ P \in bbrp\ bb \wedge q \in P) \longrightarrow^p$
$(pb.evt\ (pb.ass\ (\lambda bb.\ q \in bbop\ bb)))))$
and *bhvbb3*: $\bigwedge t\ t'\ bId\ p\ .\ [\![t{\in}arch]\!] \Longrightarrow pb.eval\ bId\ t\ t'\ 0$
$(pb.glob\ (pb.ass\ (\lambda bb.\ p \in bbop(bb)) \longrightarrow^p$
$(pb.wuntil\ (pb.ass\ (\lambda bb.\ p{\in}bbop(bb)))\ (pb.ass\ (\lambda bb.\ (p,solve(p)){\in}bbcs(bb))))))$
and *bhvks1*: $\bigwedge t\ t'\ kId\ p\ P.\ [\![t{\in}arch;\ p = prob\ kId]\!] \Longrightarrow sb.eval\ kId\ t\ t'\ 0$
$(sb.glob\ ((sb.ass\ (\lambda ks.\ sub\ kId\ P = ksrp\ ks)) \wedge^s$
$(sb.all\ (\lambda q.\ (sb.pred\ (q{\in}P)) \longrightarrow^s (sb.evt\ (sb.ass\ (\lambda ks.\ (q,solve(q)) \in kscs\ ks)))))$
$\longrightarrow^s (sb.evt\ (sb.ass\ (\lambda ks.\ (p,\ solve\ p) \in ksns\ ks)))))$
and *bhvks2*: $\bigwedge t\ t'\ kId\ p\ P\ q.\ [\![t \in arch;p = prob\ kId]\!] \Longrightarrow sb.eval\ kId\ t\ t'\ 0$
$(sb.glob\ (sb.ass\ (\lambda ks.\ sub\ kId\ P = ksrp\ ks \wedge q \in P \longrightarrow (q,p) \in sb)))$
and *bhvks3*: $\bigwedge t\ t'\ kId\ p.\ [\![t{\in}arch;p = prob\ kId]\!] \Longrightarrow sb.eval\ kId\ t\ t'\ 0$
$(sb.glob\ ((sb.ass\ (\lambda ks.\ p{\in}ksop\ ks)) \longrightarrow^s (sb.evt\ (sb.ass\ (\lambda ks.\ (\exists P.\ sub\ kId\ P = ksrp\ ks))))))$
and *bhvks4*: $\bigwedge t\ t'\ kId\ p\ P.\ [\![t{\in}arch;\ p{\in}P]\!] \Longrightarrow sb.eval\ kId\ t\ t'\ 0$
$(sb.glob\ ((sb.ass\ (\lambda ks.\ sub\ kId\ P = ksrp\ ks)) \longrightarrow^s$
$(sb.wuntil\ (\neg^s\ (\exists_s\ P'.\ (sb.pred\ (p{\in}P') \wedge^s (sb.ass\ (\lambda ks.\ unsub\ kId\ P' = ksrp\ ks)))))$
$(sb.ass\ (\lambda ks.\ (p,solve\ p) \in kscs\ ks)))))$

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

— Assertions about connections.
and *conn1*: $\bigwedge k\ bid.\ bbactive\ bid\ k$
$\Longrightarrow bbns\ (bbcmp\ bid\ k) = (\bigcup kid{\in}\{kid.\ ksactive\ kid\ k\}.\ ksns\ (kscmp\ kid\ k))$
and *conn2*: $\bigwedge k\ kid.\ ksactive\ kid\ k$
$\Longrightarrow ksop\ (kscmp\ kid\ k) = (\bigcup bid{\in}\{bid.\ bbactive\ bid\ k\}.\ bbop\ (bbcmp\ bid\ k))$
begin
notation *pb.lNAct* ($\langle$- $\leftarrow$ -$\rangle$_)
notation *pb.nxtAct* ($\langle$- $\rightarrow$ -$\rangle$_)

3.2.1 The Blackboard Component

In the following we introduce an abbreviation for the unique blackboard component.

abbreviation *the-bb* ≡ *pb.the-singleton*

3.2.2 Knowledge Sources

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

definition *sKs*:: *PROB* ⇒ *'kid* **where**
sKs p ≡ (*SOME kid*. *p* = *prob kid*)

lemma *sks-prob*:
p = *prob* (*sKs p*)
⟨*proof*⟩

3.2.3 Verifying Blackboards

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.

theorem *pSolved*:
fixes *t* **and** *t'*::*nat* ⇒*'BB* **and** *p* **and** *t''*::*nat* ⇒*'KS*
assumes *t*∈*arch* **and**
∀ *n*. ∀ *p*∈*bbop*(*bbcmp the-bb* (*t n*)).
∃ *n'*≥*n*. *ksactive* (*sKs p*) (*t n'*)
shows
∀ *n*. *p*∈*bbop*(*bbcmp the-bb* (*t n*)) ⟶
(∃ *m*≥*n*. (*p*,*solve*(*p*)) ∈ *bbcs* (*bbcmp the-bb* (*t m*)))
— The proof is by well-founded induction over the subproblem relation *sb*
⟨*proof*⟩
end

end

4 A Theory of Blockchain Architectures

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

4.1 Additions for Dynamic Components

These additions should go to theory Configuration_Traces for the next version of the AFP.

context *dynamic-component*
begin

lemma *disjE3*: $P \vee Q \vee R \Longrightarrow (P \Longrightarrow S) \Longrightarrow (Q \Longrightarrow S) \Longrightarrow (R \Longrightarrow S) \Longrightarrow S$ ⟨*proof*⟩

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

lemma *nxtAct-eq*:
assumes $n' \geq n$
and $\|c\|_{t\ n'}$
and $\forall n'' \geq n.\ n''<n' \longrightarrow \neg\ \|c\|_{t\ n''}$
shows $n' = \langle c \rightarrow t \rangle_n$
⟨*proof*⟩

lemma *globEANow*:
fixes $c\ t\ t'\ n\ i\ \gamma$
assumes $n \leq i$
and $\|c\|_{t\ i}$
and *eval* $c\ t\ t'\ n\ (\Box\gamma)$
shows *eval* $c\ t\ t'\ i\ \gamma$
⟨*proof*⟩

abbreviation *lastAct-cond*:: *'id* ⇒ *trace* ⇒ *nat* ⇒ *nat* ⇒ *bool*
where *lastAct-cond* $c\ t\ n\ n' \equiv n'<n \wedge \|c\|_{t\ n'}$

definition *lastAct*:: *'id* ⇒ *trace* ⇒ *nat* ⇒ *nat* (⟨- ⇐ -⟩-)
where *lastAct* $c\ t\ n$ = (*GREATEST* n'. *lastAct-cond* $c\ t\ n\ n'$)

lemma *lastActEx*:
assumes $\exists n'<n.\ \|nid\|_{t\ n'}$
shows $\exists n'.$ *lastAct-cond nid* $t\ n\ n' \wedge (\forall n''.$ *lastAct-cond nid* $t\ n\ n'' \longrightarrow n'' \leq n')$
⟨*proof*⟩

lemma *lastAct-prop*:
assumes $\exists n'<n.\ \|nid\|_{t\ n'}$
shows $\|nid\|_{t\ (lastAct\ nid\ t\ n)}$ **and** *lastAct nid* $t\ n<n$
⟨*proof*⟩

lemma *lastAct-less*:
assumes *lastAct-cond nid* $t\ n\ n'$
shows $n' \leq \langle nid \Leftarrow t \rangle_n$
⟨*proof*⟩

lemma *lastActNxt*:
assumes $\exists n'<n.\ \|nid\|_{t\ n'}$
shows $\langle nid \rightarrow t \rangle_{\langle nid \Leftarrow t \rangle_n} = \langle nid \Leftarrow t \rangle_n$
⟨*proof*⟩

lemma *lastActNxtAct*:
assumes $\exists\, n'{\geq}n.\ \|tid\|_{t\ n'}$
and $\exists\, n'{<}n.\ \|tid\|_{t\ n'}$
shows $\langle tid \rightarrow t\rangle_n > \langle tid \Leftarrow t\rangle_n$
⟨*proof*⟩

lemma *lastActless*:
assumes $\exists\, n'{\geq}n_S.\ n'{<}n \wedge \|nid\|_{t\ n'}$
shows $\langle nid \Leftarrow t\rangle_n{\geq}n_S$
⟨*proof*⟩

end

4.2 Blockchains

A blockchain itself is modeled as a simple list.

type-synonym *'a BC* = *'a list*

abbreviation *max-cond*:: (*'a BC*) *set* ⇒ *'a BC* ⇒ *bool*
where *max-cond B b* ≡ *b* ∈ *B* ∧ (∀ *b'*∈*B*. *length b'* ≤ *length b*)

definition *MAX*:: (*'a BC*) *set* ⇒ *'a BC*
where *MAX B* = (*SOME b*. *max-cond B b*)

lemma *max-ex*:
fixes *XS*::(*'a BC*) *set*
assumes *XS* ≠ {}
and *finite XS*
shows ∃ *xs*∈*XS*. (∀ *ys*∈*XS*. *length ys* ≤ *length xs*)
⟨*proof*⟩

lemma *max-prop*:
fixes *XS*::(*'a BC*) *set*
assumes *XS* ≠ {}
and *finite XS*
shows *MAX XS* ∈ *XS*
and ∀ *b'*∈*XS*. *length b'* ≤ *length* (*MAX XS*)
⟨*proof*⟩

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

4.3 Blockchain Architectures

In the following we describe the locale for blockchain architectures.

locale *Blockchain = dynamic-component cmp active*
  for *active* :: *'nid ⇒ cnf ⇒ bool* (‖-‖- [0,110]60)
    and *cmp* :: *'nid ⇒ cnf ⇒ 'ND* (σ-(-) [0,110]60) +
  fixes *pin* :: *'ND ⇒ ('nid BC) set*
    and *pout* :: *'ND ⇒ 'nid BC*
    and *bc* :: *'ND ⇒ 'nid BC*
    and *mining* :: *'ND ⇒ bool*
    and *trusted* :: *'nid ⇒ bool*
    and *actTr* :: *trace ⇒ nat ⇒ 'nid set*
    and *actUt* :: *trace ⇒ nat ⇒ 'nid set*
    and *PoW*:: *trace ⇒ nat ⇒ nat*
    and *trNxt*:: *trace ⇒ nat ⇒ bool*
  defines *actTr t n* ≡ {*nid*. ‖*nid*‖$_{t\ n}$ ∧ *trusted nid*}
    and *actUt t n* ≡ {*nid*. ‖*nid*‖$_{t\ n}$ ∧ ¬ *trusted nid*}
    and *PoW t n* ≡ (*LEAST x*. ∀ *nid*∈*actTr t n*. *length* (*bc* (σ$_{nid}$(*t n*))) ≤ *x*)
    and *trNxt t n* ≡ (∃ *n'*≥*n*. *PoW t n'* > *PoW t n* ∧ (∀ *n''*>*n*. *n''* ≤ *n'* ⟶ ¬(∃ *nid*∈*actUt t n''*. ‖*nid*‖$_{t\ n''}$ ∧ *mining* (σ$_{nid}$(*t n''*))))) ∨ (∀ *n'*>*n*. ¬(∃ *nid*∈*actUt t n'*. ‖*nid*‖$_{t\ n'}$ ∧ *mining* (σ$_{nid}$(*t n'*))))
  assumes *consensus*: ⋀*kid t t' bc'*::('*nid BC*). ⟦*trusted kid*⟧ ⟹ *eval kid t t' 0* (□(*ass* (λ*kt*. *bc'* = (*if* (∃ *b*∈*pin kt*. *length b* > *length* (*bc kt*)) *then* (*MAX* (*pin kt*)) *else* (*bc kt*))) ⟶^{b}
      ○ (*ass* (λ*kt*.(¬ *mining kt* ∧ *bc kt* = *bc'* ∨ *mining kt* ∧ *bc kt* = *bc'* @ [*kid*])))))
    and *attacker*: ⋀*kid t t' bc'*. ⟦¬ *trusted kid*⟧ ⟹ *eval kid t t' 0* (□(*ass* (λ*kt*. *bc'* = (*SOME b*. *b* ∈ (*pin kt* ∪ {*bc kt*}))) ⟶^{b}
      ○ (*ass* (λ*kt*.(¬ *mining kt* ∧ *prefix* (*bc kt*) *bc'* ∨ *mining kt* ∧ *bc kt* = *bc'* @ [*kid*])))))
    and *forward*: ⋀*kid t t'*. *eval kid t t' 0* (□(*ass* (λ*kt*. *pout kt* = *bc kt*)))
    — At each time point a node will forward its blockchain to the network
    and *conn*: ⋀*k kid*. *active kid k*
      ⟹ *pin* (*cmp kid k*) = (⋃ *kid'*∈{*kid'*. *active kid' k*}. {*pout* (*cmp kid' k*)})
    and *act*: ⋀*t n*::*nat*. *finite* {*kid*::'*nid*. ‖*kid*‖$_{t\ n}$}
    and *actTr*: ⋀*t n*::*nat*. ∃ *nid*. *trusted nid* ∧ ‖*nid*‖$_{t\ n}$ ∧ ‖*nid*‖$_{t\ (Suc\ n)}$
    and *fair*: ⋀*t kid n*::*nat*. ⟦¬ *trusted kid*; *mining* (σ$_{kid}$(*t n*))⟧ ⟹ *trNxt t n*
    and *closed*: ⋀*t kid b n*::*nat*. ⟦*b* ∈ *pin* (σ$_{kid}$(*t n*))⟧ ⟹ ∃ *kid'*. ‖*kid'*‖$_{t\ n}$ ∧ *pout* (σ$_{kid'}$(*t n*)) = *b*
begin

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 kid*
  assumes ‖*kid*‖$_{t\ n}$
  shows *finite* (*pin* (*cmp kid* (*t n*))) ⟨*proof*⟩

lemma *nempty-input*:
  fixes *t n kid*
  assumes ‖*kid*‖$_{t\ n}$

shows *pin (cmp kid (t n))≠{}* ⟨*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*⟩

4.3.1 Component Behavior

lemma *bhv-tr-ex*:
fixes *t* **and** *t'*::*nat* ⇒ *'ND* **and** *tid*
assumes *trusted 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 $\neg\ mining\ (\sigma_{tid} t\ \langle tid \rightarrow t\rangle_n) \wedge bc\ (\sigma_{tid} t\ \langle tid \rightarrow t\rangle_n) = Blockchain.MAX\ (pin\ (\sigma_{tid} t\ \langle tid \Leftarrow t\rangle_n)) \vee mining\ (\sigma_{tid} t\ \langle tid \rightarrow t\rangle_n) \wedge bc\ (\sigma_{tid} t\ \langle tid \rightarrow t\rangle_n) = Blockchain.MAX\ (pin\ (\sigma_{tid} t\ \langle tid \Leftarrow t\rangle_n))$ @ [*tid*]
⟨*proof*⟩

lemma *bhv-tr-in*:
fixes *t* **and** *t'*::*nat* ⇒ *'ND* **and** *tid*
assumes *trusted 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 $\neg\ mining\ (\sigma_{tid} t\ \langle tid \rightarrow t\rangle_n) \wedge bc\ (\sigma_{tid} t\ \langle tid \rightarrow t\rangle_n) = bc\ (\sigma_{tid} t\ \langle tid \Leftarrow t\rangle_n) \vee mining\ (\sigma_{tid} t\ \langle tid \rightarrow t\rangle_n) \wedge bc\ (\sigma_{tid} t\ \langle tid \rightarrow t\rangle_n) = bc\ (\sigma_{tid} t\ \langle tid \Leftarrow t\rangle_n)$ @ [*tid*]
⟨*proof*⟩

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

lemma *bhv-tr-context*:
assumes *trusted tid*
and $\|tid\|_{t\ n}$
and $\exists\, n'{\geq}n_S.\ n'{<}n \wedge \|tid\|_{t\ n'}$
shows $prefix\ (bc\ (\sigma_{tid} t\ \langle tid \Leftarrow t\rangle_n))\ (bc\ (\sigma_{tid} t\ n)) \vee (\exists\, nid'.\ \|nid'\|_{t\ \langle tid \Leftarrow t\rangle_n} \wedge length\ (bc\ (\sigma_{nid'} t\ \langle tid \Leftarrow t\rangle_n)) \geq length\ (MAX\ (pin\ (\sigma_{tid} t\ \langle tid \Leftarrow t\rangle_n))) \wedge prefix\ (bc\ (\sigma_{nid'} t\ \langle tid \Leftarrow t\rangle_n))\ (bc\ (\sigma_{tid} t\ n)))$
⟨*proof*⟩

lemma *bhv-ut-context*:
assumes $\neg$ *trusted uid*
and $\|uid\|_{t\ n}$
and $\exists\, n'{\geq}n_S.\ n'{<}n \wedge \|uid\|_{t\ n'}$
shows (*mining* ($\sigma_{uid} t\ n$) $\wedge$ *prefix* (*bc* ($\sigma_{uid} t\ n$)) (*bc* ($\sigma_{uid} t\ \langle uid \Leftarrow t\rangle_n$) @ [*uid*])) $\vee$ $\neg$ *mining* ($\sigma_{uid} t\ n$) $\wedge$ *prefix* (*bc* ($\sigma_{uid} t\ n$)) (*bc* ($\sigma_{uid} t\ \langle uid \Leftarrow t\rangle_n$)) $\vee$
($\exists\, nid'.\ \|nid'\|_{t\ \langle uid \Leftarrow t\rangle_n} \wedge$ (*mining* ($\sigma_{uid} t\ n$) $\wedge$ *prefix* (*bc* ($\sigma_{uid} t\ n$)) (*bc* ($\sigma_{nid'} t\ \langle uid \Leftarrow t\rangle_n$) @ [*uid*]) $\vee$ $\neg$ *mining* ($\sigma_{uid} t\ n$) $\wedge$ *prefix* (*bc* ($\sigma_{uid} t\ n$)) (*bc* ($\sigma_{nid'} t\ \langle uid \Leftarrow t\rangle_n$))))
⟨*proof*⟩

4.3.2 Maximal Trusted Blockchains

abbreviation *mbc-cond*:: *trace* ⇒ *nat* ⇒ *'nid* ⇒ *bool*
where *mbc-cond t n nid* ≡ *nid*∈*actTr t n* ∧ (∀ *nid'*∈*actTr t n*. *length* (*bc* ($\sigma_{nid'}(t\ n)$)) ≤ *length* (*bc* ($\sigma_{nid}(t\ n)$)))

lemma *mbc-ex*:
fixes *t n*
shows ∃ *x*. *mbc-cond t n x*
⟨*proof*⟩

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

lemma *mbc-prop*:
shows *mbc-cond t n* (*MBC t n*)
⟨*proof*⟩

4.3.3 Trusted Proof of Work

An important construction is the maximal proof of work available in the trusted 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* ⇒ *nat* ⇒ *nat* ⇒ *bool*
where *pow-cond t n n'* ≡ ∀ *nid*∈*actTr t n*. *length* (*bc* ($\sigma_{nid}(t\ n)$)) ≤ *n'*

lemma *pow-ex*:
fixes *t n*
shows *pow-cond t n* (*length* (*bc* ($\sigma_{MBC\ t\ n}(t\ n)$)))
and ∀ *x'*. *pow-cond t n x'* ⟶ *x'*≥*length* (*bc* ($\sigma_{MBC\ t\ n}(t\ n)$))
⟨*proof*⟩

lemma *pow-prop*:
pow-cond t n (*PoW t n*)
⟨*proof*⟩

lemma *pow-eq*:
fixes *n*

assumes $\exists\, tid{\in}actTr\ t\ n.\ length\ (bc\ (\sigma_{tid}(t\ n))) = x$
and $\forall\, tid{\in}actTr\ t\ n.\ length\ (bc\ (\sigma_{tid}(t\ n))) \leq x$
shows $PoW\ t\ n = x$
⟨*proof*⟩

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

lemma *pow-less*:
fixes $t\ n\ nid$
assumes *pow-cond* $t\ n\ x$
shows $PoW\ t\ n \leq x$
⟨*proof*⟩

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

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

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

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

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

4.3.4 Trusted Next

lemma *pow-eq-trnxt*:
assumes $PoW\ t\ n = PoW\ t\ n'$

and *trNxt t n*
and $n' \geq n$
shows *trNxt t n'*
⟨*proof*⟩

lemma *trnxt-pow-gr*:
assumes *trNxt t n*
and ¬ *trusted nid*
and *mining* ($\sigma_{nid} t\ n'$)
and $\|nid\|_{t\ n'}$
and $n' > n$
shows $PoW\ t\ n' > PoW\ t\ n$
⟨*proof*⟩

4.3.5 Secure Blockchains

lemma *ut-src-tr*:
assumes *prefix sbc* (*bc* ($\sigma_{nid'} t$ ⟨*nid* ⇐ *t*⟩$_n$))
and *build*: *mining* ($\sigma_{nid} t\ n$) ∧ *prefix* (*bc* ($\sigma_{nid} t\ n$)) (*bc* ($\sigma_{nid'} t$ ⟨*nid* ⇐ *t*⟩$_n$) @ [*nid*]) ∨ ¬ *mining* ($\sigma_{nid} t\ n$) ∧ *prefix* (*bc* ($\sigma_{nid} t\ n$)) (*bc* ($\sigma_{nid'} t$ ⟨*nid* ⇐ *t*⟩$_n$))
and $PoW\ t\ n > length\ sbc \vee PoW\ t\ n = length\ sbc \wedge trNxt\ t\ n$
shows *Suc* (*length* (*bc* ($\sigma_{nid} t\ n$))) < $PoW\ t\ n \vee$ *Suc* (*length* (*bc* ($\sigma_{nid} t\ n$))) = $PoW\ t\ n \wedge trNxt\ t\ n \vee$ *prefix sbc* (*bc* ($\sigma_{nid} t\ n$))
⟨*proof*⟩

lemma *ut-src-ut-less*:
assumes ¬ *trusted nid*
and *Suc* (*length* (*bc* ($\sigma_{nid'} t$ ⟨*nid* ⇐ *t*⟩$_n$))) < $PoW\ t$ ⟨*nid* ⇐ *t*⟩$_n$
and ¬*mining* ($\sigma_{nid} t\ n$) ∧ *prefix* (*bc* ($\sigma_{nid} t\ n$)) (*bc* ($\sigma_{nid'} t$ ⟨*nid* ⇐ *t*⟩$_n$)) ∨ *mining* ($\sigma_{nid} t\ n$) ∧ *prefix* (*bc* ($\sigma_{nid} t\ n$)) (*bc* ($\sigma_{nid'} t$ ⟨*nid* ⇐ *t*⟩$_n$) @ [*nid*])
and $\exists n' \geq n_S.\ n' < n \wedge \|nid\|_{t\ n'}$
and $\|nid\|_{t\ n}$
shows *Suc* (*length* (*bc* ($\sigma_{nid} t\ n$))) < $PoW\ t\ n \vee$ *Suc* (*length* (*bc* ($\sigma_{nid} t\ n$))) = $PoW\ t\ n \wedge trNxt\ t\ n$
⟨*proof*⟩

lemma *ut-src-ut-eq*:
assumes ¬ *trusted nid*
and *Suc* (*length* (*bc* ($\sigma_{nid'} t$ ⟨*nid* ⇐ *t*⟩$_n$))) = $PoW\ t$ ⟨*nid* ⇐ *t*⟩$_n$
and *trNxt t* ⟨*nid* ⇐ *t*⟩$_n$
and ¬*mining* ($\sigma_{nid} t\ n$) ∧ *prefix* (*bc* ($\sigma_{nid} t\ n$)) (*bc* ($\sigma_{nid'} t$ ⟨*nid* ⇐ *t*⟩$_n$)) ∨ *mining* ($\sigma_{nid} t\ n$) ∧ *prefix* (*bc* ($\sigma_{nid} t\ n$)) (*bc* ($\sigma_{nid'} t$ ⟨*nid* ⇐ *t*⟩$_n$) @ [*nid*])
and $\exists n' \geq n_S.\ n' < n \wedge \|nid\|_{t\ n'}$
and $\|nid\|_{t\ n}$
shows *Suc* (*length* (*bc* ($\sigma_{nid} t\ n$))) < $PoW\ t\ n \vee$ *Suc* (*length* (*bc* ($\sigma_{nid} t\ n$))) = $PoW\ t\ n \wedge trNxt\ t\ n$
⟨*proof*⟩

lemma *sbc-pow*:
fixes *t*::*nat*⇒*cnf* **and** n_S **and** *sbc* **and** *n*
assumes ∀ *nid*. *bc* (σ_{nid}(*t* (⟨*nid* → *t*⟩$_{n_S}$))) = *sbc*

and *trNxt t* n_S
shows $n{\geq}n_S \Longrightarrow PoW\ t\ n > length\ sbc \vee PoW\ t\ n = length\ sbc \wedge trNxt\ t\ n$
⟨*proof*⟩

theorem *blockchain-save*:
fixes *t*::*nat*⇒*cnf* **and** n_S **and** *sbc* **and** *n*
assumes $\forall\, nid.\ bc\ (\sigma_{nid}(t\ (\langle nid \rightarrow t\rangle_{n_S}))) = sbc$
and *trNxt t* n_S
and *prems*:$n{\geq}n_S$
shows $n{\geq}n_S \Longrightarrow \forall\, nid.\ (trusted\ nid \wedge \|nid\|_{t\ n} \longrightarrow prefix\ sbc\ (bc\ (\sigma_{nid}(t\ n)))) \wedge (\neg trusted\ nid \wedge \|nid\|_{t\ n} \longrightarrow Suc\ (length\ (bc\ (\sigma_{nid}(t\ n)))) < PoW\ t\ n \vee Suc\ (length\ (bc\ (\sigma_{nid}(t\ n)))) = PoW\ t\ n \wedge trNxt\ t\ n \vee prefix\ sbc\ (bc\ (\sigma_{nid}(t\ n))))$
⟨*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*, pages 1–65, July 2017. 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.